Institut für Architektur von Anwendungssystemen
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3335

Framework für die Visualisierung von Datenqualität in Simulation-Workflows

Marcel Russ

Studiengang:	Softwaretechnik
Prüfer:	Jun.-Prof. Dr. Dimka Karastoyanova
Betreuer:	Dipl.-Math. Michael Reiter
Begonnen am:	30. April 2012
Beendet am:	30. Oktober 2012
CR-Klassifikation:	C.2.1, C.2.4, D.2.2, D.2.11, D.2.12 D.2.13, E.1, E.1, H.1.2, H.3.5, H.5.2, H.5.3, I.3.2, I.6.7, J.2

Zusammenfassung

In dieser Arbeit wurde ein Konzept für das Visualisieren von Datenqualität in Simulation-Workflows entwickelt und durch ein Java Framework (*Java Data Quality Visualization Framework*) realisiert. Es ermöglicht Wissenschaftlern laufende Simulationen, anhand visualisierter Datenqualitätswerte, zu überwachen und bei Bedarf in diese einzugreifen.
Das Framework unterstützt dabei mehrere Simulationen und unterscheidet die abgestuften Rechte der beteiligen Wissenschaftler – sowohl bei der Generierung der Visualisierungen, als auch bei der Weiterleitung von Steuerungsbefehlen an den Simulation-Workflow.
Während der Überwachung behält der Wissenschaftler die Kontrolle über die gesamte Visualisierungspipeline. Er kann dadurch interaktiv in die einzelnen Schritte eingreifen und die Visualisierung, seinen individuellen Anforderungen entsprechend, anpassen.
Des Weiteren unterscheidet das Konzept die unterschiedlichen Arten von Anzeigegeräten bei der Erstellung der Visualisierung. Dadurch ermöglicht es das Framework, für leistungsstarke Geräte komplexe Geometriemodelle und für leistungsschwache einfache Bilder zu erzeugen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

JDQCF	Java Data Quality Calculation Framework
JDQVisF	Java Data Quality Visualization Framework
JDQVisClient	Wissenschaftler mit einem Anzeigegerät
QoD	Datenqualität (Quality of Data)
WS	WebServices
WSDL	Web Service Description Language
XML	Extensible Markup Language
SOA	Service Oriented Architecture
WfMS	Workflow Management System

Farbenverzeichnis

1 Einleitung

In den letzten Jahren wurden Workflow-Technologien zur Durchführung von daten- und zeitintensiven Berechnungen unter dem Begriff *Scientific-Workflows* in die Wissenschaft übertragen. Ein Teilgebiet stellen dabei *Simulation-Workflows* dar, bei denen beispielsweise das Wachstum eines Tumors oder eines Knochens simuliert werden. Ein solcher *Simulation-Workflow* hat typischerweise eine lange Laufzeit und bearbeitet verschiedene Arten von Daten. Dabei hat die Qualität dieser Daten großen Einfluss auf das endgültige Simulationsergebnis. Eine schlechte Datenqualität führt mit großer Wahrscheinlichkeit zu ungenauen oder im schlechtesten Fall zu unbrauchbaren Ergebnissen.

Eine Möglichkeit repräsentative Ergebnisse zu erreichen, ist die Überwachung der laufenden Simulation anhand der Qualität ihrer Daten. Dabei werden die Daten unter verschiedenen Gesichtspunkten, wie die *Genauigkeit* oder *Vollständigkeit*, betrachtet. Für die Berechnung der Datenqualitätswerte wurde 2011 eine Framework (*Java Data Quality Framework* [1]) entwickelt.

In dieser Diplomarbeit wird ein Konzept für die Visualisierung von Datenqualitätswerten in *Simulation-Workflows* entwickelt und durch das *Java Data Quality Visualization Framework (JDQVisF)* realisiert. Es generiert aus den zuvor berechneten Datenqualitätswerten aussagekräftige Visualisierungen und erleichtert den Wissenschaftlern dadurch, die Überwachung der Datenqualität innerhalb der laufenden Simulation.

Das *JDQVisF* wird dabei verschiedene Arten von Anzeigegeräten unterstützen. Das bedeutet, dass je nach gewähltem Anzeigegerät unterschiedliche Visualisierungen generiert werden.
Zusätzlich wird das *JDQVisF* die Rolle des Wissenschaftlers bei der Generierung der Visualisierungen und der Benutzerinteraktionen berücksichtigen.
Damit der Wissenschaftler auf veränderte Datenqualitätswerte reagieren kann, bietet das *JDQVisF* eine Schnittstelle zur Steuerung der laufende Simulation an. Dies ermöglicht beispielsweise das Abbrechen einer Simulation bei schlechter Datenqualität.

Abbildung 1-1 zeigt die Einordnung des JDQVisF zwischen Simulation-Workflow und Wissenschaftler.

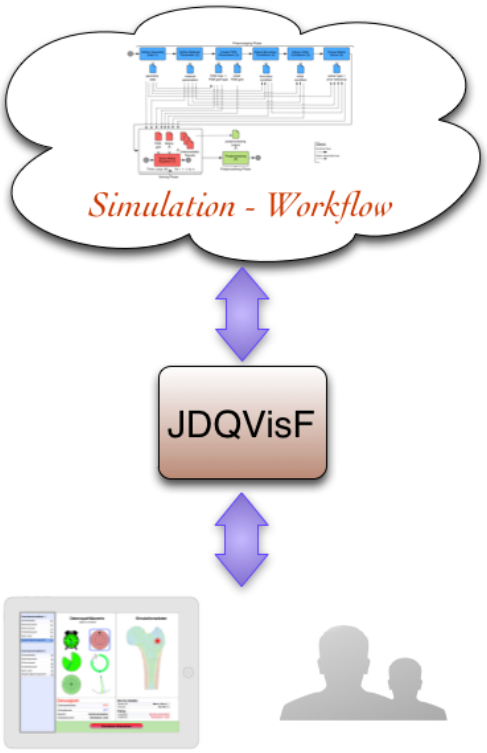


Abbildung 1-1: Zusammenhang Simulation-Workflow, Java Data Quality Visualization Framework und Wissenschaftler

1.1 Aufgabenstellung

Im Rahmen dieser Diplomarbeit sollen Konzepte für die Visualisierung von Datenqualität (QoD) in Simulation-Workflows erarbeitet und prototypisch realisiert werden. Dazu werden zunächst die Grundlagen für die visuelle Darstellung im Bereich der wissenschaftlichen QoD entwickelt. Anschließend werden die Rahmenbedingungen an ein entsprechendes Visualisierungs-Framework formuliert, beispielsweise der Zugriffssicherheit und der Registrierung am System. Dadurch wird sichergestellt, dass kein Unberechtigter die Daten einsehen oder schlimmstenfalls die Simulation manipulieren kann.

Nachdem die Grundlagen und Anforderungen aufgezeigt wurden, wird anhand derer eine Architektur vorgestellt und darauf basierend ein Framework, das folgende Funktionen erfüllt:

Bereitstellung verschiedener Visualisierungstypen – Je nach Eingabeformat werden unterschiedliche Diagrammarten und Darstellungsmöglichkeiten, z.B. Einzelwerte oder einen zeitlichen Verlauf zur Verfügung gestellt.

Generierung der Visualisierung von QoD-Daten – Die Eingabedaten werden anhand

ihres Formates, entweder QoD-Einzelwerte oder QoD + Simulationsdaten, unterschiedlich verarbeitet.

Möglichkeit zur Einbindung spezieller Visualisierungen – Abhängig vom späteren Ausgabegerät, z.B. einem Browser, Tablett oder Smartphone, gibt es unterschiedliche Ansprüche an die Visualisierung der Daten. Hierfür bietet das System die Möglichkeit über Plug-In-Schnittstellen geeignete Visualisierungen einzubinden.

Unterstützung verschiedener Arten von Darstellungsgeräten – iPad, Android-Tablet, eMail, usw.

Integration einer Autorisierungsfunktionalität des Benutzers – Die Benutzer werden bei der Anmeldung auf ihre Rechte hin überprüft.

Unterstützung bei der Steuerung der Simulation – Das Framework wird den Benutzer bei der Steuerung der Simulation unterstützen.

1.2 Verwandte Arbeiten

In [2] werden Theorien, Verfahren und Techniken zum Thema Datenqualität vorgestellt. Dabei werden verschiedene Fragestellungen wie die Relevanz von Datenqualität untersucht und an Beispielen deutlich gemacht.

In [1] wird ein Framework für die Berechnung von Datenqualität in Simulation-Workflows auf Basis konventioneller Workflow-Technologien entwickelt. Dazu analysiert es Simulationsdaten auf Basis von Metriken. Anschließend werden die Resultate durch eine Interpretationseinheit bewertet und somit Datenqualitätswerte erzeugt. Es unterstützt dabei neben der maschinelle Berechnung auch die Berechnung und Interpretation durch einen Menschen. Die Ergebnisse dieser Berechnungen dienen dem JDQVisF als Eingabedaten.

[42] ist eine Zusammenfassung einer Fachtagung der *National Center for Geographic Information and Analysis* (NCGIA) und befasst sich mit den Auswirkungen der Datenqualität in Geoinformationssystemen (GIS). In den Gesprächen werden die Rolle und der Nutzen einer Visualisierung für das Verständnis über die Qualität der GIS-Daten vorgestellt. Zudem wird gezeigt, wie wichtig die Zuverlässigkeit der Daten für die spätere Nutzung und Glaubwürdigkeit ist .

In [27] wird ein Werkzeug für die Visualisierung von allgemeinen Datenqualitätswerten vorgestellt. Es reduziert eine Datenmenge auf eine tabellarische Repräsentation. Diese soll helfen fehlende oder invalide Daten schnell zu erkennen, Inkonsistenzen aufzudecken oder verschiedene Datenversionen zu vergleichen. Dabei repräsentiert die Länge einer Tabellenzelle den Datenwert.

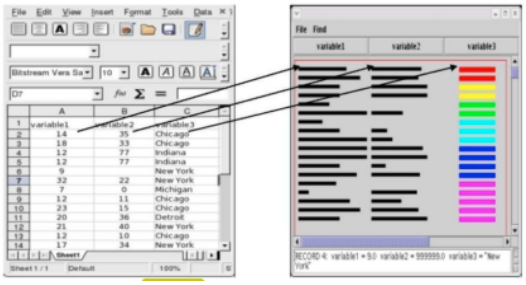


Abbildung 1-2: DaVis: Abbildung der Datenwerte auf die Länge der Tabellenzelle [27]

Leider konnte neben [27] keine weitere Referenz zu diesem Werkzeug gefunden werden.

Nach meinem besten Wissen und Gewissen gibt es zum Zeitpunkt der Erstellung dieser Diplomarbeit keine weiteren verwandten Arbeiten zum Thema Visualisierung von Datenqualitätswerten in Simulation-Workflows.

Alle Ideen und Konzepte die in dieser Arbeit vorgestellt werden, wurden durch regelmäßige Treffen mit dem Betreuer abgesprochen und abgestimmt.

1.3 Struktur der Arbeit

Nach diesem einleitenden Kapitel 1 werden in Kapitel 2 alle, für das Verständnis wichtigen, Grundlagen behandelt. Dazu zählen insbesondere die Definition der Begriffe *Daten* und *Datenqualität,* das *JDQCF*, sowie eine Einführungen in die Themen *Simulation-Workflow*, *Service Oriented Architecture* und *WebServices.*

Kapitel 3 befasst sich mit den allgemeinen Grundlagen des Themenbereichs Visualisierung. Hierzu gehören unter anderem die Ziele von Visualisierungen, alle wichtigen Definitionen zum Thema Visualisierung, die Visualisierungspipeline, die Anforderungen an eine gute Visualisierung, Einflussfaktoren auf die Visualisierung, Grundlegende Visualisierungstechniken. Zudem wird die Problemstellung des Findens passender Visualisierungen untersucht und Beispielvisualisierungen von Datenqualitätswerten aufgezeigt.

Kapitel 4 beschreibt die Anforderungen an das zu entwickelnde Visualisierungsframework. Da das *JDQVisF* zwei verschiedene Benutzergruppen besitzt, werden diese getrennt nach „*Wissenschaftler“* und „*Programmierer“* aufgestellt und formuliert.

In Kapitel 5 wird der konzeptionelle Entwurf des *JDQVisF* gezeigt. Grundsätzlich kann beim *JDQVisF* von einer logisch getrennten Dreischichtenarchitektur gesprochen werden, da es sich zwischen dem Simulation-Workflow und den Wissenschaftlern einordnet. Neben einer Visualisierungseinheit, wird es eine Komponente zur Benutzersteuerung enthalten. Bei dieser müssen sich die Wissenschaftler anmelden bevor sie die Datenqualität visualisiert auf ihr Endgerät erhalten.

Kapitel 6 beschreibt die technische Umsetzung der in Kapitel 5 erarbeiteten Konzepte. Dabei werden alle Komponenten und jeweils eine Erweiterung für das *JDQVisF* beschrieben.

Kapitel 7 beschließt diese Diplomarbeit durch die Beschreibung zukünftiger Arbeiten auf Grundlage des *JDQVisF*.

1.4 Notation und Schreibstil

Diese Diplomarbeit wird in deutsch verfasst. Einige Begriffe wie *Service Oriented Architecture*, *WebServices, Mapping oder Namespace* werden jedoch nicht übersetzt und unter der englischen Originalbezeichnung verwendet, da sie als allgemein anerkannt gelten.

In dieser Arbeit wird für die Bezeichnung von unbestimmten Personen wie *Wissenschaftler*, *Benutzer* und *Entwickler* die maskuline und feminine Form zusammengefasst und unter der maskulinen Form verwendet. Diese Konvention soll den Lesefluss gegenüber der ausgeschriebenen Form („*Wissenschaftler(in)*“) erleichtern und nicht diskriminierend sein.

2 Grundlagen

In diesem Kapitel werden die Grundlagen und Hintergrundinformationen gegeben, welche für die anschließenden Kapitel und das allgemeine Verständnis der späteren Konzepte von Bedeutung sind.

2.1 Daten

In diesem Kapitel wird zunächst der Begriff *Daten* definiert, bevor anschließend die verschiedenen Strukturen und deren Bedeutung erklärt werden. Am Ende des Kapitels wird ein Ausblick zur Visualisierung von Daten gegeben.

2.1.1 Definition des Begriffs Daten

Für den Begriff *Daten* sind in der Literatur unterschiedliche Definitionen verbreitet. Die verschiedenen Einsatzumgebung und Fachgebieten prägen die Definitionen maßgeblich. Beispielsweise spricht [2] von Daten, „*die Objekte der realen Welt darstellen“*. Dagegen werden in der Informatik Daten, „*als in erkennungsfähiger Form dargestellte Elemente einer Information, die in Systemen verarbeitet werden können.“* [4] definiert.

In Simulationen werden Informationen aus unterschiedlichen Fachbereichen mit unterschiedlicher Herkunft verarbeitet. In manchen beziehen sich Daten auf reale Objekte, beispielsweise bei der Simulation des Knochenwachstums, in anderen beziehen sie sich auf mathematische Modelle, die mögliche zukünftige Ereignisse, beispielsweise den Klimawandel der nächsten Jahre, prognostizieren. Damit dieses weite Feld der wissenschaftlichen Simulation nicht eingeschränkt wird und durch die enge Anbindung dieser Arbeit an [1], wird die Definition aus [1] übernommen.

Definition: *Daten*

Daten repräsentieren Informationen.

Durch diese Definition können *Daten* in einem sehr breiten Umfeld eingesetzt werden. Der Begriff umfasst demnach auch ein Bild, ein Stück Programmcode oder eine Textdatei welche durch [1] bewertet und anschließend visualisiert dargestellt werden können.

Die Singularform *Datum* beschreibt einen einzelnen Datenwert in einer Menge von Daten.

