Ausarbeitung zum Pipeline-Architekturstil

Alber Jonas 15444427, Schweitzer Tim 29844429

29. Juni 2025

Zusammenfassung

Der Pipeline-Architekturstil (Pipes-and-Filters) ist ein etabliertes Paradigma zur Strukturierung von Systemen in modulare, sequenzielle Verarbeitungsschritte. Aufgrund seiner konzeptionellen Einfachheit und Modularität eignet er sich besonders für datenflussorientierte Aufgaben.[1] Diese Ausarbeitung analysiert die Kernprinzipien dieses Stils, beleuchtet typische Anwendungsfelder und bewertet kritisch dessen Vor- und Nachteile. Ein besonderer Fokus liegt auf der Evolution von traditionellen, monolithischen Implementierungen hin zu modernen, verteilten Streaming-Paradigmen und den damit verbundenen neuen Herausforderungen. Die Untersuchung unterstreicht die anhaltende Relevanz des Musters für die Bewältigung komplexer Datenverarbeitungsaufgaben.

1 Einführung und Historischer Kontext

Der Pipeline-Architekturstil, auch als Pipes-and-Filters bekannt, zerlegt komplexe Funktionalität in eine Kette unabhängiger, sequenzieller Verarbeitungsschritte. Die Klarheit des Datenflusses macht ihn zu einer attraktiven Wahl für viele Probleme, insbesondere in der Datenverarbeitung. Trotz neuerer Architekturparadigmen hat der Stil seine Relevanz bewahrt und erlebt durch moderne Technologien eine Renaissance.

Die philosophische Grundlage ist das Prinzip "Teile und Herrsche". Anstatt ein Problem monolithisch zu lösen, wird es in kleinere, handhabbare Teilprobleme zerlegt, die von spezialisierten Komponenten – den Filtern – gelöst werden. Diese Vorgehensweise reduziert die kognitive Komplexität und fördert die Modularität und Wartbarkeit des Systems.[1]

Der Erfolg des Musters wird eindrücklich durch die Unix-Philosophie illustriert. Die Möglichkeit, einfache Kommandozeilenwerkzeuge (Filter) über das Pipe-Symbol (—) zu mächtigen Verarbeitungsketten zu verbinden, demonstriert die Kernstärke des Ansatzes.[1, 2] Die bekannte Anekdote, die ein komplexes Pascal-Programm von Donald Knuth zur Worthäufigkeitsanalyse einem prägnanten Unix-Shell-Skript von Doug McIlroy gegenüberstellt, verdeutlicht ein fundamentales Prinzip: Die Komposition einfacher, wiederverwendbarer Komponenten führt oft zu eleganteren und verständlicheren Lösungen als ein monolithischer Ansatz.[1]

2 Kernarchitektur: Filter, Pipes und Designprinzipien

Die Pipeline-Architektur basiert auf zwei fundamentalen Komponenten: **Filtern** als autonome Verarbeitungseinheiten und **Pipes** als unidirektionale Kommunikationskanäle, die den Datenfluss steuern.[1, 3]

2.1 Filter: Die Verarbeitungseinheiten

Filter sind die aktiven Komponenten, die Daten transformieren. Ihre Effektivität hängt von mehreren Kerneigenschaften ab:

- **Unabhängigkeit:** Filter operieren autonom und kennen ihre Nachbarn nicht. Sie kommunizieren nur über Pipes. Dies ermöglicht den Austausch oder die Änderung eines Filters, ohne andere Teile des Systems zu beeinflussen, solange der Datenvertrag der Pipe eingehalten wird.[1]
- **Zustandslosigkeit (Statelessness):** Idealerweise hängt die Ausgabe eines Filters nur von der aktuellen Eingabe ab. Diese Eigenschaft ist die fundamentalste Voraussetzung für horizontale Skalierbarkeit und Robustheit, da Anfragen einfach auf mehrere Instanzen verteilt oder im Fehlerfall wiederholt werden können. Ist Zustand unvermeidbar (z.B. bei der Aggregation von Daten über Zeitfenster), muss dieser extern verwaltet werden (etwa in einer Datenbank oder einem verteilten

Cache wie Redis), um den Filter selbst zustandslos zu halten. Dies führt jedoch neue Abhängigkeiten und potenzielle Engpässe durch Latenz und Konsistenzmanagement ein.[4]

- **Einzige Verantwortlichkeit (Single Responsibility):** Jeder Filter erfüllt genau eine klar definierte Aufgabe. Komplexe Logik wird auf mehrere spezialisierte Filter aufgeteilt, was die Wiederverwendbarkeit, Testbarkeit und Verständlichkeit erhöht.[1]

Es gibt vier primäre **Filtertypen**:[2, 1]

- **Producer (Quelle):** Initiiert den Datenfluss (z.B. aus einer Datei oder Datenbank) und hat keinen Eingang.
- **Transformer (Umwandler):** Empfängt Daten, modifiziert sie (z.B. Konvertierung, Anreicherung) und leitet sie weiter.
- **Tester (Filter i.e.S.):** Bewertet Daten anhand von Kriterien und leitet sie weiter, verwirft sie oder routet sie bedingt.
- **Consumer (Senke):** Bildet den Endpunkt der Pipeline (z.B. Speichern in einer Datenbank) und hat keinen Ausgang.