2.1.2 Struktur

Nach [2] lassen sich Daten anhand ihrer Struktur in drei verschiedene Klassen einteilen. S*trukturierte Daten* sind Daten bei denen jedes Datenelement einer festen Struktur zugeordnet ist. Beispielsweise enthalten relationale Tabellen strukturierte Daten.
Semistrukturierte Daten sind Daten deren Struktur flexibel ist. Das heißt, die Daten sind nicht an ein festes Schema gebunden und sind selbstbeschreibend. Als ein Beispiel für *semistrukturierte Daten* kann hier die Markupsprache XML genannt werden. Eine XML-Datei kann ein zu Grunde liegendes Schema haben, muss es aber nicht, solange es die grundsätzlichen Anforderungen an ein XML-Dokument erfüllt.
Unstrukturierte Daten sind Daten, die in natürlicher Sprache ausgedrückt werden und somit keiner speziellen Struktur zuzuordnen sind.

Je nach Klasse können unterschiedliche Visualisierungen sinnvoll sein. Zum Beispiel können strukturierte Daten leicht auf einer Skala abgebildet werden, da sie in direkter Beziehung zu dieser existieren. Wohingegen der Informationsgehalt aus unstrukturierten Daten vor einer Visualisierung herausgearbeitet werden muss. Die Problemstellung des Findens guter Visualisierungen wird in Kapitel 3.8 genauer betrachtet.

2.1.3 Änderungsrate

Daten lassen sich, neben der Struktur, entsprechend der Häufigkeit ihrer Änderungen in drei Klassen unterteilen [2].
Stabile Daten (*stable data*) sind Daten die sich mit hoher Wahrscheinlichkeit nicht ändern werden. Als Beispiel können hier wissenschaftliche Arbeiten genannt werden. Es kommen zwar stetig neue hinzu, die alten verbleiben aber unverändert. Die zweite Klasse sind die *langzeitbeständige Daten* (*Long-term-changing data*). Sie beinhaltet Daten, die sich nur sehr selten verändern. Adressen oder Telefonnummern sind typische Beispiele für Daten in dieser Klasse.
In der dritten Klasse liegen die sich *häufig ändernde Daten* (*frequently-changing data*). Sie zeichnen sich durch einer hohen Änderungsrate aus. Typische Beispiele sind Daten zur Temperaturangabe oder zu Stauinformationen.

Unter Berücksichtigung der Änderungsrate ergibt sich, dass für einige Daten die Datenqualität häufiger berechnet und untersucht werden muss, als für andere. Aus diesem Grund spielt die Änderungsrate eines Datums bei den Auswahl einer passenden Visualisierung eine wichtige Rolle. *Stabile* oder *langzeitbeständige* Daten können komplexer und aufwändiger visualisiert werden, als sich schnell verändernde Daten. Dem Betrachter bleibt für eine effektive Auswertung des Informationsgehaltes länger Zeit. Schnell verändernde Daten und damit auch schnell verändernde Visualisierungen hingegen, müssen den Informationsgehalt auf das Wesentliche reduzieren, um dem Betrachter einen schnellen und effizienten Zugang zu ermöglichen.

Die wichtigsten Anforderungen, die bei der Auswahl guter Visualisierungen beachtet werden müssen, werden in Kapitel 3.4 beschrieben.

2.1.4 Visualisierung von Daten

Der Durchbruch des Computers führte in der Wissenschaft zu dem sogenannten „Fourth Paradigm For Science“ [5]. Es beschreibt das computergestützte Berechnen sehr großer Simulationen mit teilweise sehr langer Laufzeit und riesiger Datenmengen als Ergebnis. Große wissenschaftliche Einrichtungen, wie beispielsweise das australische *Square Kilometre Array* [10], der Teilchenbeschleuniger LHC am *CERN* [11] oder *Pan-STARRS* [12], können am Tag leicht mehrere Petabyte an Daten erzeugen [5]. Der Umgang mit diesen Datenmengen stellt die heutige Wissenschaftler vor große Herausforderungen.

Ein Ansatz für eine effektive und effiziente Auswertung dieser Datenflut bieten Visualisierungen. Sie erlauben durch das Auswählen relevanter Daten, die ursprüngliche Datenmenge zum Teil drastisch zu verringern und somit das Simulationsziel in einem engeren Umfeld zu betrachten [2]. Visualisierungen ermöglichen den Wissenschaftlern einen vereinfachten Einstieg und bieten eine gemeinsame Kommunikationsgrundlage.

In Kapitel 3 werden dazu Konzepte, Methoden und Techniken vorgestellt.

2.2 Simulation

Dieses Kapitel gibt einen Einblick in wissenschaftliche Simulationen. Hierzu wird zunächst der Begriff *Simulation* definiert und mit einer FEM (*Finite Element Methode*) basierten Simulation ein Beispiel aufgezeigt.

Definition: *Simulation*

> *A simulation imitates one process by another process. In this definition, the term ‘process’ refers solely to some object or system whose state changes in time. If the simulation is run on a computer, it is called computer simulation.* [13]

In den Naturwissenschaften werden häufig so genannte FEM basierte Simulationen eingesetzt [28]. In diesen werden komplexe Differentialgleichungen numerisch für diskrete Zeitschritte gelöst. Dabei werden Ergebnisse für einen einzelnen Zeitschritt durch das Lösen von Matrizengleichungen berechnet. FEM basierte Simulation können sehr langläufig sein und produzieren dabei komplexe Datenstrukturen.

Nach [28] gliedert sich eine FEM-basierte Simulation in drei Phasen, welche jeweils wiederum in verschiedene Schritte unterteilt sind. (Abbildung 2-1 zeigt einen Beispielaufbau.)
Die erste Phase ist die *Vorverarbeitungsphase* (*Preprocessing Phase*). In dieser werden alle relevanten Eingabedaten für die spätere Berechnungsphase gesammelt. Dazu zählen beispielsweise das Geometriemodell, Materialparameter und FEM-Parameter.

In der *Berechnungsphase* (*Equation Solving Phase*) werden Matrix-Gleichungen, basierend auf den zuvor festgelegten Parametern, gelöst. Die Matrixgleichungen werden für jeden Zeitschritt gelöst, wobei die entstandenen Zwischenergebnisse wieder als Parameter in die Berechnung mit einfließen. Die Anzahl der Wiederholungen wird in diesem Beispiel vor der *Berechnungsphase* festgelegt.
In abschließenden *Auswertungsphase* (*Postprocessing Phase*) werden die finalen Ergebnisse ausgewertet. Da durch FEM basierten Simulationen sehr große Datenmengen entstehen können, werden in dieser Phase Visualisierungen eingesetzt. Diese können die entstandene Datenmenge durch gezielte Abbildungen reduzieren und präsentieren. Visualisierungen helfen dadurch den Wissenschaftlern, die Ergebnisdaten in einem bstimmten Kontext zu betrachten.

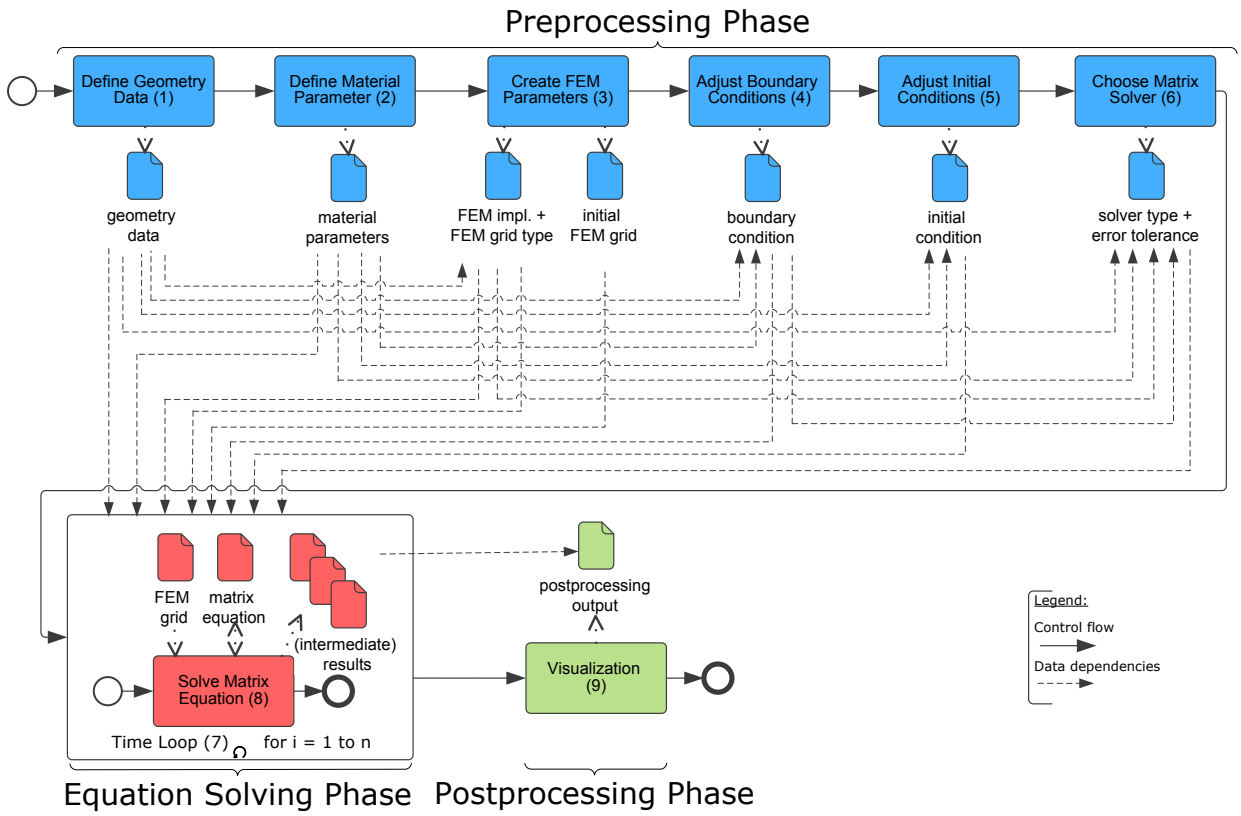


Abbildung 2-1: Aufbau einer FEM basierten Simulation [28]

2.3 Datenqualität

Die Berechnungen in komplexen und langläufige Simulationen können oft mehrere Wochen andauern. Um sicher zu stellen, dass die berechneten Endergebnisse korrekt sind, ist es wichtig die laufende Simulation zu überwachen. Dadurch kann bei Bedarf

frühzeitig in die laufende Simulation eingegriffen werden. Damit die Simulation repräsentierbare Ergebnisse liefert, werden insbesondere an die verwendeten Daten bestimmte Ansprüche gestellt. Beispielsweise darf die Genauigkeit der Zwischenergebnissen einen bestimmten Schwellenwert nicht unterschreiten, um die Korrektheit der Endergebnisse nicht zu beeinflussen [7].
Diese Ansprüche werden im Folgenden unter dem Begriff der *Datenqualität* zusammengefasst.

Dieses Kapitel gibt neben der Definition, einen Überblick über die wichtigsten Gesichtspunkte, so genannte *Dimensionen*, nach denen die Daten untersucht werden können.

2.3.1 Definition von Datenqualität

Die *International Association for Information and Data Quality* [6] setzt *Datenqualität* und *Informationsqualität* gleich und fasst sie unter dem Begriff „*Information quality*“ zusammen:

Information quality – (1) Consistently meeting all knowledge worker and end-customer expectations in all quality characteristics of the information products and services required to accomplish the enterprise mission (internal knowledge worker) or personal objectives (end customer). (2) The degree to which information consistently meets the requirements and expectations of all knowledge workers who require it to perform their processes. [6]

Da in dieser Arbeit von einem sehr allgemeinen *Daten*-Begriff ausgegangen wird und in Anlehnung an [1], wird der zweite Teil dieser Definition angepasst und *Datenqualität* für die Visualisierung in Simulation-Workflows wie folgt definiert:

Definition: *Datenqualität (Quality of Data, QoD)*
Das Ausmaß, in dem Daten die Anforderungen und Erwartungen der Wissenschaftler konsistent erfüllen.

Diese Definition erlaubt es Datenqualität skalenunabhängig betrachten zu können. Beispielsweile können Datenattribute auf Skalen der Form {*Gut*, *Schlecht*} oder auf Zahlenwerte von 0 bis 1 abgebildet und entsprechend interpretiert werden.

2.3.2 Dimensionen von Datenqualität

In [2] wird Datenqualität durch die sechs Dimensionen *Genauigkeit*, *Vollständigkeit*, *Aktualität*, *Rechtzeitigkeit*, *Schwankungsfreudigkeit* und *Konsistenz* beschrieben. Diese Dimensionen werden in dieser Arbeit übernommen und dienen den späteren Visualisierungen als Eingabedaten.

In den folgenden Abschnitten werden die einzelnen Datenqualitäts-Dimensionen genauer betrachtet und in Zusammenhang mit Simulation-Workflows gebracht.

2.3.2.1 Genauigkeit (Accuracy)

Bei wissenschaftlichen Simulationen spielt vor allem die *Genauigkeit* eine bedeutende Rolle. Sie beschreibt die Nähe zwischen einem Wert v und einem Wert v', wobei v' ein reales Objekt korrekt repräsentiert und v die Annäherung an diesen Wert [2]. Dabei bezieht sich die Genauigkeit nicht nur auf die mathematische Bedeutung, sondern kann auf jede Art von Attributen angewendet werden. Beispielsweise könnte der Name einer Person v' = John sein. Eine Annäherung v = Jhn ist damit unkorrekt. Nach [2] kann Genauigkeit auf eine syntaktische und semantische Ebene untersucht werden.
Die *syntaktische Genauigkeit* beschreibt die Nähe eines Wertes v zu den Elementen des dazugehörigen Definitionsbereiches. Beispielsweise ist der Wert v = 0.59 in einem Definitionsbereich [0...1] syntaktisch korrekt, der Wert v = -0.59 hingegen nicht.
Die *semantische Genauigkeit* hingegen beschreibt die Nähe eines Wertes v zu einem wahren Wert v'. In dem Beispiel würde v = 0.59 und v' = 0.6 eine hohe semantische Genauigkeit bedeuten, wenn die Zahlenwerte bei der Interpretation ähnlich groß sein sollen.

Bei komplexen Simulationen spielt in erster Linie die Genauigkeit bei mathematischen Berechnungen eine entscheidende Rolle. Wenn sich beispielsweise die Berechnung auf mehrere Berechnungseinheiten verteilt, kann es bei mangelnder Genauigkeit der Zwischenergebnisse leicht zu einer Fehlerfortpflanzung kommen und das Berechnungsergebnis maßgeblich beeinflussen oder sogar unbrauchbar machen.

2.3.2.2 Vollständigkeit (Completness)

Die Dimension *Vollständigkeit* beschreibt allgemein das Ausmaß, in dem Daten von ausreichender Breite, Tiefe und Umfang für die jeweilige Aufgabe vorhanden sind [8].
Sie taucht innerhalb eines Simulation-Workflow an unterschiedlichen Stellen auf. Beispielsweise kann direkt beim Start die Vollständigkeit der Ausgangsdaten untersucht werden.

2.3.2.3 Aktualität (Currency)

Aktualität beschreibt wie schnell die Daten bei einer Änderung aktualisiert werden [2]. Als ein Beispiel können bei einer verteilten Simulation die Eingabedaten der einzelnen Services genommen werden. Wenn ein Service Daten eines anderen Services als Eingabedaten erhält und das sofort nach jeder Datenänderung, dann ist die Aktualität hoch.

2.3.2.4 Rechtzeitigkeit (Timeliness)

Nach [2] beschreibt die *Rechtzeitigkeit* wie gegenwärtig die Daten für die aktuelle Aufgabe sind.
In Simulationen mit verteilten Berechnungseinheiten ist diese Dimension besonders wichtig: Wenn eine Berechnungseinheit einer darauf folgenden Einheit die Ergebnisse zu spät zur Verfügung stellt, kann es zu Engpässen und somit zur Verzögerung der gesamten Berechnung kommen.

2.3.2.5 Schwankungsfreudigkeit (Volatility)

Die Dimension *Schwankungsfreudigkeit* beschreibt die Frequenz mit der sich die Daten mit der Zeit verändern [2]. Dabei gibt es einen direkten Zusammenhang mit den in Kapitel 2.1.3 vorgestellten Klassen zur Änderungsrate. Bei einer hohen Änderungsrate, etwa bei dem Wert einer Aktie ist die Schwankungsfreudigkeit hoch. Bei stabilen Daten, etwa dem Geburtsdatum einer Person, entsprechend bei 0.