Abbildung 1: Grundlegende Struktur einer Pipeline-Architektur. Daten durchlaufen verschiedene Zustände, während sie von einer Quelle (Producer) durch verarbeitende Filter (Transformer, Tester) zu einer Senke (Consumer) fließen.

2.2 Pipes: Die Kommunikations- und Kopplungselemente

Pipes sind mehr als nur passive Kanäle; sie sind gerichtete, unidirektionale Kopplungspunkte, die Filter verbinden und entkoppeln.[3] Sie ermöglichen asynchrone Verarbeitung und puffern Daten zwischen Filtern, die mit unterschiedlichen Geschwindigkeiten arbeiten.

Die *Implementierung* von Pipes variiert stark: von In-Memory-Queues in Monolithen bis zu robusten Message Brokern (z.B. Apache Kafka, RabbitMQ) in verteilten Systemen. Letztere bieten zusätzliche Garantien wie Persistenz und Lastverteilung.[1] Entscheidender als das *Payload-Format* (z.B. JSON, XML) ist der **Datenvertrag** – die genaue Struktur und Semantik der Daten. Dieser stellt trotz der Entkopplung eine starke, aber oft implizite, logische Abhängigkeit dar und ist eine der Hauptfehlerquellen in solchen Systemen. Änderungen am Vertrag (Schema-Evolution) erfordern oft Anpassungen in allen nachfolgenden Filtern. Um diese "Vertragsbrüche"zu managen, setzen moderne Systeme auf explizite Mechanismen wie *Schema-Registries* (z.B. Confluent Schema Registry), die die Kompatibilität von Datenformaten versionieren und erzwingen können.

2.3 Zentrale Designprinzipien

- **Komposition:** Komplexe Verarbeitungsketten werden durch das Zusammensetzen einfacher, fokussierter Filter erstellt. Dies erlaubt eine iterative Entwicklung und Modifikation des Systems.[1]
- **Modularität und Wiederverwendbarkeit:** Unabhängige Filter mit klaren Schnittstellen können in verschiedenen Pipelines wiederverwendet werden, was Entwicklungszeit spart.
- **Unidirektionaler Datenfluss:** Der lineare Datenfluss vereinfacht das Verständnis und die Fehleranalyse. Er limitiert jedoch die Eignung für interaktive Systeme, die bidirektionale Kommunikation erfordern.[1, 2]

3 Anwendungsfelder des Pipeline-Stils

Der Stil eignet sich besonders für Aufgaben, die eine sequenzielle Datentransformation erfordern.[1]

- **ETL-Prozesse (Extrahieren, Transformieren, Laden):** Ein klassisches Anwendungsfeld. Ein Producer-Filter extrahiert Daten, Transformer- und Tester-Filter bereinigen, aggregieren und formatieren sie, und ein Consumer-Filter lädt sie in ein Zielsystem wie ein Data Warehouse.[6]
- **Compilerbau:** Der Quellcode durchläuft sequenziell Phasen wie lexikalische und syntaktische Analyse, Optimierung und Zielcode-Generierung, die jeweils als Filter modelliert werden können.[1]
- **Streaming-Datenverarbeitung:** Moderne Frameworks (z.B. Apache Flink, Kafka Streams) nutzen das Pipeline-Paradigma für die Echtzeitanalyse von Sensordaten, Klickströmen oder Finanztransaktionen. Apache Kafka dient dabei oft als hochperformante, skalierbare Pipe-Infrastruktur.[9]
- **Electronic Data Interchange (EDI):** Pipelines konvertieren und validieren Geschäftsdatenaustauschformate, wobei Filter Aufgaben wie Parsen, Schema-Validierung und Feld-Mapping übernehmen.[1]

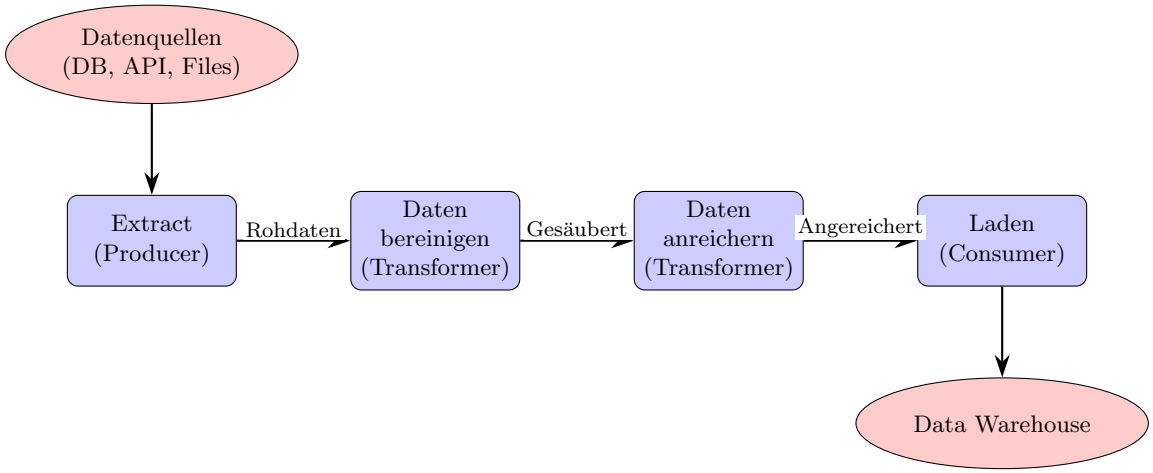

Abbildung 2: Anwendungsbeispiel einer Pipeline für einen ETL-Prozess (Extract, Transform, Load). Der Datenfluss ist klar von der Quelle über die Transformationsschritte zur Senke visualisiert.

4 Kritische Bewertung und Kompromisse

4.1 Vorteile: Analyse und Begründung

Die Hauptvorteile des Pipeline-Stils sind analytisch begründbar:

- **Einfachheit der Anwendungslogik:** Der lineare Datenfluss und die klare Verantwortungstrennung der Filter vereinfachen das Verständnis der Geschäftslogik und das Debugging im Vergleich zu stark vernetzten Architekturen.[1]
- **Modularität und Wiederverwendbarkeit:** Filter als "Blackboxes"mit definierten Verträgen ermöglichen Austausch und Wiederverwendung, was Entwicklungs- und Wartungskosten senkt.[11]
- **Flexibilität und Parallelisierbarkeit:** Die Entkopplung durch Pipes (insbesondere Message Queues) ermöglicht eine natürliche Parallelisierung, da mehrere Instanzen eines Filters konkurrierend Nachrichten verarbeiten können, was die Skalierbarkeit fördert.[4]

4.2 Herausforderungen und der Kompromiss der Komplexität

Die scheinbare Einfachheit des Pipeline-Stils birgt versteckte Komplexitäten. Gemäß dem Grundsatz der Ërhaltung der Komplexität"(Conservation of Complexity) verschwindet diese nicht, sondern verlagert sich lediglich – typischerweise von der Anwendungslogik in die Infrastruktur und den Betrieb. Die Entscheidung für eine Pipeline-Architektur, insbesondere in verteilter Form, ist daher ein bewusster Tausch.

Traditionelle, monolithische Implementierungen leiden unter geringer Skalierbarkeit und Fehlertoleranz, da das gesamte System als eine Einheit skaliert und ein Fehler in einem Filter die ganze Kette lahmlegen kann.[1]

Moderne, verteilte Implementierungen (z.B. mit Microservices) lösen diese Probleme, führen aber neue, gravierende Herausforderungen ein:

- **Massiv erhöhte operationelle Komplexität:** Die einfache Anwendungslogik wird mit dem Aufwand für Deployment, Konfiguration, Service Discovery und verteiltes Monitoring ërkauft". Dies erfordert spezialisierte Werkzeuge (z.B. Kubernetes) und tiefes Know-how.[1] Cloud-Plattformen wie Microsoft Azure bieten hierfür dedizierte, verwaltete Dienste (z.B. Azure Functions als Filter und Azure Service Bus als Pipe), welche die Implementierung erleichtern, die zugrundeliegende Komplexität aber nicht vollständig eliminieren.[5]
- **Fehlerbehandlung und Resilienz:** Dieser Aspekt wird oft vernachlässigt. Was passiert, wenn ein Filter in der Mitte der Kette ausfällt? Eine End-to-End-Transaktionalität existiert typischerweise nicht. Robuste Systeme erfordern daher explizite Resilienz-Muster:
  - **Dead Letter Queues (DLQ):** Um Nachrichten, die wiederholt nicht verarbeitet werden können, zur Analyse auszulagern und die Pipeline nicht zu blockieren.
  - **Wiederholungsmechanismen (Retries):** Temporäre Fehler (z.B. Netzwerkprobleme) müssen durch Retries (oft mit exponentiellem Backoff) abgefangen werden.
  - **Idempotenz:** Filter müssen so gestaltet sein, dass eine mehrfache Verarbeitung derselben Nachricht (z.B. nach einem Retry) nicht zu Datenkorruption führt.
- **Latenz vs. Durchsatz:** Der Stil ist auf hohen **Durchsatz** (Throughput) optimiert, nicht auf geringe **Latenz** (Latency). Die End-to-End-Latenz ist die Summe aller Verarbeitungs- und Netzwerklatenzen zwischen den Filtern. Dies macht den Stil für interaktive Echtzeitanwendungen, die sofortiges Feedback erfordern, ungeeignet.[2]
- **Observability und Datenkonsistenz:** Das Nachverfolgen von Datenflüssen über asynchrone Dienste hinweg erfordert verteiltes Tracing und zentrale Log-Aggregation. Die Gewährleistung von Datenkonsistenz (z.B. Exactly-Once-Semantik) über verteilte Pipes hinweg ist eine nichttriviale Herausforderung.[1]