2.3.2.6 Konsistenz (Consistency)

Die Dimension *Konsistenz* erfasst die Verletzung von semantischen Regeln die über eine Menge von Datenelemente definiert ist [2]. Insbesondere beschreibt sie die Beziehung des Simulationsziel zu den zu Grunde liegenden Daten. Soll beispielsweise bei einer Simulation das Knochenwachstum eines 40-jährigen Mannes nach einer Sportverletzung berechnet werden und bekommt dazu die Knochendaten eines dreijährigen Mädchens als Eingabedaten, wird das Ergebnis unbrauchbar sein.

2.3.3 Java Data Quality Calculation Framework

Das *Java Data Quality Framework* (*JDQCF*) ist ein Framework für die Berechnung von Datenqualität in Simulation-Workflows. Es wird in [1] genauer beschrieben und soll hier nur kurz angesprochen werden.
Das JDQCF ordnet sich logisch vor dem hier zu entwickelnden *Java Data Quality Visualization Framework* (*JDQVisF*) in den Simulation-Workflow ein. Das JDQCF berechnet die Datenqualität und versendet die Daten an das JDQVisF. Es liefert somit die Eingabedaten für die Visualisierung. Neben den Datenqualitätswerten kann das JDQCF zusätzlich die originalen Simulationswerte versenden oder eine Referenz auf diese geben. Sie können somit bei der Visualisierung berücksichtigt werden.

2.4 Service Oriented Architecture

In vielen Branchen werden Geschäftsprozesse mit Hilfe von Software umgesetzt. Dabei enthalten unterschiedliche Geschäftsprozesse häufig gleiche Teilprozesse und oder

Funktionen. *Service Oriented Architecture* (SOA) beschreibt einen Software-Architekturstil mit dem Ziel, diese Funktionen, anstatt durch prozessspezifische Lösungen, durch so genannte *Services* den einzelnen Geschäftsprozessen zur Verfügung zu stellen. Sie erhöht dadurch die Flexibilität der Gesamtarchitektur und die Wiederverwendbarkeit einzelner Komponenten. [18]

Abbildung 2-2 zeigt das Zusammenspiel der einzelnen Komponenten einer SOA durch das so genannte SOA Dreieck. Es enthält jeweils eine Komponente für das Veröffentlichen von Services, das Finden und das Verknüpfen des Service-Benutzers mit dem Service-Anbieter.

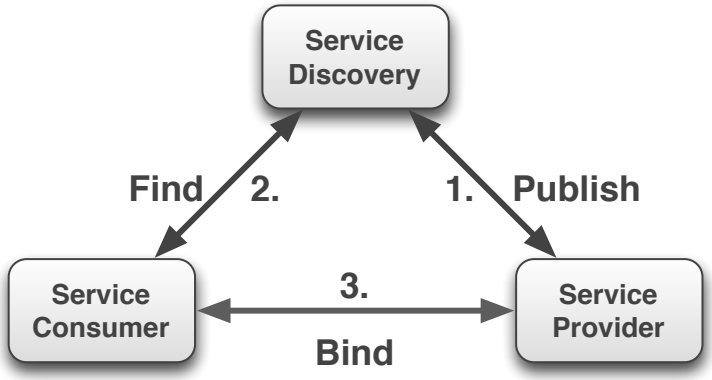


Abbildung 2-2: SOA Dreieck

Im ersten Schritt registriert ein Service-Anbieter (*Service Provider*) seine funktionalen und nichtfunktionalen Eigenschaften bei der *Service Discovery* (1). Im zweiten Schritt stellt ein möglicher Konsument (*Service Consumer*) an die *Service Discovery* eine Anfrage über die gewünschten funktionalen und nichtfunktionalen Anforderungen (2). Die *Service Discovery* wählt im nächsten Schritt einen passenden *Service Provider* aus und übergibt dem *Service Consumer* Metadaten (z.B. IP Adresse des *Service Provider*) mit deren Hilfe er sich an den *Service Provider* binden und dessen Funktionen verwenden kann (3).

2.4.1 WebServices

In diesem Abschnitt wird ein Überblick über die *WebService*-Technologie gegeben, die für das Verständnis dieser Arbeit wichtige Eigenschaften beschreibt. Für eine detaillierte Ausführung sei an dieser Stelle auf [18] und [33] verwiesen.

WebServices sind eine weitverbreitete Technologie für die Umsetzung einer SOA. Sie zeichnen sich durch die Verwendung standardisierter, plattformunabhängiger Technologien, wie das XML-Format und SOAP-Nachrichten, aus [1]. Ein WebService bietet dabei über standardisierte Schnittstellen ausgeschriebene und über das Internet zur

Verfügung gestellte Funktionalitäten an. Er realisiert dadurch eine Schnittstelle zwischen *Service Consumer* und *Service Provider*.

Das W*orld Wide Web Consortium* (*W3C*) definiert WebServices wie folgt:

Definition: *WebService*

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards. [29]

Aus der Definition lassen sich drei wichtige Eigenschaften von WebServices ableiten:

WebServices sind miteinander kombinierbar – WebServices zeichnen sich durch standardisierte Schnittstellen aus mit denen sie untereinander verknüpft werden können. Beispielsweise wird das JDQVisF als WebService implementiert werden, der von anderen WebServices verwendet werden kann und selber einen WebServices (z.B. das JDQCF) für die Dateneingabe benutzt.

WebServices sind lose gekoppelt – WebServices können auf unterschiedlichen Umgebungen laufen und in unterschiedlichen Programmiersprachen implementiert sein. Damit sie trotzdem miteinander kombinierbar sind, benutzen sie definierte Schnittstellen zur Kommunikation. Dadurch entsteht eine lose Kopplung und einzelne WebServices lassen sich leicht austauschen oder anpassen. Beispielsweise werden zur Visualisierung von Datenqualität, diese Werte zuerst vom JDQCF berechnet und anschließend über eine SOAP-Nachricht an das Visualisierungsframework übergeben.

WebService sind immer verfügbar – Das heißt, ein WebService ist 24/7 erreichbar und kann die Anfragen des *Service Consumer* mit einer ausreichenden Qualität bearbeiten. Für das hier zu entwickelnde Visualisierungsframework bedeutet das, dass ein Wissenschaftler zu jeder Tages- und Nachtzeit die Datenqualität visuell auf seinem Endgerät dargestellt bekommt und so seine laufenden Simulation überwachen kann.

2.4.1.1 WebService Beschreibung

Um die lose Kopplung eines *Service Consumer* und des *Service Provider* zu erreichen, werden standardisierte Beschreibungen der angebotenen Schnittstellen benötigt. Die

Web Service Description Language (*WSDL*) erlaubt es einem *Service Producer* seine funktionalen Eigenschaften plattformunabhängig zu definieren.
Die Definitionen von nichtfunktionalen Eigenschaften können durch den WS-Policys Standard [32] umgesetzt werden.

Da das Visualisierungsframework später als WebService zur Verfügung gestellt wird, wird im Folgenden der Aufbau eines WSDL-Dokumentes beschrieben.

WSDL

WSDL ist eine auf XML-basierte Metasprache für die Beschreibung der funktionalen Eigenschaften wie Methoden, Parameter und Austauschprotokolle eines WebService. Die syntaktischen Informationen eines WSDL Dokumentes wird durch die WSDL Spezifikation gegeben. Sie ist in Versionen v1.1 und v2.0 verfügbar. Da die neue Version v2.0 nur wenig verbreitet ist, beziehen sich alle Aussagen in dieser Arbeit auf die Version v1.1.

Abbildung 2-3 zeigt den Aufbau eines WSDL-Dokumentes. Es enthält die Beschreibungen der Schnittstellen, das Zugangsprotokoll und Details zum *Deployment* sowie alle Notwendigen Informationen für den Zugriff auf den WebService.

Ein WSDL-Dokument kann grundsätzlich in zwei Bereiche eingeteilt werden.
Im abstrakten Teil wird beschrieben *was* der WebService anbietet. Es werden die benötigten Datentypen (*types*), abstrakte Definitionen der Nachrichten für die Eingabe-, Ausgabe- und Fehlermeldungen (*messages*) und eine Menge von abstrakten Beschreibungen der Operationen (*operation*) die vom WebService unterstützt werden (*portType*) definiert [33].
Im konkreten Teil werden Informationen darüber hinzugefügt, *wie* mit dem WebService kommuniziert werden kann (*binding*) und *wo* dieser erreichbar ist (*service*). Das *binding* enthält konkrete Protokolle und Datenformate, um einen *portType* zu implementieren. Das Element *service* enthält eine Menge von individuellen „Endpunkten“ (*port*), die über eine Netzwerkadresse erreichbar sind und ein bestimmtest *binding* unterstützen. [33]

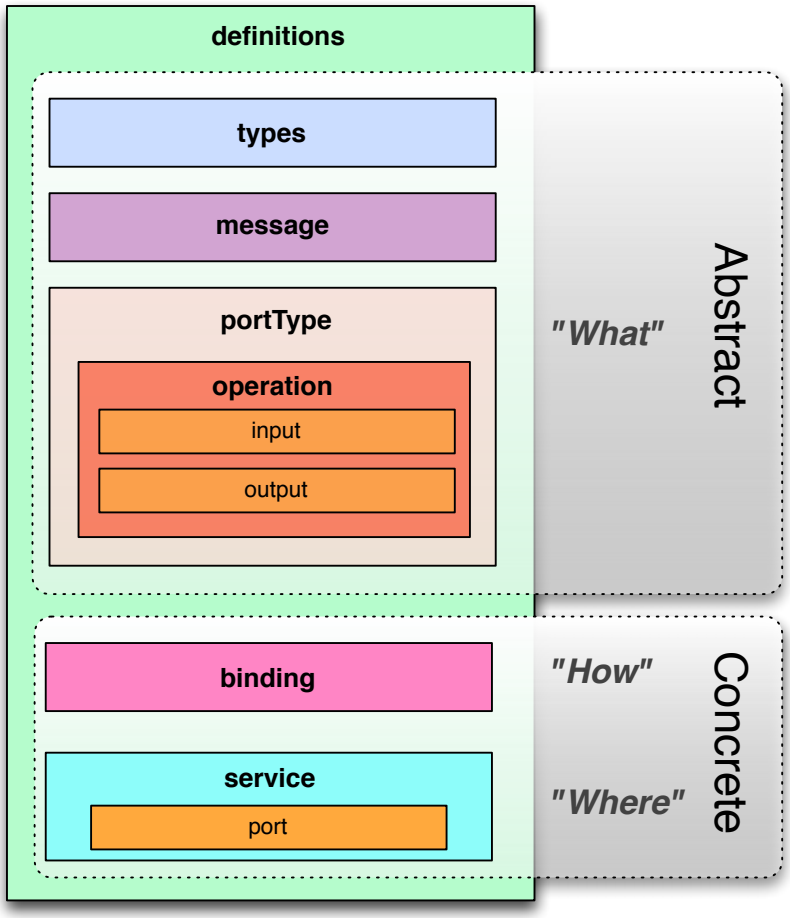


Abbildung 2-3: Aufbau eines WSDL-Dokuments nach der WSDL-Spezifikation v1.1 mit Kennzeichnung des abstrakten und konkreten Teils

2.5 Simulation-Workflow

Workflows beschreiben Kompositionen von voneinander abhängigen Aufgaben, die auf einem Computer unter Verwendung eines *Workflow-Management-Systems* (*WfMS*) ausgeführt werden [30]. Klassische Workflows dienen dazu, Geschäftsprozesse und IT zusammenzuführen. Dabei können die einzelnen Aufgaben durch WebServices realisiert sein, die auf unterschiedlichen Umgebungen und auf unterschiedlichen Rechnern ausgeführt werden.

Seit einigen Jahren werden bekannte Workflow-Technologien in die Wissenschaft unter dem Namen *Scientific-Workflows* übertragen [30]. Speziell im Gebiet der Simulationen bieten diese mehrere Vorteile. Durch ihre Hilfe kann beispielsweise das Auswerten der Simulationsergebnisse auf mehrere Wissenschaftler verteilt werden. Außerdem erleichtern sie den Umgang mit großen Datenmengen, wie sie in komplexen und langläufigen Simulationen entstehen können.

Abbildung 2-4 zeigt die Architektur eines Simulation-Workflow unter Verwendung von WebService-Technologien als eine Implementierung einer SOA und der Workflowsprache BPEL [30]. Der Service-Bus startet die verschiedenen Services für die Bearbeitung der Aufgaben innerhalb der Simulation. Das in [30] vorgestellte *Scientific WfMS* verwendet die konventionelle Workflow-Technologie, um Simulation-Workflows auf Basis kausalen Abhängigkeiten oder Datenabhängigkeiten zu modellieren. Genauer gesagt, kann ein Wissenschaftler durch die Verwendung von Workflow-Technologie ein Workflow-Modell erstellen, welches die Aufgaben definiert und die Reihenfolge in der diese verarbeitet werden müssen festlegt. Ein solches Modell dient als Vorlage aus der jeder Workflow instanziiert werden kann. Das bedeutet, dass jeder Workflow aus einem zugrunde liegenden Workflow-Modell heraus erstellt wird. Diese konkrete Workflowinstanz kann dann durch eine Workflow-Engine eines WfMS ausgeführt werden. [30]

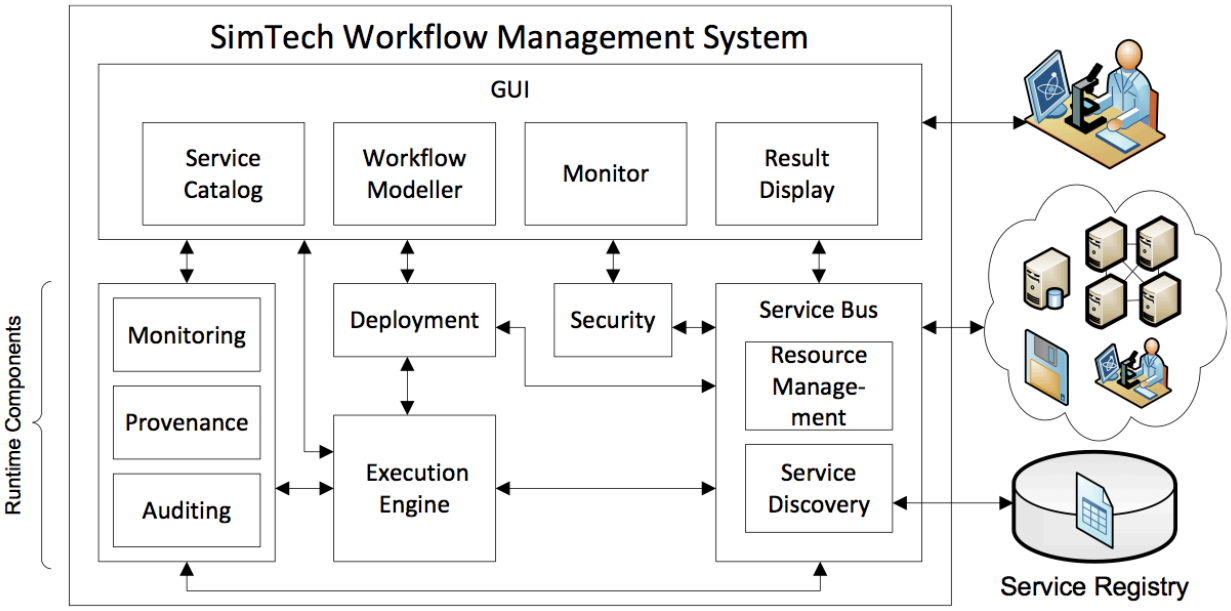


Abbildung 2-4: Architektur eines Simulation-Workflows auf Basis konventioneller Workflow-Technologie [30]

Simulation-Workflows, basierend auf kausalen Abhängigkeiten oder Datenabhängigkeiten, können langläufige Berechnungen und die Verwendung komplexen Datenstrukturen unterstützen. Dabei können durch konventionelle Workflow-Technologie einzelne WebServices zu einem übergeordneter Prozess zusammengestellt werden, der zur Bearbeitung einzelner Aufgaben eingesetzt werden kann. Diese Technik ermöglicht den Simulation-Workflow auf einer abstrahierten Ebene zu betrachten.