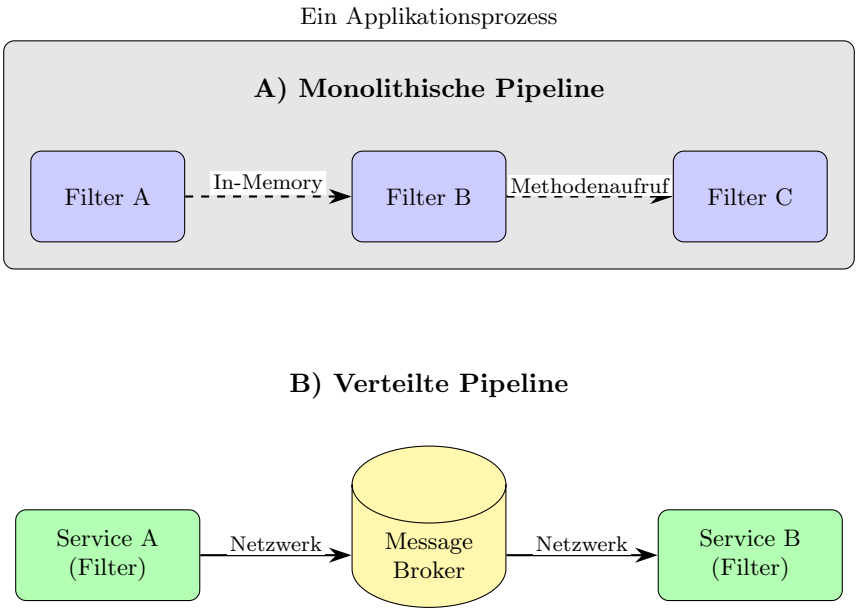

Abbildung 3: Vergleich einer monolithischen Implementierung (A), bei der alle Filter im selben Prozess laufen, mit einer verteilten Implementierung (B), bei der unabhängige Dienste über einen robusten Message Broker (Pipe) via Netzwerk kommunizieren.

5 Schlussfolgerung

Der Pipeline-Architekturstil bleibt ein wertvolles Muster, dessen Stärke in der Modularisierung komplexer Aufgaben in sequenzielle, unabhängige Verarbeitungsschritte liegt. Die Prinzipien der klaren Verantwortungstrennung und des linearen Datenflusses fördern die Verständlichkeit der Anwendungslogik.

Die kritische Analyse zeigt jedoch, dass diese Einfachheit oft mit einer signifikanten Verlagerung der Komplexität in den Betrieb und die Infrastruktur erkauft wird. Während monolithische Implementierungen an Skalierbarkeits- und Robustheitsgrenzen stoßen, lösen moderne, verteilte Ansätze diese Probleme, führen aber neue Herausforderungen in den Bereichen operationelle Komplexität, Resilienz und Latenz ein. Die Notwendigkeit von Idempotenz, Fehlerbehandlungsstrategien wie Dead Letter Queues und dem Management von Datenverträgen wird in solchen Systemen überlebenswichtig.

Die Effektivität des Pipeline-Stils ist somit weniger eine Frage des Musters selbst, sondern der bewussten architektonischen Entscheidung für einen Kompromiss: Er ist ideal für Systeme, die auf hohen Durchsatz und nicht auf niedrige Latenz optimiert sind und bei denen die Vorteile der modularen, unabhängigen Skalierung die Nachteile der erhöhten Betriebskomplexität überwiegen.

6 Praktische Umsetzung: Beispiel einer Datenverarbeitungs-Pipeline

6.1 Einleitung

Die Pipeline-Architektur ist ein leistungsfähiges Entwurfsmuster zur strukturierten Verarbeitung von Daten in sequenziellen Schritten. Sie zerlegt komplexe Prozesse in überschaubare, voneinander unabhängige Stufen, was die Modularität, Wartbarkeit und Wiederverwendbarkeit von Softwarekomponenten erheblich verbessert.

Die vorliegende Ausarbeitung erläutert die Prinzipien und Vorteile dieser Architektur anhand einer praktischen Beispielanwendung. Diese Anwendung demonstriert eine konfigurierbare Bildverarbeitungspipeline, bei der der Benutzer verschiedene Verarbeitungsstufen dynamisch aktivieren und deren Ergebnisse in Echtzeit verfolgen kann. Das vollständige Projekt, das dieser Ausarbeitung zugrunde liegt, ist als Open-Source-Anwendung im zugehörigen GitHub-Repository einsehbar. Für die Installation und Ausführung sind die Anweisungen in der `README.md`-Datei des Repositories zu befolgen.

6.2 Gesamtarchitektur

Um die Konzepte der Pipeline-Architektur zu verdeutlichen, ist dieser Ausarbeitung eine Beispielanwendung beigelegt. Diese Anwendung zeigt anschaulich, wie eine Pipeline aufgebaut ist und welche Vorteile sie bietet.

Abbildung 4 illustriert den Aufbau der Demo-Anwendung. Im Bereich „Pipeline Start“ befinden sich zwei Schaltflächen: „Open“ und „Update“. Mit „Open“ wird der Pipeline-Prozess gestartet, mit „Update“ kann dieser aktualisiert werden. Im Bereich „Pipeline Config“ kann der Nutzer verschiedene Pipeline-Segmente aktivieren und deren Parameter konfigurieren, um unterschiedliche Ergebnisse zu erzielen. Die verarbeiteten Bilder werden anschließend im Bereich „Ergebnis Ausgabe“ angezeigt, sodass der Benutzer die Pipeline-Ergebnisse live verfolgen kann.

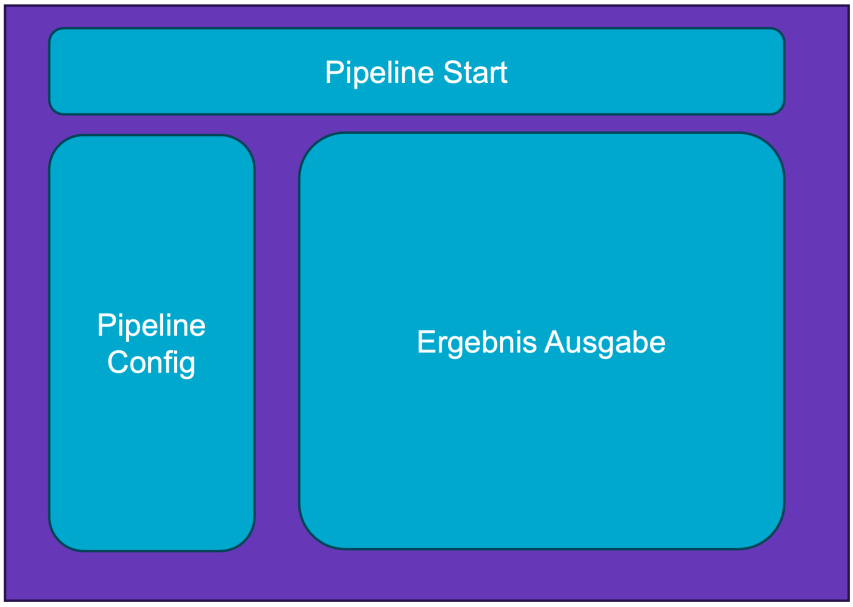

Abbildung 4: Aufbau der Demo-GUI.

Die GUI selbst ist nicht Teil der Pipeline, sondern nach dem MVC-Pattern aufgebaut. Sie nutzt lediglich die Pipeline-Architektur. Dadurch bleibt die Pipeline unabhängig von der GUI und kann auch an anderer Stelle wiederverwendet werden. Abbildung 5 verdeutlicht diese Trennung: Die GUI steuert die Pipeline lediglich an und zeigt die Ergebnisse an, ohne direkt in deren Logik einzugreifen.

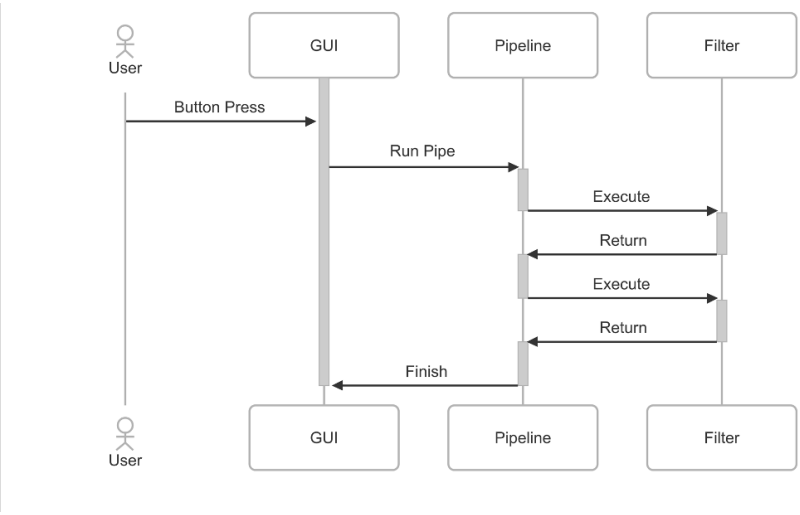

Abbildung 5: Kommunikation zwischen GUI und Pipeline.

Wird die Pipeline über den „Open“-Button gestartet, öffnet sich zunächst ein Dateiauswahldialog, in dem der Benutzer die zu verarbeitende Datei auswählt. Dies ist der Einstiegspunkt in die Pipeline, die aus den folgenden Stufen besteht:

1. **Loader:** In dieser ersten Stufe werden die Rohdaten von der Datenquelle (z.B. Bilddateien wie JPG, SVG etc.) eingelesen. Dazu gehört das Öffnen der Datei und das Parsen des Inhalts in eine interne Datenstruktur.

2. **Transformer:** Auf die Daten werden Filter angewendet. Das können einfache Bildbearbeitungsschritte wie Drehen oder Spiegeln sein, aber auch komplexere Transformationen wie eine Bildanalyse mittels neuronaler Netze, um Merkmale zu extrahieren oder zu klassifizieren.

3. **Tester:** Tester-Stufen prüfen, ob die Transformer korrekt angewendet wurden und die Ergebnisse den Erwartungen entsprechen.

4. **Consumer:** Die verarbeiteten Ergebnisse werden gespeichert (z.B. in einer Datei oder Datenbank) oder an nachfolgende Systeme weitergegeben, etwa an die GUI zur Anzeige.