Abbildung 2-5 zeigt die generelle Architektur eines Simulations-Workflows. Diese ist in drei Schichten eingeteilt [28].

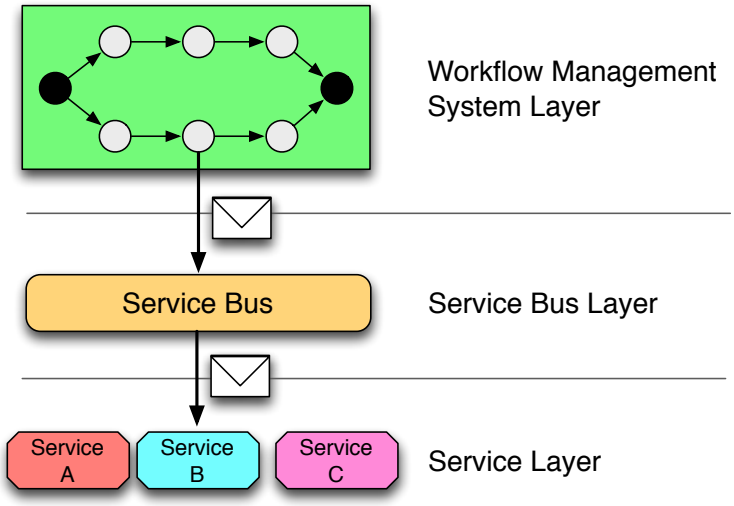


Abbildung 2-5: Generelle Architektur eines Simulation-Workflows [28]

Die WfMS Schicht ist für die Ausführung des Simulation-Workflows auf Basis konventioneller Workflow-Technologien verantwortlich. Sie verbindet einzelne WebServices um das Simulationsziel zu erreichen.
Die Service-Bus Schicht dient dem WfMS für den Aufruf der einzelnen Services. Das bedeutet, dass das WfMS nicht direkt die einzelnen Services aufruft. Der Service-Bus wählt einen passenden Service für die Bearbeitung der Aufgabe aus. Bei der Auswahl berücksichtigt er sowohl funktionale (z.B. die möglichen Operationen) als auch nichtfunktionale Anforderungen (z.B. Datenqualitätsanforderungen) der Services. Ist ein passender Service gefunden, übergibt der Service-Bus die Anfrage des Workflows an diesen. Nach der Bearbeitung gibt der Service-Bus das Ergebnis an den Workflow zurück.
Die Service-Schicht beschreibt alle Services die vom Workflow zusammengestellt und verwendet werden können. Jeder Service besitzt dazu standardisierte Schnittstellen und Beschreibungen seiner funktionalen und nichtfunktionalen Anforderungen. Zu den Funktionalen zählen zum Beispiel mögliche Operationen, Parameter und Datentypen. Zu den Nichtfunktionalen gehören Anforderungen wie Kosten, durchschnittliche Rechenzeit oder Anforderungen an die Datenqualität. Diese Schnittstellenbeschreibungen werde vom Service-Bus als Auswahlkriterien verwendet. [30]

3 Visualisierung von Datenqualität

Visualisierungen sollen den Betrachter bei der Auswertung einer Datenmenge unterstützen. Um dieses grundsätzliche Ziel zu erreichen, müssen diese Visualisierungen bestimmte Kriterien erfüllen. In diesem Kapitel werden hierzu Konzepte, Methoden und Techniken aufgezeigt und mögliche Visualisierungen für Datenqualitätswerte vorgestellt.

Im ersten Teil werden zunächst die grundsätzlichen Ziele von Visualisierungen gezeigt. Anschließend werden die Begriffe *Visualisierung*, *Informations-* und *Datenvisualisierung* und *Datenqualitätsvisualisierung* definiert. Danach wird die allgemeine Visualisierungspipeline vorgestellt und die drei Stufen *Filtering*, *Mapping* und *Rendering* erklärt.
Anschließend werden die Anforderungen *Expressivität*, *Effektivität* und *Angemessenheit* für gute Visualisierungen und die verschiedenen Einflussfaktoren gezeigt. Darauf aufbauend werden grundsätzliche Visualisierungstechniken vorgestellt und das Problem des *Findens passender Visualisierungen* beschrieben. Abschließend werden Techniken für die Visualisierung von Datenqualitätswerten vorgestellt.

3.1 Ziele der Visualisierung

Rolf Däßler begründet die Bedeutung der Visualisierungen wie folgt:

> „*Visualisierung entspricht der Neigung der menschlichen Spezies und unserer Kultur, visuelle Repräsentationsformen zu bevorzugen*.[...] *Nur ca. 13% der Information werden mit dem Gehör und 12% mit Hilfe anderer Sinnesorgane aufgenommen*.“ [3].

Das bedeutet, dass zwischen 60% und 80% der Informationen aus visuellen Eindrücken gewonnen werden [14].
Das Ziel jeder Visualisierung ist es, diese Erkenntnisse auszunutzen und die Informationen innerhalb abstrakter Daten verständlich wiederzugeben und dadurch eine effiziente Analyse zu ermöglichen. Speziell in wissenschaftlichen Simulationen, bei denen riesige Datenmengen entstehen können, ist die visuelle Repräsentation eine Möglichkeit die Auswertung und Bewertung zu vereinfachen.

In diesem Kapitel werden zunächst die allgemeinen Aufgaben der Visualisierung und ihre Ziele in ähnlichen Einsatzgebieten aufgezeigt. Anschließend werden diese allgemeingültig formuliert und auf die Visualisierung von Datenqualität in Simulation-Workflows übertragen.

Die Visualisierung ist immer auch Teil eines kreativen Prozesses, bei dem Strukturen und Zusammenhänge untersucht und kommuniziert werden. 1987 wurden von Mc Cormick, De Fanti und Brown die zwei Hauptaufgaben der Visualisierung beschrieben, welche auch heute noch aktuell sind.

Ergebnispräsentation – Visualisierungen sollen Ergebnisse präsentieren und somit *„...das Verständnis und die Kommunikation über die Daten und die zugrunde liegenden Modelle und Konzepte erleichtern*.“ [17]

Datenanalyse – Bilder sollen dem Betrachter helfen verborgene Zusammenhänge der Daten *„...nicht nur zu sehen, sondern auch zu erkennen, zu verstehen und zu bewerten [...] die allein aus Interpretation von Zahlenkolonnen nicht ableitbar wären.*“ [17]

Das Gebiet der *Informationsvisualisierung* geht noch einen Schritt weiter und definiert unabhängig vom Anwendungszweck und der Präsentationsform drei Ziele der Visualisierung von Daten:

1. „*Die Veranschaulichung und gegebenenfalls Vereinfachung von komplexen Prozessabläufen und Objektbeziehungen anhand von Symbolen, Diagrammen oder Animationen.*“ [3]

2. „*Die Vereinfachung des Zugangs zu Massendaten, z.B. durch Klassifikation und Datenstrukturierung*.“ [3]

3. „*Unterstützung bei der Analyse und Interpretation von Daten, z.B. Sichtbarmachung verborgener Trends, sowie Erleichterung der Mustererkennung.*“ [3]

Aus diesen beiden Ansätzen kann ein allgemeines Ziel für Visualisierungen in wissenschaftlichen Bereichen formuliert werden:

„*Es sollen die Analyse, das Verständnis und die Kommunikation von Modellen, Konzepten und Daten in der Wissenschaft erleichtert werden*.“ [16]

Davon abgeleitet, ergeben sich für die Visualisierung von Datenqualität in Simulation-Workflows folgende Ziele:

Analyse – Dem Wissenschaftler soll anhand der grafischen Darstellung der Datenqualitätswerte eine vereinfachte Analyse ermöglicht und so bei der Entscheidungsfindung, z.B. einem möglichen Eingriff in die laufende Berechnung, unterstützt werden. Dies kann zum Beispiel durch das Hervorheben kritischer Datenwerte oder das Filtern uninteressanter Metadaten geschehen.

Verständnis – Durch eine vom Betrachter leicht erfassbare visuelle Repräsentation von abstrakten Datenqualitätswerten, kann das Verständnis gesteigert werden. So kann je nach abgebildeter Skala, die jeweils geeignetste Darstellung gewählt werden.

Kommunikation – In Kooperation mit verschiedenen Domänenspezialisten, kann die grafischen Darstellung helfen, allen Beteiligten einen vereinfachten Zugang zu den Daten zu ermöglichen und kritische Komponenten zu erkennen und entsprechend anzupassen. Auch unbekannte Skalen können so leichter von Nicht-Domänenspezialisten erfasst und kritisch bewertet werden.

Daten – Da Datenqualitätswerte abstrakte Werte sind die nach [1] auf verschiedene Skalen abgebildet werden können, helfen grafische Darstellungen diese leichter zu erfassen und zu verarbeiten. Metaphern wie beispielsweise ein Wecker für die Dimension *Rechtzeitigkeit* kann die Bedeutung eines Zahlenwertes verdeutlichen.

3.2 Definitionen von Visualisierungen

In den folgenden Abschnitten werden die Begriffe *Visualisierung*, *Informations- und Datenvisualisierung* und *Datenqualitätsvisualisierung* definiert.

3.2.1 Visualisierung

Da das Visualisierungsframework in unterschiedlichen Bereichen eingesetzt wird und dabei die unterschiedlichsten Arten von Visualisierungen generiert werden müssen, wird von einem breiten Visualisierungs-Begriff ausgegangen und die Definition aus [3] übernommen.

Definition: *Visualisierung*

Der Prozess und das Ergebnis einer Darstellung oder Repräsentation von Informationen, die mit dem Auge wahrgenommen werden kann.

Diese grundsätzliche Definition erlaubt es einen breiten Einsatzbereich von Visualisierungen zu betrachten. So können Visualisierungen für reine Datenqualitätswerte, für Datenqualitätswerte in Kombination mit Simulationsdaten oder nur für Simulationsdaten generiert werden.

3.2.2 Informations- und Datenvisualisierung

Die Abgrenzung der Begriffe *Informationsvisualisierung* und *Datenvisualisierung* ist noch nicht einheitlich gelöst. Dies liegt vor allem an einer fehlenden, allgemein anerkannten Definition des Informationsbegriffes [17].

Aus diesem Grund, werden im Folgenden die beiden Begriffe als Synonyme behandelt und unter dem Begriff *Informationsvisualisierung* verwendet.

In der Literatur existieren unterschiedliche Definitionen von *Informationsvisualisierung.* Um im anschließenden Kapitel *Datenqualitätsvisualisierung* definieren zu können, wird hier eine Definition von [19] übernommen.

Definition: *Informationsvisualisierung*

Die Informationsvisualisierung nutzt Computergrafiken und -Interaktionen um dem Menschen bei der Lösung von Problemen zu unterstützen.

Die Informationsvisualisierung ist demnach ein Teilgebiet der Visualisierung. Sie hat das Ziel, große Informations- bzw. Datenmengen in ein, für Menschen leicht zu erfassendes Format zu bringen, um so die Entscheidungsfindung zu erleichtern.

3.2.3 Datenqualitätsvisualisierung

In Anlehnung an die *Informationsvisualisierung*, wird *Datenqualitätsvisualisierung* im Kontext wissenschaftlicher Simulationen wie folgt definiert:

Definition: *Datenqualitätsvisualisierung*

Die Datenqualitätsvisualisierung beschreibt die computergestützte visuelle Repräsentation von Datenqualitätswerten mit dem Ziel, Wissenschaftler bei der Analyse von Simulationen zu unterstützen.

Diese allgemeine Definition erlaubt es, Datenqualitätsvisualisierungen formatunabhängig zu betrachten. Dadurch sind einfache Bilder, genauso wie komplexe 3D Visualisierungen, Datenqualitätsvisualisierungen.
Durch den Verzicht einer Skala in dieser Definition, ermöglicht sie, Datenqualitätsvisualisierungen vom jeweiligen Einsatzgebiet und Fachbereich kontextabhängig zu gestalten.
Zudem berücksichtigt diese Definition die subjektiven Eigenschaften wie Sehschwächen oder Farbenblindheit eines Betrachters und kann diese in die Generierung der Datenqualitätsvisualisierung einfließen lassen.

3.3 Visualisierungspipeline

Der Visualisierungsprozess beschreibt das grundsätzliche Vorgehen bei der Generierung von Visualisierungen. Er besteht aus den drei grundsätzlichen Schritten: *Filtering*, *Mapping* und *Rendering*, die unter dem Begriff *Visualisierungspipeline* zusammengefasst werden.

Abbildung 3-1 zeigt den Ablauf bei der Generierung von Visualisierungen. Zuerst werden die Rohdaten aufbereitet, anschließend ein Geometriemodell erstellt und daraus im letzten Schritt ein Bild generiert.

Abbildung 3-1: Stufen der Visualisierungspipeline [17]

In den folgenden Abschnitten werden die einzelnen Stufen genauer betrachtet.

3.3.1 Datenaufbereitung (Filtering)

Die erste Stufe der Visualisierungspipeline ist die *Datenaufbereitung*. Sie realisiert eine *Daten-zu-Daten*-Abbildung [17]. Das heißt, dieser Schritt bekommt *Rohdaten* als Eingabe und bereitet diese für alle weiteren Schritte auf.

Das *Filtering* hat im Bereich der Datenqualitätsvisualisierung zwei Hauptaufgaben.
Zum einen soll die Datenmenge reduziert werden können, um dadurch Rechenkapazitäten einzusparen. Werden beispielsweise die originalen Simulationswerte für die aktuelle Problemlösung von den Wissenschaftlern nicht benötigt, können diese entfernt und nur die reinen Datenqualitätswerte an die Visualisierung weitergeleitet werden.
Zum anderen sollen, entgegen der englischen Bezeichnung, extern liegende Simulationsdaten geladen werden können, wenn sie für die Visualisierung benötigt werden.

Eine weitere Aufgabe eines Filters, könnte das Verwalten von Metadaten sein, welche für das Steuern der späteren Visualisierungsschritte benötigt werden. Falls es sich um ein verteiltes System handelt, können an dieser Stelle beispielsweise die Rücksendeadressen spezifiziert werden.

Als Ergebnis des *Filtering*, liegen aufbereiteten Daten vor, die an die nachfolgenden Schritte übergeben werden [17].

3.3.2 Mapping

Die zweite Stufe der Visualisierungspipeline, das *Mapping*, ist das Kernstück des Visualisierungsprozess. Sie realisiert eine *Daten-zu-Geometrie*-Abbildung. Dabei werden auch nicht-geometrische Daten, wie etwa Datenqualitätswerte, auf geometrische Primitive einschließlich der zugehörigen Attribute, wie Farbe und Textur, abgebildet [17].

In diesem Schritt entscheidet sich, wie die Daten später visuell repräsentiert werden. Die einzelnen Bestandteile eines guten Mappings werden in den nachfolgenden Kapiteln ausführlich behandelt.

3.3.3 Bildgenerierung (Rendering)

Die letzte Stufe der Visualisierungspipeline ist die *Bildgenerierung*. In diesem Schritt werden aus den abstrakten Geometriedaten die später angezeigten Bilder generiert. Die Bildgenerierung beschreibt eine Abbildung von Geometrie- auf Bilddaten. Nach der Bilderzeugung folgt die Ausgabe auf einem Anzeigegerät.

Abbildung 3-2 fasst die Visualisierungspipeline aus Sicht des Datenflusses zusammen.