6.3 Implementierung der Pipeline-Steuerung: Die Wrapper-Klasse

Die Implementierung einer Pipeline kann auf verschiedene Arten erfolgen, die sich hinsichtlich Flexibilität, Erweiterbarkeit und Entwicklungsaufwand unterscheiden. Gängige Ansätze sind:

- **Direkte Implementierung im Code:** Die Reihenfolge der Stufen wird direkt im Code festgelegt. Das ist für kleine Pipelines einfach, wird aber bei wachsender Komplexität schnell unübersichtlich und schwer wartbar.

- **Konfigurationsbasierter Ansatz:** Die Pipeline-Struktur wird in einer externen Datei (z.B. YAML, JSON) definiert und zur Laufzeit geladen. Das bietet hohe Flexibilität und ermöglicht Anpassungen ohne Code-Änderungen, erfordert aber zusätzlichen Aufwand für das Parsen und Validieren der Konfiguration.

- **Wrapper-Klasse:** Eine eigene Klasse kapselt die Verwaltung der Stufen (Hinzufügen, Reihenfolge) und die Ausführung der Pipeline. Das fördert Struktur, Kapselung und Wiederverwendbarkeit.

Für die Beispielanwendung ist die Wrapper-Klasse der beste Ansatz. Sie trennt die Pipeline von der GUI, macht sie aber dennoch leicht zugänglich und ermöglicht dynamische Änderungen. Die zentrale Idee ist, eine Klasse (z.B. `Pipeline`) zu verwenden, die eine Liste von Pipeline-Stufen verwaltet. Jede Stufe wird als `Stage` bezeichnet und leitet von einer abstrakten Basisklasse ab. Dadurch muss jede konkrete Stufe die Methode `process` implementieren, was sicherstellt, dass alle Stufen einheitlich behandelt und von der Pipeline-Klasse aufgerufen werden können.

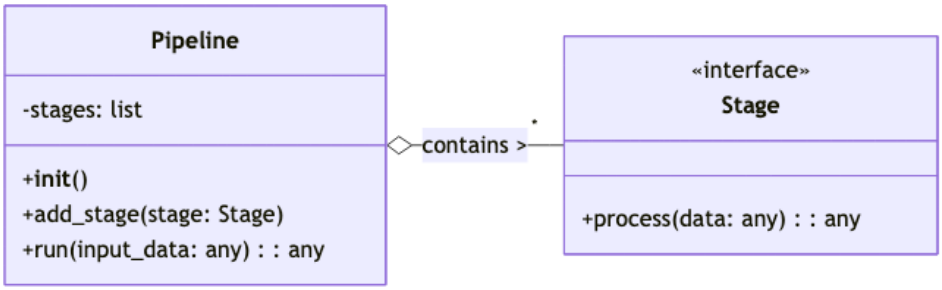

Abbildung 6: Klassendiagramm der Pipeline Wrapper-Klasse (`Pipeline`) und der abstrakten Stufenklasse (`Stage`).

Die Wrapper-Klasse prüft zudem, dass nur gültige Stufen hinzugefügt werden. Dies geschieht durch Typ-Prüfungen, wie im folgenden Python-Codeausschnitt für die Methode `add_stage` gezeigt:

Damit Änderungen in einer Stage nur gezielt nachfolgende Stages beeinflussen, ist es sinnvoll, eine Datenklasse für den Datentransfer zwischen den Stages zu definieren. In der Beispielanwendung übernimmt dies die `PipeDataClass`. Jede Stage erwartet eine Instanz dieser Klasse als Parameter in der `process`-Methode und gibt eine modifizierte Instanz zurück. So bleiben die Daten zwischen den Stufen konsistent und nachvollziehbar.

Da die Demo eine Bildbearbeitung demonstriert, muss zwischen globalen und lokalen Änderungen unterschieden werden. Globale Änderungen betreffen das gesamte Bild (z.B. Drehen), während lokale Änderungen (z.B. Bounding Boxes von neuronalen Netzen oder Poster, die per Arucomarker platziert werden) nicht nachfolgende Filter beeinflussen sollen. Globale Änderungen greifen direkt auf das `base_image_from_source` zu, während lokale Änderungen über `add_optional_layer` in eine Liste optionaler Layer eingetragen werden. Diese Layer werden in der Consumer-Stage zusammengeführt und als Ergebnis ausgegeben. Der Aufbau der `PipeDataClass` ist in Abbildung 7 dargestellt.

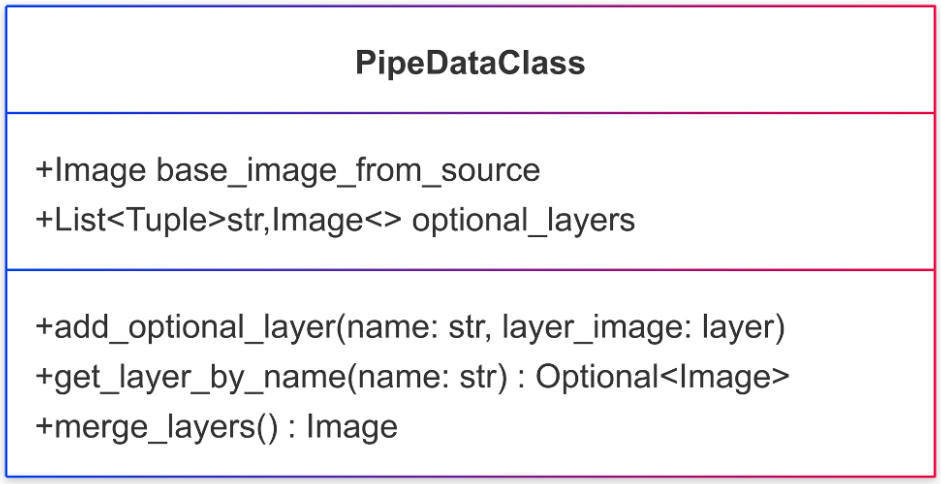

Abbildung 7: Aufbau der PipeDataClass

6.4 Factory Pattern zur dynamischen Komponentenauswahl

Ein weiteres Entwurfsmuster, das die Pipeline-Architektur sinnvoll ergänzt, ist das Factory Pattern. Es dient dazu, Objekte zu erzeugen, ohne die konkrete Klasse im Voraus festlegen zu müssen. Die Verantwortung für die Objekterzeugung übernimmt eine spezialisierte Factory-Klasse oder -Methode.

Das Factory Pattern entkoppelt den aufrufenden Code von der konkreten Implementierung der zu erzeugenden Objekte. Die Factory entscheidet anhand der übergebenen Parameter oder des Kontexts, welche Unterklasse instanziiert und zurückgegeben wird.

6.4.1 Anwendung im Datenlader

Im Beispiel kommt das Factory Pattern beim Laden der Bilddaten zum Einsatz. Bilder liegen in unterschiedlichen Formaten vor, von pixelbasierten Formaten wie PNG und JPEG bis zu vektorbasierten wie SVG. Damit nachfolgende Stufen immer mit einheitlichen Eingangsdaten arbeiten können, erfolgt eine Transformation auf ein Standardformat (hier: pixelbasiert). Anstatt die Logik zur Erkennung und Verarbeitung jedes Formats direkt in die Pipeline-Stufe einzubauen, wird eine `LoaderFactory` verwendet.

Der Ablauf ist typischerweise wie folgt:

1. Eine vorherige Pipeline-Stufe oder der Initialaufruf übergibt den Dateipfad oder eine Kennung der zu ladenden Daten an die zuständige Stufe.
2. Diese Stufe delegiert die Erzeugung des passenden Ladeobjekts an die `LoaderFactory` und übergibt relevante Informationen (z.B. Dateipfad oder -format).
3. Die `LoaderFactory` analysiert die Informationen (z.B. Dateiendung wie `.png` oder `.svg`).
4. Basierend darauf instanziiert die Factory das passende Ladeobjekt (z.B. `PNG_Loader` oder `SVG_Loader`), das eine gemeinsame Schnittstelle (`DataLoader`) implementiert.
5. Die Factory gibt das erzeugte Ladeobjekt an die aufrufende Pipeline-Stufe zurück.
6. Die Pipeline-Stufe verwendet das Ladeobjekt, um die Daten zu laden, ohne die konkrete Implementierung kennen zu müssen.

Abbildung 8 zeigt diesen Entscheidungsprozess. Die Factory prüft das angeforderte Format und wählt den passenden Loader aus. Ist kein passender Loader registriert oder existiert die Datei nicht, wird ein Fehler ausgelöst oder ein Null-Objekt zurückgegeben. Für die Pipeline ist dieser Prozess transparent – der Loader verhält sich wie eine normale Stage-Klasse.

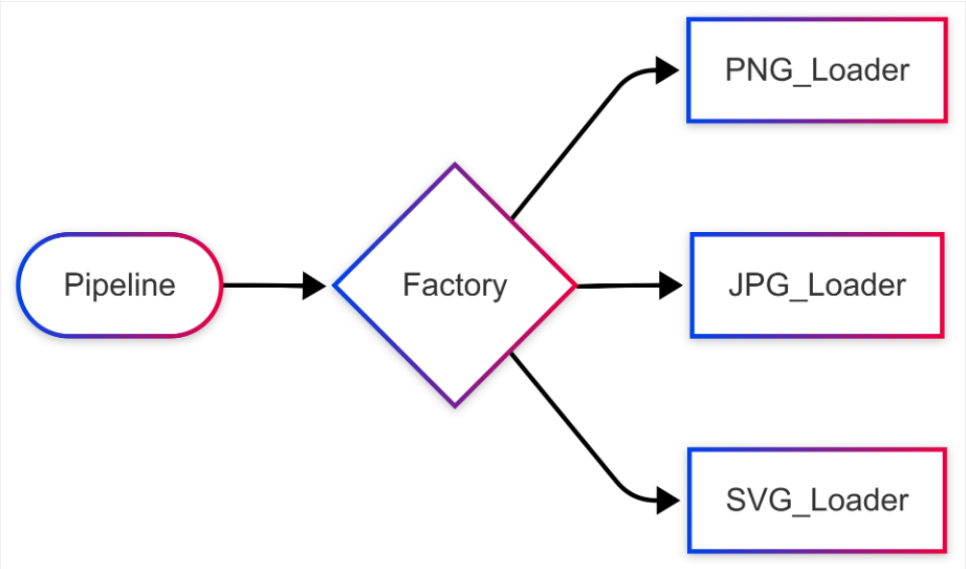

Abbildung 8: Entscheidungsdiagramm der LoaderFactory zur Auswahl des passenden Datenladers basierend auf dem Dateiformat.