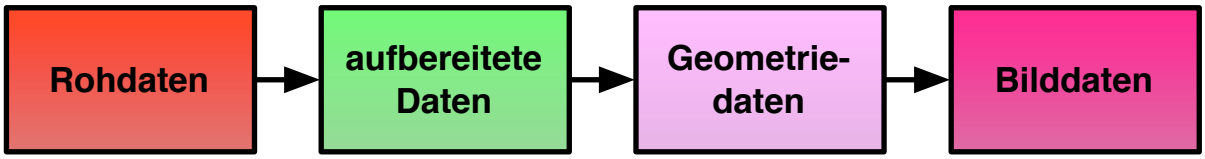


Abbildung 3-2: Datenfluss in der Visualisierungspipeline [17]

3.3.4 Verteilung der Stufen der Visualisierungspipeline

Wie schon bei der *Datenaufbereitung* erwähnt, können die einzelnen Schritte der Visualisierungspipeline auf mehrere Rechner verteilt sein. Beispielsweise lassen sich so *Visualisierungs-Services* als Teil einer SOA einsetzen. Da das hier zu entwickelnden Framework später Teil einer großen Simulation-Workflow-Umgebung wird, werden hier die vier Varianten aus [17] zur Verteilung der drei Stufen gezeigt (siehe Abbildung 3-3). Dabei wird zwischen dem *Autor* (später *JDQVisF*) und dem *Betrachter* (später *JDQVisClient*) als mögliche Rollen unterschieden. Das Framework wird alle Stufen der Verteilung unterstützen.

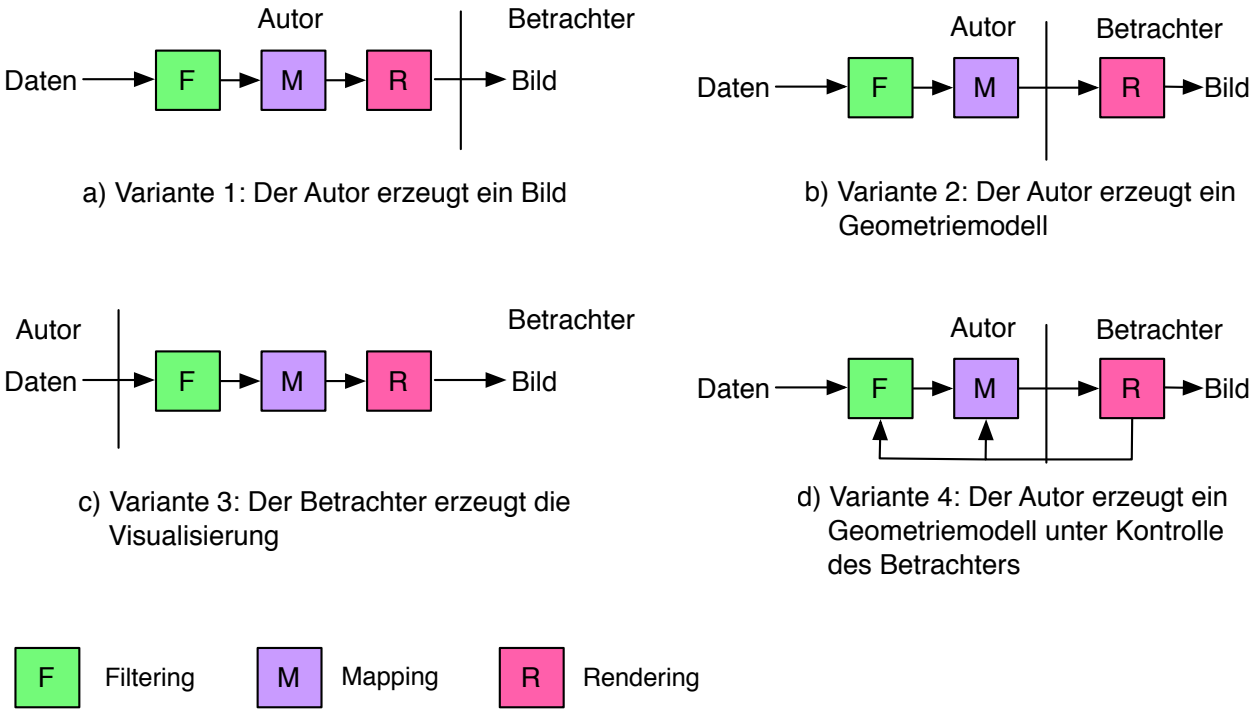


Abbildung 3-3: Varianten zur Verteilung der Schritte der Visualisierungspipeline [17]

Variante 1 – Der Autor erzeugt ein Bild oder eine Bildsequenz (a)

Hier führt der Autor alle Schritte der Visualisierungspipeline durch. Der Betrachter hat keine Möglichkeit in den Visualisierungsprozess einzugreifen. Praktisch bedeutet das, dass der Autor Bilder erzeugt, die er anschließend an den Betrachter versendet. Diese Variante setzt im Allgemeinen eine sehr hohe Bandbreite voraus, wenn ohne Kompressionsverluste und in Echtzeit Datenqualitätswerte visualisiert für den Betrachter zur Verfügung stehen sollen.

Variante 2 – Der Autor erzeugt ein Geometriemodell der Daten (b)

Bei dieser Variante erstellt der Autor ein Geometriemodell der Daten, legt also die Art und Weise fest wie das Bild später beim Betrachter generiert wird. Der Betrachter kann die Bildgenerierung nach seinen speziellen Anforderungen und Vorstellungen steuern. Obwohl diese Variante den Vorteil bringt, dass der Betrachter beliebig im Geometriemodell navigieren kann und somit auch Details analysieren kann, die bei der ersten Variante verborgen geblieben wären, so hat er doch keinen Einfluss auf das Mapping, bei dem wichtige Entscheidungen für die Darstellungsmöglichkeiten getroffen werden.

Variante 3 – Der Betrachter erzeugt die Visualisierung (c)

Hier liefert der Autor lediglich die Rohdaten für die Visualisierung. Alle Schritte der Visualisierungspipeline werden vom Betrachter durchgeführt. Dies ermöglicht einen maximalen Grad der Freiheit und bietet dem Betrachter volle Kontrolle über den Visualisierungsprozess. Der Nachteil liegt aber vor allem in der Menge der Rohdaten. Bei der Berechnung von Datenqualitätswerten, können diese leicht in den Giga-

bytebereich gehen. Diese Datenmenge lässt sich nicht ohne weiteres versenden. Außerdem müsste ein einheitliches Datenformat für den Austausch festgelegt werden.

Variante 4 – Der Autor erzeugt ein Geometriemodell unter Kontrolle des Betrachters (d)
Diese Variante ist ein Kompromiss aus den oben beschriebenen. Hierbei erzeugt der Autor, wie in Variante 2 ein Geometriemodell der Daten, führt also die ersten beiden Schritte der Visualisierungspipeline durch. Er tut dies aber unter Kontrolle des Betrachters, welcher über eine definierte Schnittstelle in den Filter- und Modellierungsprozess eingreifen kann. Der Betrachter seinerseits generiert am Ende ein Bild aus dem Geometriemodell. Somit wird sicher gestellt, dass der Betrachter immer genau *die* Visualisierung bekommt, welche aktuell für seine Problemstellung am geeignetsten ist.

3.4 Anforderungen an eine Visualisierung

Die Visualisierung von Datenwerten ist eine Abbildung ihrer Eigenschaften auf visuelle Attribute und hat als Ergebnis ein Bild oder Bildsequenzen. Diese Abbildung kann eine einfach Form, wie etwa eine Abbildung auf Farbattribute sein, oder aber komplex mit Animationen und einem dreidimensionalen Geometriemodell.
Bevor die allgemeinen Anforderungen an eine Visualisierung genauer untersucht werden können, muss zunächst der Begriff *Visualisierungsqualität* definiert werden. Sie dient als Referenz für die Nützlichkeit und somit für das Erfüllen des *Effektivitätskriteriums* (siehe Kapitel 3.4.2) einer bestimmten Visualisierung.

Definition: *Visualisierungsqualität*
Die Qualität einer Visualisierung definiert sich durch den Grad, in dem die bildliche Darstellung das kommunikative Ziel der Präsentation erreicht. Sie lässt sich als das Verhältnis von der vom Betrachter in einem Zeitraum wahrgenommenen Informationen zu der im gleichen Zeitraum zu vermittelnden Informationen beschreiben. Die Qualität einer Visualisierung ist somit in starkem Maße abhängig von den Charakteristika der zugrunde liegenden Daten und ihrer Eigenschaften, dem Bearbeitungsziel, den Eigenschaften des Darstellungsmediums sowie den Wahrnehmungskapazitäten und den Erfahrungen des Betrachters. [17]

Die Visualisierungsqualität beschreibt demnach das Verständnis des Benutzers, eine reale Situation, z.B. die Datenqualitäts- und Simulationswerte, anhand ihrer Repräsentation durch die visuellen Attribute zu rekonstruieren, verstehen und Rückschlüsse zu ziehen.

Die Qualität der Visualisierung ist abhängig von vielen Einflussfaktoren. Folgende Faktoren spielen bei der Erzeugung einer geeigneten Visualisierung eine besondere Rolle [17]:

- Die Art und Struktur der Daten
- Das Bearbeitungsziel der Visualisierung
- Das Vorwissen des Anwenders / Betrachters
- Die visuelle Fähigkeiten und Vorlieben des Betrachters
- Übliche Metaphern des Anwendungsgebietes / Konventionen
- Die Charakteristika des Darstellungsmediums

In Kapitel 3.6 wird auf die Faktoren genauer eingegangen, die im Zusammenhang mit Datenqualität relevant sind.

Aus der Definition der Visualisierungsqualität lassen sich die Anforderungen *Expressivität*, *Effektivität* und *Angemessenheit* ableiten, die bei der Visualisierung einer Datenmenge eingehalten werden müssen.

3.4.1 Expressivität

Nach [17] ist das wichtigste Kriterium einer guten Visualisierung ihre Expressivität oder Ausdrucksfähigkeit. Sie besagt, dass die zugrunde liegende Datenmenge möglichst unverfälscht wiedergegeben werden muss und nur die tatsächlich enthaltenen Informationen dargestellt werden. Sie ist vor allem von der Struktur und Art der Daten abhängig. Um eine effektive Verarbeitung der Visualisierung zu gewährleisten, ist das Expressivitätskriterium Grundvoraussetzung. Das bedeutet insbesondere, dass die Auswahl der Visualisierungstechnik der ersten Schritt bei der Visualisierung von Daten sein muss [17].

In der Informationsvisualisierung wurde zur Beurteilung der Expressivität der Begriff *Lie Factor* eingeführt:

Lie Factor – *“The representation of numbers, as physically measured on the surface of the graphic itself, should be directly proportional to the quantities represented."* [19]

Er beschreibt demnach die Größe des Effektes in der grafischen Repräsentation zu dem tatsächlichen Effekt innerhalb der Daten:

$$\text{Lie Factor} = \frac{\text{Größe des Effektes in der Grafik}}{\text{Größe des Effektes in den Daten}}$$

Durch die Proportionalität ist ein Wert um 1 ein Maß für eine hohe Expressivität. Ein Wert kleiner 1 bedeutet, dass wichtige Effekte in den Daten nicht oder nur schwach ab-

gebildet wurden und ein Wert größer als 1 lässt auf eine Dramatisierung des Effektes schließen. Abbildung 3-4 zeigt ein bekanntes Beispiel für einen *Lie Factor* größer 1 und dadurch eine Verletzung des Expressivitätskriteriums. Das Bild illustriert die Ölpreisentwicklung zwischen 1973 und 1979 durch immer größer werdende Ölfässer. Dabei wächst die Größe der Ölfässer jedoch viel schneller als die tatsächliche Preissteigerung, was zu einem *Lie Factor* größer als 1 führt.

$$\text{Lie Factor} = \frac{\text{Steigerung der Größe der Ölfässer}}{\text{Tatsächliche Ölpreiserhöhung}} > 1$$

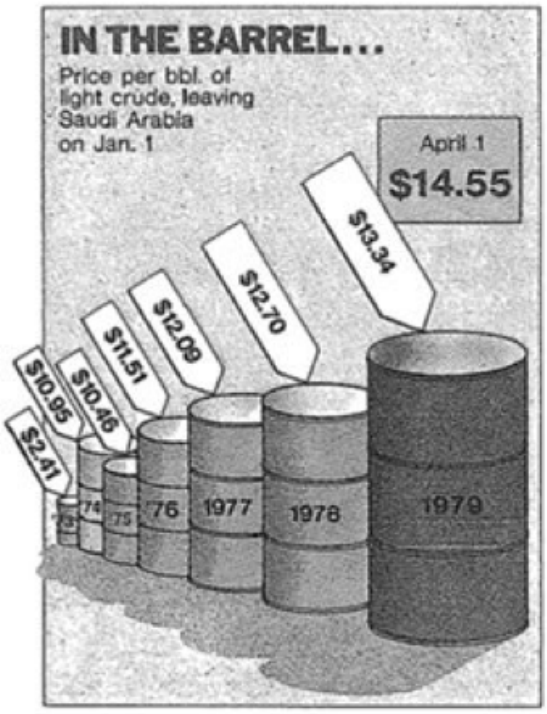


Abbildung 3-4: Beispiel Lie Factor - Die Größe der Ölfässer wächst stärker als die tatsächlichen Ölpreisentwicklung [19]

3.4.2 Effektivität

Da es für ein und dieselbe Datenmenge durchaus mehrere Visualisierungsformen geben kann, die das Expressivitätskriterium erfüllen, wird ein weiteres Auswahlkriterium benötigt. Es muss entschieden werden, welche Darstellungsform die Eigenschaften der Daten optimal wiedergibt. Anders als das Expressivitätskriterium ist die Effektivität einer Visualisierung zusätzlich zu den Daten auch von den jeweiligen Einflussfaktoren abhängig (siehe Kapitel 3.6) [17]. Beispielsweise spielt das Bearbeitungsziel oder die Rechenleistung des Anzeigegerätes eine entscheidende Rolle. Hier muss genau untersucht werden, welche Informationen und Eigenschaften auf welche Art und Weise dargestellt werden müssen, damit sie der Betrachter effektiv erkennen kann.

Das Effektivitätskriterium beschreibt also den Nutzen der gewählten Darstellungsform im aktuellen Kontext. Das heißt, es „..*gibt Aufschluss über die Fähigkeit einer Darstellungsform, die in ihr enthaltenen Informationen zu veranschaulichen und auf intuitive Weise dem Betrachter zu vermitteln*“ [17].
Für eine häufige Form der Verletzung des Effektivitätskriteriums, hat sich in der Informationsvisualisierung der Begriff *Chart Junk* gebildet [19]. Unter diesen Begriff fallen alle Eigenschaften eines Diagrammes, die für den Betrachter für die Problemlösung keine nützlichen Zusatzinformationen bereitstellen und somit für die Informationsübertragung nutzlos sind.

Abbildung 3-5 zeigt ein Beispiel für *Char Junk* anhand eines dreidimensionalen Balkendiagramms. Die eigentlichen Informationen sind dabei die Skalen und die errechneten Werte. Alle anderen grafischen Attribute, wie der 3D-Effekt oder der Schattenwurf, tragen nicht zum Verständnis bei und können den Betrachter unnötig verwirren.