6.4.2 Vorteile des Factory Patterns in der Pipeline

Das Factory Pattern bietet in diesem Kontext mehrere Vorteile:

- **Entkoppelung:** Die Pipeline-Stufe, die Daten lädt, ist von den konkreten Ladeimplementierungen entkoppelt und arbeitet nur mit der Factory und der abstrakten Schnittstelle.
- **Flexibilität und Erweiterbarkeit:** Neue Dateiformate können einfach durch Implementieren einer neuen `DataLoader`-Unterklasse und Anpassen der Factory unterstützt werden, ohne die Pipeline-Stufe zu ändern.

- **Zentralisierung der Erzeugungslogik:** Die Entscheidung, welcher Loader wann erstellt wird, ist an einer zentralen Stelle gebündelt.
- **Verbesserte Testbarkeit:** Loader-Klassen und Factory können isoliert getestet werden.
- **Wiederverwendbarkeit:** Factory und Loader-Klassen können auch in anderen Anwendungen oder Projekten genutzt werden.

Zusammengefasst verbessert das Factory Pattern die Struktur der Pipeline, indem es die Erzeugung von Komponenten wie Datenladern flexibler, wartbarer und erweiterbarer macht.

6.5 Pipeline mit Checkpoints

Ein Nachteil der klassischen Pipeline-Architektur ist ihre lineare Struktur: Die Verarbeitung erfolgt in fester Reihenfolge, und es ist schwierig, den Prozess zu unterbrechen oder an einem bestimmten Punkt neu zu starten. Das kann problematisch sein, wenn Fehler auftreten oder die Verarbeitung lange dauert und der Benutzer einzelne Stufen verändern möchte.

Um dieses Problem zu lösen, können Checkpoints eingesetzt werden. Sie ermöglichen es, den Zustand der Pipeline zu einem bestimmten Zeitpunkt zu speichern, sodass die Verarbeitung später an dieser Stelle fortgesetzt oder erneut durchgeführt werden kann. Das ist besonders nützlich, wenn z.B. ein neuronales Netz ein Bild analysiert und dieser Schritt viel Zeit beansprucht.

Wie oft und an welcher Stelle Checkpoints erzeugt werden, entscheidet der Entwickler. Im Demobeispiel wird vor jeder Stage im Transformer- oder Tester-Bereich ein Checkpoint erzeugt. So kann der Benutzer jederzeit Änderungen an beliebigen Teilen der Pipeline vornehmen. Durch Drücken des Update-Buttons in der GUI (siehe Abbildung 5) wird die Pipeline an der zuletzt unveränderten Stelle neu gestartet.

Literatur

[1] Mark Richards und Neal Ford. *Handbuch Moderner Softwarearchitektur: Architekturstile, Patterns und Best Practices*. O'Reilly Verlag GmbH Co. KG, 2020.

[2] CSSE6400. *Pipeline Architecture*. https://csse6400.uqcloud.net/handouts/pipeline.pdf

[3] O'Reilly. *Software Architecture with Python: Pipe and Filter architectures*. https://www.oreilly.com/library/view/software-architecture-with/9781786468529/ch08s04.html

[4] ResearchGate. *The Pipes and Filters Pattern: A Functional Parallelism Architectural Pattern for Parallel Programming*. https://www.researchgate.net/publication/221034471_The_Pipes_and_Filters_Pattern_A_Functional_Parallelism_Architectural_Pattern_for_Parallel_Programming

[5] Explore Azure Cloud. *Pipes and Filters Pattern in Azure - Part 1*. https://exploreazurecloud.com/pipes-and-filters-pattern-in-azure-part-1

[6] ProjectPro. *Top ETL Use Cases for BI and Analytics: Real-World Examples*. https://www.projectpro.io/article/etl-use-cases/768

[7] ResearchGate. *Distributed ETL Architecture for Processing and Storing Bigdata*. https://www.researchgate.net/publication/382522030_Distributed_ETL_Architecture_for_Processing_and_Storing_Bigdata

[8] WSO2 Docs. *Pipes and Filters EIP*. https://wso2docs.atlassian.net/wiki/spaces/EIP/pages/48791632/Pipes+and+Filters

[9] Acceldata. *Mastering Streaming Data Pipelines for Real-Time Data Processing*. https://www.acceldata.io/blog/mastering-streaming-data-pipelines-for-real-time-data-processing

[10] Dagster. *Data Pipeline Frameworks: Key Features 10 Tools to Know in 2024*. https://dagster.io/guides/data-pipeline/data-pipeline-frameworks-key-features-10-tools-to-know-in-2024

[11] Packt. *Software Architecture with C++: Architectural and System Design*. https://www.packtpub.com/fr-cy/product/software-architecture-with-c-9781838554590/chapter/architectural-and-system-design-6/section/pipes-and-filters-pattern-ch06lvl1sec35

[12] Dagster. *Data Pipeline*. https://dagster.io/guides/data-pipeline