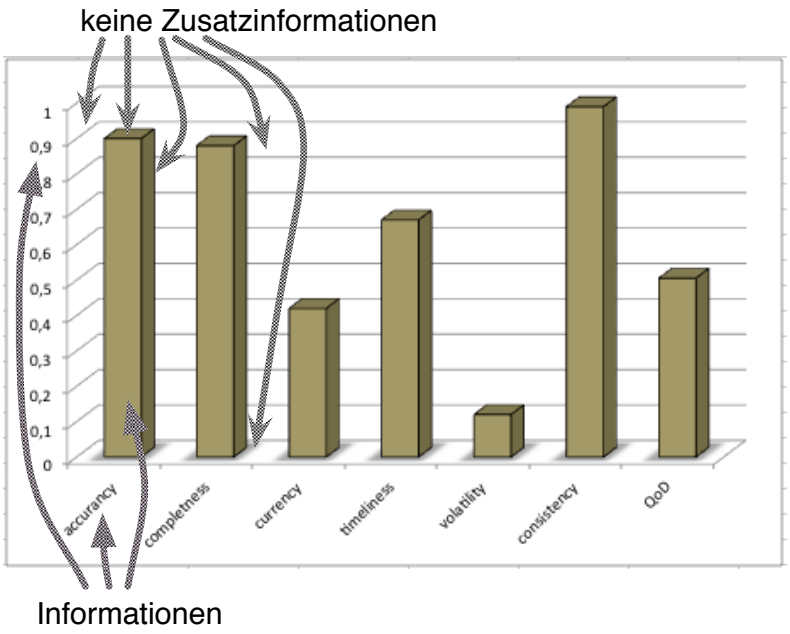


Abbildung 3-5: Beispiel Chart Junk – Kein Informationsgewinn durch das Hinzufügen eines 3D-Effektes, Schatten und horizontalen Linien zum Diagramm [19]

3.4.3 Angemessenheit

Die Angemessenheit beschreibt den, für eine Visualisierung benötigten, Rechenaufwand und ihren physikalischen Ressourcenbedarf. Sie beschreibt also weniger die Qualität der resultierenden Visualisierung aus Sicht des Betrachters, als vielmehr den Aufwand und die Kosten für deren Generierung. [17]
Die Angemessenheit ist eng mit dem Effektivitätskriterium und den Einflussfaktoren verbunden und spielt bei der Auswahl passender Visualisierungen eine wichtige Rolle. Beispielsweise müssen, bei sich schnell ändernden Daten, in kurzen Abständen neue

Visualisierungen generiert und präsentiert werden, die nur die wichtigsten Eigenschaften der Datenmenge abbilden. Im Gegensatz dazu, können bei langzeitbeständigen Daten komplexere und rechenintensivere Visualisierungen berechnet werden.

3.5 Beschreibung der Daten als Ausgangspunkt der Visualisierung

Der Ausgangspunkt jeder Visualisierung ist die ihr zugrunde liegende Datenmenge. Wie in Kapitel 2.1 definiert, können Daten verschiedene Strukturen und Formate besitzen.

Diese Kapitel zeigt *Daten* im Kontext der Visualisierung.

3.5.1 Datenformate

Im Kapitel 2.1.2 und 2.1.3 wurden Daten im Zusammenhang nach ihren logischen Eigenschaften, *Struktur* und *Änderungsrate,* untersucht. Nach [17] darf zusätzlich bei praktischen Visualisierungsanwendungen die Art die Art und Weise, wie die Daten und ihre Eigenschaften physikalisch gespeichert sind, nicht vernachlässigt werden. Da das hier entwickelte Visualisierungsframework in einer heterogenen Simulationsumgebung mit unterschiedlichen Komponenten realisiert werden soll, ist die Unterstützung einheitlicher Datenformate für die Kommunikation unentbehrlich.

Für die Repräsentation der Datenqualitätswerte wird das *Extensible Markup Language* (XML) [34] Format verwenden. Es erlaubt eine plattformunabhängige hierarchische Beschreibung der Daten und eignet sich durch die verbreitete Unterstützung innerhalb der WebServices Technologie, besonders für die Verwendung innerhalb von Simulation-Workflows.
Für den interessierten Leser sei an dieser Stelle auf die Internetseite des W3C [34] verwiesen, da eine detaillierte Beschreibung von XML den Rahmen dieser Arbeit übersteigt.

Listing 1 zeigt beispielhaft die Struktur eines Datenqualitätswert als Eingabedatum für die Visualisierung.

```
<InterpretionCalculationResult>
    <Value> 0.95 </Value>
    <InterpretionId> Accuracy </InterpretionId>
</InterpretionCalculationResult>
```

Listing 1: Beispiel für die Repräsentation eines Datenqualitätswertes in XML

Das Format der generierten Visualisierungen ist maßgeblich von den Eigenschaften der Anzeigegeräte abhängig. So können leistungsstarke Geräte, wie Tablet-Computer, komplexe Visualisierungen verarbeiten, wohingegen leistungsschwache Geräte nur Bilder anzeigen können.
Das *JDQVisF* wird grundsätzlich jedes Datenformat, das auf den jeweiligen Geräten verarbeitet werden kann, unterstützen. Beispielsweise können einfache Bilder durch das freie, erweiterbare und verlustfreie Grafikformat PNG [35] dargestellt werden. Komplexe 3D-Visualisierungen hingegen, können durch die Datenformate VRML oder X3D [36] an die Anzeigegeräte versendet werden.
Für eine genaue Beschreibung der vorgestellten Formate sei an dieser Stelle auf die Quellen [35] und [36] verwiesen, da eine detaillierte Beschreibung den Umfang dieser Arbeit übersteigen würden.

3.5.2 Reduktion einer Datenmenge

Große Datenmengen lassen sich, auf Grund ihres komplexen Informationsgehalts, selten in einem einzigen Bild verständlich wiedergeben. Insbesondere bei Simulationsdaten, deren Anzahl leicht in die Millionen gehen kann, wäre eine Visualisierung aller Werte restlos überladen. Aus diesem Grund, kann es notwendig sein, die ursprüngliche Datenmenge vor der Visualisierung zu reduzieren und somit den Betrachter bei der Auswertung der Daten zu unterstützen.

Im Folgenden werden die nach [17] existierenden Möglichkeiten zur Datenreduktion im Bezug auf Datenqualitätswerte und Simulationsdaten gezeigt:

Entfernung irrelevanter Daten – Je nach Aufgabenstellung und Einsatzgebiet ist es sinnvoll, uninteressante Daten vor der Visualisierung zu entfernen. Uninteressant sind in diesem Zusammenhang alle Daten, die nichts zum Verständnis des Betrachters beitragen oder keinen Einfluss auf das betrachtete Problem haben. Beispielsweise können bei FEM basierten Simulationen alle Metadaten, die nicht direkt in Zusammenhang mir der Datenqualität oder der Simulationsergebnisse stehen, entfernt werden.

Abstraktion der Datenmenge – Bei diesem Ansatz werden nur die wichtigsten Eigenschaften der Daten visualisiert, anstelle der gesamten Datenmenge. Beispielsweise können bei einer Übersichtsanzeige, die einzelnen Dimensionen der Datenqualitätswerte zu einem einzelnen *QoD-Wert* aggregiert werden. Hier könnte zum Beispiel der Maximal/ Minimalwert, der Durchschnitt oder Median der sechs Dimensionen berechnet und visualisiert werden. Wobei, unter dem Gesichtspunkt der Datenqualität, der Minimalwert eine besondere Rolle spielt. Es kann in vielen Fällen ausreichen, wenn eine Dimension, zum Beispiel die Genauigkeit, die an sie gestellten Anforderungen nicht erfüllt und dadurch das Ergebnis der Berechnung unbrauchbar macht.

Angabe eines Bereiches von Interesse – Bei großen Simulationen kann es sinnvoll sein dem jeweiligen Domänenspezialisten nur die Datenqualitätswerte im Detail zu zeigen, die für sein Gebiet von Interesse sind. Das heißt, die Datenwerte können in zwei Klassen eingeteilt werden. Die „*wichtigen Daten*“, die genauer betrachtet werden müssen und in „*übrige Daten*“, bei denen eine aggregierte Darstellung ausreichend ist. Ein weiteres Beispiel im Zusammenhang mit Datenqualität könnte eine Auswahl der Datenqualitätswerte sein, welche an den Rändern der Metrik liegen. Bei einem Schwellenwert von 0,7 wäre es demnach sinnvoll, nur die Werte besonders hervorzuheben, bzw. visuell zu kennzeichnen, die nahe an diesem Wert oder darunter liegen. So können mögliche kritische Komponenten frühzeitig erkannt werden.

Auswahl von Teilmengen – Hier werden aus der Ausgangsdatenmenge Teilmengen erzeugt, welche anschließend einzeln visualisiert werden. Im Zusammenhang mit Datenqualität kann hier der Ansatz des „*Focusing & Linking*“ betrachtet werden. Er beschreibt die Auswahl von Datenwerten die für das aktuelle Problem am wichtigsten sind. So kann bei der Visualisierung von Simulationsdaten oder Datenqualitätswerten der Fokus auf den Bereich der Datenmenge gesetzt werden, an dem das untersuchte Problem vermutet wird.

3.6 Einflussfaktoren auf die Visualisierung

Das Finden einer Visualisierung die alle in Kapitel 3.4 gezeigten Anforderungen erfüllt, ist von vielen Einflussfaktoren abhängig. Hierzu zählen neben den Charakteristiken der zu visualisierenden Datenmenge, vor allem auch die Spezifikation von Bearbeitungszielen, die somit die eigentlichen Ziele einer Visualisierung festlegen [17].
Die Bearbeitungsziele werden im ersten Unterkapitel genauer betrachtet. Anschließend werden die Eigenschaften menschlicher visueller Wahrnehmung gezeigt. Die letzten Abschnitte befassen sich mit den Faktoren *Anwendungsumgebung* und *Ressourcen*.

3.6.1 Bearbeitungsziele

Die Ziele, die mit einer Visualisierung verfolgt werden, haben zusammen mit den Eigenschaften der zugrunde liegenden Datenmenge einen entscheidenden Einfluss auf die Erzeugung expressiver Bilder [17]. Sie legen fest, welche Informationen im Bild repräsentiert werden und somit bei der späteren visuellen Analyse leicht und eindeutig zu erkennen sein sollen.

Nach [17] ist die Beschreibung der Bearbeitungsziele nicht problemlos. Aus Anwendersicht wäre eine detaillierte, problemangepasste Beschreibung ideal. Zum Beispiel, wäre bei der Untersuchung von Datenqualitätswerten innerhalb einer Knochensimulation eine Zielvorgabe, „*Farbiges Erkennen von numerischen Problemzonen“*. Diesen Detailie-

rungsgrad können aber nur hochspezialisierte Visualisierungswerkzeuge verarbeiten. Aus diesem Grund ist die Formulierung allgemeingültiger Bearbeitungsziele, die Fachbereichsübergreifend gelten, wichtig.
[17] formuliert mit *Directed Search*, *Comparision* und *Exploration* drei allgemeine Bearbeitungsziele und leitet daraus verschiedene Problemklassen ab, die als Oberklassen für die allgemeinen Bearbeitungsziele eingesetzt werden können. Im Bezug auf Datenqualitätsvisualisierung spielen die Folgenden eine wichtige Rolle:

Identifikationsproblem – Welchen Wert haben Daten in einem bestimmten Gebiet?

Lokalisierungsproblem – Wo liegen Daten in einem bestimmten Gebiet?

Korrelationsproblem – Gibt es Zusammenhänge zwischen zwei oder mehreren Variablen oder Datenwerten und bestimmten Gebieten des Beobachtungsraumes?

Vergleichsproblem – Wie unterscheiden sich die Datenwerte in einem bestimmten Gebiet oder zu unterschiedlichen Zeitpunkten?

Verteilungsproblem – Wo liegen Extremwerte und Ausreißer?

Um den Anforderungen der einzelnen Fachbereiche (z.B. innerhalb einer Simulation) gerecht zu werden, sind Abbildungen der speziellen Bearbeitungsziele auf die allgemeinen Bearbeitungsziele notwendig (siehe Abbildung 3-6).

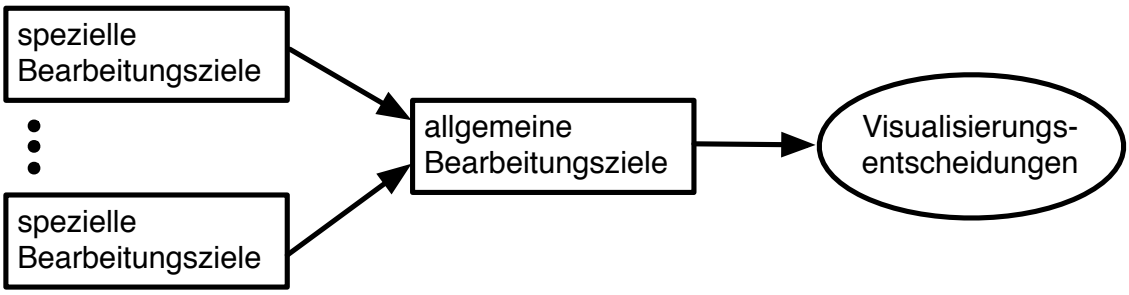


Abbildung 3-6: Abbildung von speziellen Bearbeitungszielen auf allgemeine Bearbeitungszielen zur Beeinflussung von Visualisierungsentscheidungen [17]

Das oben genannte domainspezifische Bearbeitungsziel „*Farbigen Erkennens von numerischen Problemzonen“,* könnte also auf die allgemeinen Ziele „*Identifikation“* und „*Lokalisation“* abgebildet werden.

3.6.2 Menschliche Wahrnehmung – Objekterkennung und Gestaltgesetze

Der Mensch hat spezielle Fähigkeiten, die für effektive Visualisierungen berücksichtigt werden müssen. Dieses Kapitel fasst die physischen und psychologischen Eigenschaf-

ten zusammen und verdeutlicht diese an verschiedene Beispiele. Der genaue Ablauf der Bilderkennung durch das Auge und Gehirn würde den Umfang dieser Arbeit übersteigen und wird deshalb nur sehr vereinfacht aufgezeigt. Der Interessierte Leser sei an dieser Stelle auf [38] verwiesen.

Die im Folgenden gezeigten Gesetze und Prinzipien zur menschlichen visuellen Wahrnehmung sind elementar wichtig und bilden die Basis für das Entwickeln und Finden guter Visualisierungen.

> *"Perception is our window to the world that enables us to experience what is out there in our environment. Thus, perception is the first step in the process that eventually results in all of our cognitions. Paying attention, forming and recalling memories, using language, and reasoning and solving problems all depend right at the beginning on perception. Without perception, these processes would be absent or greatly degraded. Therefore it is accurate to say that perception is the gateway to cognition."* [19]

3.6.2.1 Präattentive Wahrnehmung

Die präattentive Wahrnehmung ist die Fähigkeit des Menschen, innerhalb kürzester Zeit (200 – 250 Millisekunden) die Elemente zu erkennen, die aus einer Menge hervorstechen [19]. Sie unterscheidet dabei zwischen Farbe, Form, Ausrichtung, Größe, Abgeschlossenheit, Gruppierungen, Anzahl oder Luminanz.

Die präattentive Wahrnehmung soll hier an einem Beispiel durch das Erkennen eines roten Punktes in einer Menge von blauen Punkten gezeigt werden (Abbildung 3-7) [19]. Es ist ohne Absuchen des Bildes leicht zu Erkennen, ob dieses einen roten Punkt enthält oder nicht.

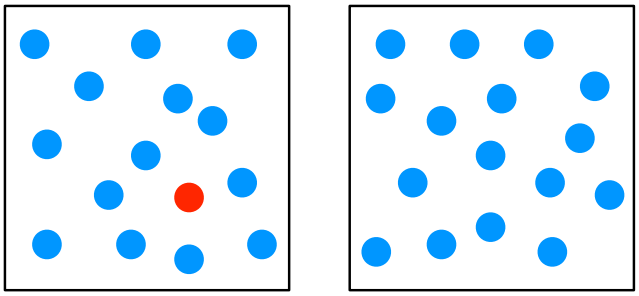


Abbildung 3-7: Beispiel Präattentive Wahrnehmung – Entscheiden ob ein roter Punkt in der Menge enthalten ist [19]

3.6.2.2 Gestaltprinzipien nach Max Wertheimer

Max Wertheimer (1880 – 1943) war Psychologe und Philosoph und gilt als Begründer der Gestaltpsychologie. Er befasste sich unter anderem mit der Wahrnehmung von Objekten.
Das es sich bei der Objektwahrnehmung um einen sehr komplexen Vorgang handelt, bleibt den meisten Menschen verborgen, da ihnen die Erkennung und Wahrnehmung als selbstverständlich und einfach vorkommt. Bekannte Bilder, wie beispielsweise im Straßenverkehr Autos und andere Verkehrsteilnehmer, werden leicht und oftmals unbewusst wahrgenommen und verarbeitet [20].

Wenn man jedoch die Ausgangsbedingungen betrachtet, unter denen ein Objekt als solches erkannt wird, erkennt man die dahinterliegende Komplexität. Ein Körper erzeugt zweidimensionale Abbilder auf unserer Netzhaut. Bei der Objekterkennung müssen diese dann in eine dreidimensionale Abbildung mit Hilfe der Umwelt umwandelt werden [20].
Dabei können leicht Probleme entstehen. Beispielsweise muss, wenn sich Linien schneiden, entschieden werden, ob es sich nach dem Schnitt um das selbe Objekt handelt oder nicht[20].

Ein in der Literatur anerkannter Ansatz zur Erklärung der Objektwahrnehmung liefern die Gestaltgesetze beruhend auf Max Wertheimer [20]:

Das Gesetz der Nähe – besagt, dass gleiche Elemente (Elemente mit gleichem Reiz) mit geringeren Abständen zueinander als zusammengehörig wahrgenommen werden. Wie in Abbildung 3-8 zu sehen, werden die Punkte links in Zeilen und rechts in Spalten geordnet.

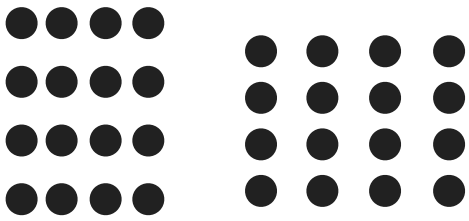


Abbildung 3-8: Gesetz der Nähe – Gruppierung der Punkte in Zeilen und Spalten [20]

Das Gesetz der Ähnlichkeit – besagt, dass sich ähnliche Elemente als zusammengehöriger empfunden werden, als sich unähnlich stehende. Dabei ist es irrelevant ob diese sich in Form, Farbe oder anderem ähnlich sind. Abbildung 3-9 illustriert dies an einem Beispiel.

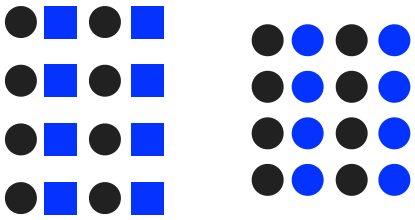


Abbildung 3-9: Gesetz der Ähnlichkeit – Gruppierung der Elemente nach Farbe und Form [20]

Das Gesetz der guten Gestalt – besagt, dass sich gestalthafte Wahrnehmungseinheiten so ausbilden, dass sie im Ergebnis eine möglichste einfache und einprägsame Gestalt, wie Vierecke, Kreuze, usw., darstellen. Abbildung 3-10 zeigt dieses Verhalten an einem Beispiel. In Abbildung 3-10 b entscheidet das Gehirn, dass sich dieses Bild aus zwei Quadraten zusammenstellt, wobei in Abbildung 3-10 c verschiedenen Vielecke wahrgenommen werden.

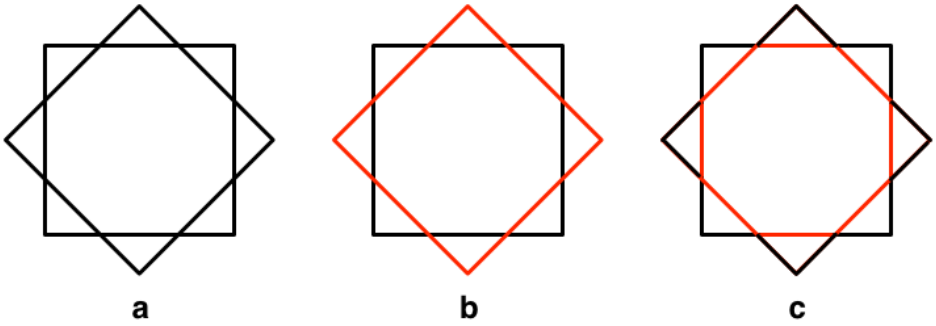


Abbildung 3-10: Gesetz der guten Gestalt – Durch Farbkodierung entstehen verschiedene Vielecke [20]

Das Gesetz der guten Fortsetzung – besagt, dass im Fall zweier sich treffender Linien a und b davon ausgegangen wird, dass sich diese im Punkt x schneiden und nicht, wie in Abbildung 3-11 rechts, ein V-förmiges Gebilde repräsentieren.

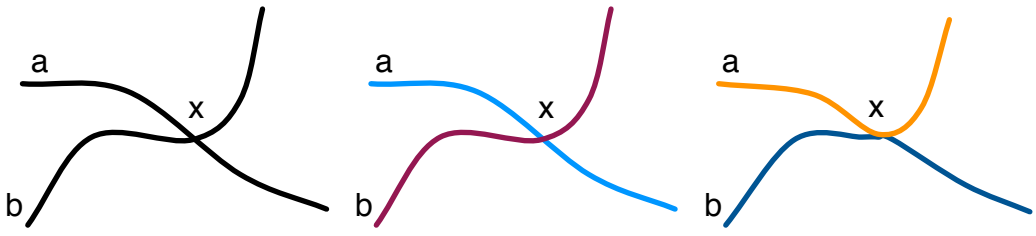


Abbildung 3-11: Gesetz der guten Fortsetzung – Das Gehirn setzt die Linien so fort, dass sie sich schneiden [20]

Das Gesetz der Geschlossenheit – verweist auf die Tatsache, dass in geometrischen Gebilden, diejenigen Strukturen als Figur wahrgenommen werden, die eher geschlossen (Abbildung 3-12 links und Mitte) wirken als offene (Abbildung 3-13 rechts).

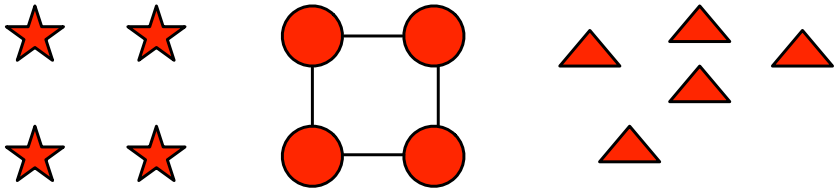


Abbildung 3-12: Gesetz der Geschlossenheit – Enge und regelmäßig Strukturen werden eher als Figur wahrgenommen als unregelmäßige [20]

Das Gesetz des gemeinsamen Schicksaals – besagt, dass sich Elemente einer Reizvorlage, die eine Veränderung oder Bewegung, z.B. durch Drehung oder Verschiebung in die gleiche Richtung, als Einheit wahrgenommen werden (Abbildung 3-13).

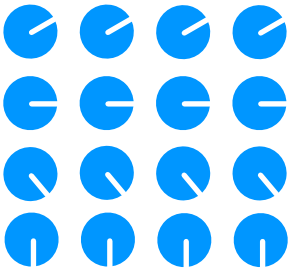


Abbildung 3-13: Gesetz des gemeinsamen Schicksaals – Gleichzeitig bewegende Objekte gehören zusammen [20]

3.6.2.3 Figur-Grund-Unterscheidung

Neben den oben aufgeführten Gestaltgesetzen existiert noch eine weitere, für die Objekterkennung wichtige Eigenschaft. Die *Figur-Grund*-Unterscheidung (oder *Vordergrund-Hintergrund*-Unterscheidung) beschreibt die grafische und räumliche Organisation von Einheiten. Dies geschieht in der Regel unter Bildung einer Markoeinheit [20]. Beispielsweise gruppieren sich viele *Menüpunkte* auf dem Bildschirm zu einem *Menü*.

Im Folgenden werden diese Prinzipien genauer betrachtet [20]:

- Die kleinere Einheit wird eher als Figur vor einem größeren Hintergrund wahrgenommen als umgekehrt. Die Figur liegt dabei phänomenal vor dem Hintergrund.

- Die dunklere Einheit wird eher als Figur auf einem helleren Hintergrund wahrgenommen als eine hellere vor dunklem Grund.

Abbildung 3-14 verdeutlicht die beiden Prinzipien.

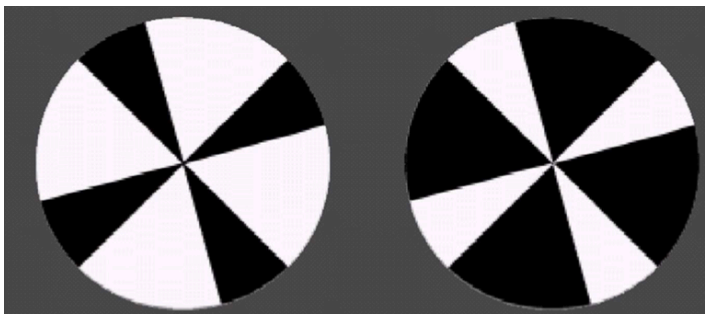


Abbildung 3-14: Vordergrund - Hintergrund: Objekt ist klein, mit Silhouette und im Vordergrund [14]. Links ist das schwarze Objekt im Vordergrund, rechts das weiße.

- Eine räumliche zentrale Einheit wird eher wahrgenommen als eine periphere, wie beispielsweise ein Anmelde- oder Hinweisfenster auf dem Hauptbildschirm.

- Eine Einheit mit einer vertikalen oder horizontalen Hauptachse wird eher als Figur wahrgenommen. Dabei ist die Wirkung der vertikalen größer als die einer horizontalen.

- Eine symmetrische Einheit wird eher als Figur wahrgenommen als eine asymmetrische. Die Symmetrie um die senkrechte Mittelachse hat dabei die stärkste Wirkung.

Abbildung 3-15: Symmetrische Einheiten werden als Figur wahrgenommen – Links die mit symmetrischen Rand. In der Mitte die schwarzen und rechts die weißen Einheiten. [14]

- Eine Einheit mit konvexen Rändern wird eher als Figur wahrgenommen als eine mit konkaven (nach innen gewölbten) Rändern [20].

In Kombination mit den oben gezeigten Gestaltprinzipien lassen sich noch folgende Aussagen zur Objekterkennung machen die für eine gute Visualisierung zusätzlich berücksichtigt werden müssen [14]:

Einfache Fortsetzung – beschreibt das Vervollständigen von grafischen Attributen zu bekannten Formen. In Abbildung 3-16 werden die gestrichelten Linien zu zwei ineinander verschachtelten Objekten fortgesetzt.

Abbildung 3-16: Einfache Fortsetzung – Das Gehirn formt aus den gestrichelten Linien ein Rechteck und ein Sechseck [14]

Dünn ist im Hintergrund – besagt, dass dünnere Formen eher in den Hintergrund gesetzt werden als dicke. Abbildung 3-17 illustriert diesen Zusammenhang.

Abbildung 3-17: Dünn ist im Hintergrund, dick im Vordergrund [14]

3.6.3 Anwendungsumgebung

Nach [17] sind neben den im vorangegangen Kapitel aufgezeigten wahrnehmungspsychologischen Aspekten der Visualisierung von Daten auch der Anwendungskontext, die anwendungsspezifischen Eigenschaften sowie spezielle Anwenderpräferenzen zu beachten.
Dabei umfasst der Anwendungskontext existierende Konventionen im Anwendungsgebiet und die Anwenderpräferenzen.
Das Anwendungsgebiet beschreibt alle schon vorhanden Konventionen zur Darstellung bestimmter Informationen in dem jeweiligen Fachbereich. Darunter fallen auch allgemeingeltende Regeln und Metaphern des adressierten Kulturkreises. Ein wichtiger und in der Literatur gut untersuchter Aspekt, ist dabei die Farbe. Während in unserem Kulturraum die Farbe Weiß für Freude und Reinheit und Schwarz für Trauer in Verbindung

gebracht wird, sind in Indien diese Farbdeutungen genau umgekehrt. Auch in verschiedenen Fachbereichen kann die Bedeutung von Farbe stark variieren. So steht die Farbe Rot in der Wissenschaft als Gefahrensignal, die Farben Blau als neutrales und Grün als positives Signal. In der Medizin steht rot jedoch für Leben und ist daher eher positiv, grün und blau dagegen werden mit Infizierungen und dem Tod in Verbindung gebracht und daher als negativ empfunden.
Die Anwenderpräferenzen beschreiben dagegen die Präferenzen und speziellen Anforderungen des einzelnen Anwenders im speziellen Anwendungsfall. Hierunter fallen auch physische Einschränkungen wie Farbenblindheit und andere Sehschwächen. [17]

Aus diesen Beispielen lässt sich ableiten, dass die Anwendungsumgebung beim Finden guter Visualisierungen genau untersucht werden muss. Andernfalls kann der Anwendungskontext einen zu großen limitierender Faktor darstellen, der die Expressivität und Effektivität von Visualisierungsverfahren wesentlich beeinflussen kann. [17]

3.6.4 Ressourcen

Ein weiterer wichtiger Einflussfaktor bei der Erzeugung von Visualisierungen sind die zur Verfügung stehenden Ressourcen. Dabei fallen unter diesen Begriff alle eingesetzten Hardware-, Software- und Peripheriekomponenten die zur Visualisierung verwendet werden.

Nach [17] haben vor allem die Eigenschaften *Farbvisualisierung* und die *Texturdarstellung* des Anzeigegerätes, großen Einfluss auf die Effektivität. Die wichtigsten, im Zusammenhang mit Datenqualitätsvisualisierung, werden nachfolgend genauer betrachtet.

Ortsauflösung – Die *Ortsauflösung* ist einer der wichtigsten Aspekte, der bei der Visualisierung berücksichtigt werden muss. Es wird im allgemeinen durch die Parameter *absolute Größe des Darstellungsbereiches*, die *Rastergröße des Bildspeichers* und *die Rastergröße des Darstellungs- und Ausgabemediums* beschrieben.

Farbwiedergabe – Die Farbwiedergabe spielt bei der Visualisierung von Datenqualitätswerten eine besondere Rolle. Mit ihrer Hilfe kann die präattentive Wahrnehmung besonders gut ausgenutzt werden um schnell unzureichende Qualitätswerte aus einer großen Datenmenge zu erkennen. Damit werden an das Ausgabegerät Mindestanforderungen an die Farbwiedergabe gestellt, die vor der Entwicklung von Visualisierungen untersucht werden müssen.

Rechenleistung – Die unterschiedlichen Rechenleistungen der Anzeigegeräte müssen bei der Generierung der Visualisierungen berücksichtigt werden. Je nach Einsatzgebiet kann das ein oder andere Gerät zur Darstellung unbrauchbar sein. Soll beispielsweise eine 3D-Annimation eines Knochens angezeigt werden, können Anzeigegräte ohne die Möglichkeit des 3D-Renderings nicht zur Darstellung verwendet werden.

3.7 Grundlegende Techniken

Das in Kapitel 3.2.2 vorgestellte *Mapping* ist das Kernstück der Visualisierungspipeline. Bei dieser Stufe werden die Eigenschaften der zu Grunde liegenden Datenmenge auf visuelle Attribute, auch *visuelle Variablen* genannt, abgebildet.

Diese Kapitel beschreibt die grundlegenden Techniken, die später für das Visualisieren von Datenqualitätswerten eingesetzt werden.

3.7.1 Visuelle Variablen

In der Literatur haben sich die, von [21] eingeführten, visuellen Variablen durchgesetzt und werden in diesem Abschnitt genauer betrachtet.

Definition: *Visuelle Variable*

Die Mittel der graphischen Darstellung zur Transkription von Ähnlichkeits-, Ordnungs- und Proportionalitätsbeziehungen sind die acht Variationen, die das Auge in Bezug auf „Flecken“ wahrnehmen kann [21].

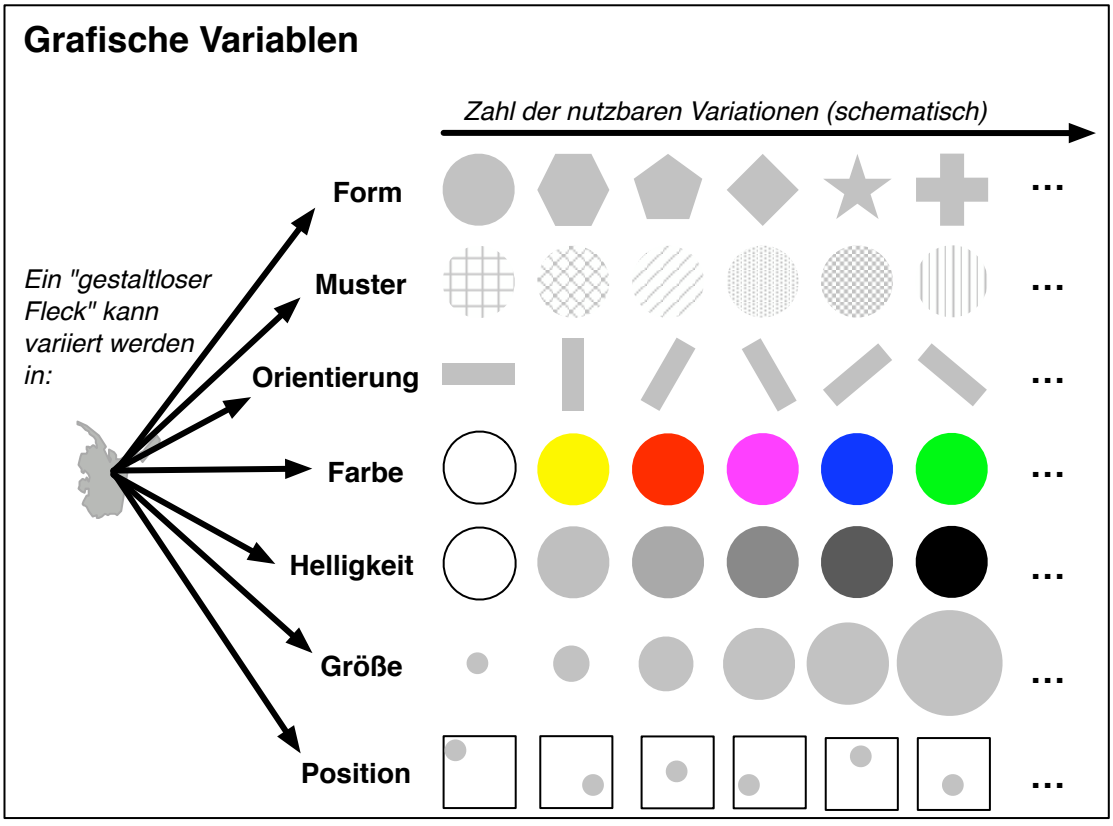


Abbildung 3-18: Grafische Variablen nach [22]

Nach [17] hat jede dieser Variablen seine eigene spezifische Wirkung, die [21] in drei Formen unterscheidet:

Selektive Wirkung – Diese Variablen können unterschiedlichen Datenwerte in Gruppen aufteilen und unterscheiden. [21] nennt diese explizit *trennende Variable* und zählt vor allem *Muster*, *Farbe*, *Orientierung* und *Form* dazu. Diese Variablen eignen sich nach [17] besonders gut dazu Daten auf einer nominalen Skala darzustellen.

Ordinale Wirkung – Datenwerte, die durch solche Variablen kodiert sind, werden vom Betrachter spontan in eine Ordnung gebracht. Es lassen sich also besonders Datenwerte mit einer ordinalen Ordnung gut visualisieren.

Proportionale Wirkung – Zusätzlich zu der Ordnung der Daten, lassen sich Datenwerte durch eine Kodierung mit solchen Variablen mit der zu Grunde liegenden Messgrößen in Beziehungen bringen.

Seit [21] im Jahr 1982 die vorgestellten Variablen zum ersten mal nannte und einführte, finden immer wieder Diskussionen über Erweiterungen statt. Alle neuen Ideen aufzuführen würde aber den Umfang dieser Arbeit übersteigen. Ein spannender Ansatz, der später auch zur Visualisierung der Datenqualitätsdimension *Konsistenz* weiter verfolgt wird, stellt die visuelle Variable *Regularität* dar [23]. Sie beschreibt den visuellen Zusammenhang durch die Anordnung mehrerer Elemente innerhalb eines Bildausschnitts (siehe Abbildung 3-19).

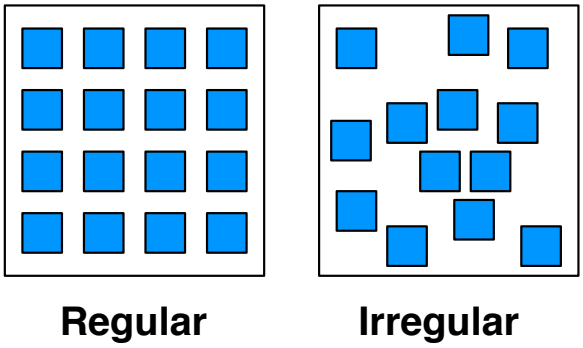


Abbildung 3-19: Regularität [23]

3.7.2 Visuelle Abbildungen

Die Abbildung der Datenwerte auf die grafischen Variablen ist die wichtigste Aufgabe beim *Mapping.* Hier entscheidet sich, wie gut die spätere Visualisierung ist und ob sie dem Betrachter bei der Lösung seines Problems helfen kann. Diagramme stellen dabei die allgemeinste Form der grafischen Darstellung von Daten dar. Von denen in [17] vorgestellten Abbildungen, werden in diesem Abschnitt die, für die Datenqualitätsvisualisierung wichtigen, gezeigt und auf ihre speziellen Eigenschaften eingegangen.

3.7.2.1 Abbildung auf Position, Größe und Orientierung

Die Lokalisation von Objekten im Sehfeld, sowie die Fähigkeiten leicht Positions- und Größenvergleiche anstellen zu können, sind grundsätzliche Abläufe der menschlichen visuellen Wahrnehmung [17]. Diese Prozesse können, dank der hohen örtlichen Auflösung im menschlichen Auge, besonders genau durchgeführt werden. Eine Vielzahl der heute eingesetzten Diagramme nutzen diese Fähigkeiten aus.
Die wichtigsten sind:

Punktdiagramme – Sie nutzen die Eigenschaft der menschlichen Wahrnehmung für die Visualisierung aus, sehr gut relative Positionen einschätzen und vergleichen zu können.

Linien- und Kurvendiagramme – Auch diese Diagrammarten nutzen die gute Wahrnehmbarkeit von Positions- und Längendifferenzen, um Informationen effektiv wiederzugeben. Dabei werden die Werte auf einer gemeinsamen Skala quantitativ abgebildet. Um Trends und lokale Strukturen besser erkennen zu können, werden benachbarte Punkte miteinander verbunden. Diese Diagrammart ist besonders dafür geeignet, zeitabhängige Strukturänderungen anzuzeigen, wie beispielsweise Datenqualitätswerte einer Komponente im Laufe der gesamten Simulation.

Säulen- und Balkendiagramme – Liegen die Datenwerten auf einer ordinalen oder diskreten Skala, können auf der waagerechten Achse des Koordinatensystems die Skala, zum Beispiel die Datenqualitätsdimensionen, und auf der senkrechten Achse die dazugehörigen abhängigen Größen eingezeichnet werden.

Histogramme – Sie stellen eine spezielle Form von Säulendiagrammen dar, mit dem Unterschied, dass nicht die einzelnen Datenwerte abgetragen werden, sondern deren Häufigkeit. Eine Anwendung im Zusammenhang mit Datenqualität in Simulation-Workflows, könnte die Auswahl eines Services sein, dessen Ergebnisse mit einer hohen Häufigkeit eine gute Qualität besitzen.

Kreisdiagramme – Es wird wie Säulendiagramme zur Darstellung von Dateneigenschaften verwendet, die auf einer quantitativen Skala liegen. Das Kreisdiagramm besitzt jedoch, statt eines rechtwinkligen Koordinatensystems, einen Kreis als Bezugssystem. Auf diesem werden die Werte der verschiedenen Eigenschaften, durch unterschiedlich gefärbte Kreisteilen dargestellt werden.

3.7.2.2 Abbildung auf Farbe

Wie im Kapitel über menschliche Wahrnehmung aufgezeigt wurde, stellt die Abbildung auf Farbe ein wichtiges Mittel zur Visualisierung dar. Sie wird vom Betrachter, sofern er keine Einschränkungen wie Farbenblindheit hat, präattentiv wahrgenommen. Zusätzlich lässt sich Farbe mit allen zuvor genannten Visualisierungstechniken kombinieren und

stellt somit in vielen Fällen einen zusätzlichen Freiheitsgrad für die Visualisierung von Daten zur Verfügung [17].

Das Gebiet der Farbenlehre ist ein sehr großes Feld und wird daher in dieser Arbeit nur im Kontext *Datenqualitätsvisualisierung* in Kapitel 3.10 eingeschränkt behandelt. Dort werden die Farben Grün und Rot zur Unterscheidung von guter und schlechter Datenqualität verwendet.
Für Menschen mit einem eingeschränkten Sehvermögen, etwa einer rot-grün-Blindheit, können andere Unterscheidungsmerkmale wie Texturen oder Muster verwendet werden. In dieser Arbeit wird jedoch davon ausgegangen, dass der Leser keine beeinträchtigende Sehschwäche besitzt und die vorgestellten Visualisierungen vollständig erkennen kann.

3.8 Finden von Visualisierungen

Dieses Kapitel beschreibt die Problematik des Findens guter Visualisierungen, die für einen konkreten Anwendungsfall eingesetzt werden können und alle an sie gestellten Anforderungen erfüllen.

3.8.1 Herausforderungen beim Entwerfen passender Symbole

Die Schwierigkeit bei der Entwicklung von guten Visualisierungen entsteht vor allem durch die in Kapitel 3.6 genannten Einflussfaktoren. Damit aussagekräftige Visualisierungen entwickelt werden können, müssen diese genau untersucht und berücksichtigt werden. Dazu ist es in erster Linie notwendig, das Anwendungsgebiet sowie die Präferenzen und Gewohnheiten der späteren Anwender zu verstehen und sie in den Visualisierungsprozess mit einzubeziehen.

In der ersten Phase stellen sich nach [25] vor allem folgende drei Fragen die mit Hilfe der späteren Benutzer und Experten des Anwendungsgebietes zu beantworten sind:

Was soll dargestellt werden? – Diese Frage zielt vor allem auf den informativen Inhalt und damit auf die expressiven Eigenschaften der Visualisierung ab. In der Antwort sind alle Eigenschaften der Datenmenge enthalten, welche auf visuelle Attribute abgebildet werden sollen.

Wozu soll die Darstellung dienen? – Diese Frage dient zur Formulierung des Visualisierungsziels und ist somit Teil des Effektivkriteriums. Im Ergebnis befinden sich vor allem die Wünsche und Erwartungen der speziellen Benutzer.

Wer soll informiert oder überzeugt werden? – In der letzten Fragen werden die Benutzergruppen untersucht, welche mit der Visualisierung arbeiten werden. Hier

müssen alle Standardsymbole, Farbsemantiken und alle weiteren Randbedingungen sehr genau abgesteckt werden, um möglichst eindeutige und unmissverständlichen Visualisierungen zu erreichen.

3.8.2 Entwerfen von Symbolen

Der zweite Schritt bei der Entwicklung guter Visualisierungen, ist das Entwerfen von passenden Symbolen und Metaphern. Diese werden nach [24] typischerweise durch ausgebildete Designer gefertigt und für viel Geld eingekauft.
Anschließend werden diese dann durch Versuche mit Probanden in mehreren Stufen evaluiert und verfeinert. Ein Teil bei diesen Visualisierungsstudien ist zum Beispiel der schrittweise Aufbau des Symbols, durch den Einsatz verschiedener Filter. Das jeweils entstandene Bild wird dabei durch die Probanden *frei* interpretiert. Das heißt, sie bekommen keinerlei Hintergrundinformationen und müssen beschreiben, in welchem Zusammenhang sie diese Symbole setzen würden. Je nach eingesetztem Filter können so unterschiedliche Informationen der Symbole herausgearbeitet und verbessert werden. Beispielsweise könnte in einem ersten Schritt ein *Farbfilter* eingesetzt werden, welcher das Symbol nur in Schwarz-Weiß darstellt und so den Fokus des Benutzers auf dessen Struktur und den Aufbau lenkt ohne durch grelle Farben abzulenken.
Durch diesen schrittweisen Prozess und durch das entsprechende Auswählen der Versuchsparameter und Filter, lassen sich so, für einen speziellen Anwendungsfall, Visualisierungen entwickeln, die allen in Kapitel 3.4 genannten Ansprüchen gerecht werden.

3.8.3 Beispiel für die Auswahl einer Visualisierungsform

In diesem Abschnitt soll anhand eines Beispiels gezeigt werden, wie das Lösen eines einfachen Problems maßgeblich von der gewählte Visualisierung abhängig ist.

Problem: Ordnen Sie eine Menge von Werten in absteigender Reihenfolge.

Variante 1 – Kreisdiagramm

Abbildung 3-20: Beispiel für das Finden von Visualisierungen (Teil 1)

Wie in der Darstellung zu sehen, ist die Wahl eines Kreisdiagramms für diese Art von Problemstellungen ungeeignet. Das menschliche Auge kann nur schwer Größenverhältnisse von angrenzenden, runden Objekten auflösen.

Variante 2 – Säulendiagramm

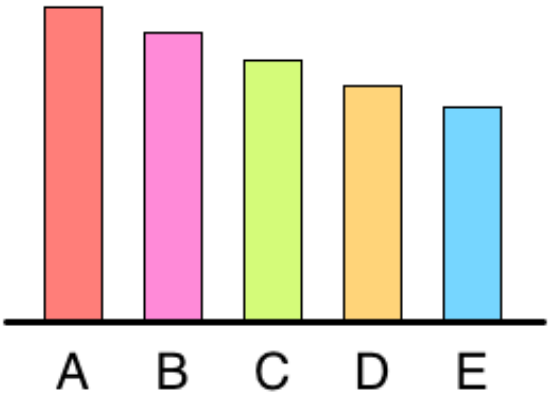


Abbildung 3-21: Beispiel des Findens von Visualisierungen (Teil 2)

Ein Säulendiagramm hingegen bietet dem Auge genügend Freiraum, um die einzelnen Balken zu vergleichen und zu bewerten. Aus diesem Grund ist es für diese Art von Problemstellung besonders gut geeignet.

3.9 Visualisierung von Datenqualität

Nachdem nun alle wichtigen Grundlagen für das Generieren expressiver und effektiver Visualisierungen gezeigt wurden, werden in diesem Kapitel spezielle Visualisierungen für Datenqualitätswerte erarbeitet. Dazu werden die einzelnen Schritte der Visualisierungspipeline entsprechend angepasst und modelliert. Anschließend werden Visualisierungen für die sechs vorgestellten Datenqualitätsdimensionen aufgezeigt.

3.9.1 Datenaufbereitung (Filtering)

Die Datenaufbereitung, oder *Filtering*, ist der erste Schritt in der Visualisierungspipeline. Hier werden die Datenqualitätswerte und Simulationsdaten aufbereitet, um in den späteren Schritten effizient verarbeitet werden zu können. Da alleine die Simulationsdaten in den Gigabytebereich gehen können, stellt dieser Schritt eine besondere Herausforderung dar. Er muss für jede Problemstellung genau untersucht werden, damit die spätere Visualisierung das *Expressivitätskriterium* erfüllt und keine wichtigen Informationen verloren gehen.
Zusätzlich muss hier nach dem Format der Eingabedaten unterschieden werden. Liegen beispielsweise die Daten als einzelne Datenqualitätswerte innerhalb eines einzel-

nen XML-Dokument vor, können diese ohne weitere Filterung an das *Mapping* übergeben werden. Liegen die Datenqualitätswerte jedoch in Kombination mit den ursprünglichen Simulationsdaten vor, beispielweise ein Geometriemodell des menschlichen Oberarmknochen, kann es auf Grund der reinen Datenmenge sinnvoll sein, nur eine bestimmte Auswahl der Simulationsdaten zu visualisieren. Dadurch kann die Rechenzeit kurz gehalten werden und Gesamtperfomance wird nicht negativ beeinträchtigt. Beim Filtering ist sowohl eine maschinelle, als auch die menschliche Datenauswahl möglich.

3.9.2 Visualisierung

Das *Mapping* ist der wichtigste und komplizierteste Schritt in der Visualisierungspipeline. Hier entscheidet sich, welche Informationen mit welchen visuellen Variablen dargestellt werden.

Dieser Abschnitt zeigt mögliche Visualisierungen von Datenqualitätswerten, aufgeteilt nach der jeweiligen Dimension.

Datenqualitätswerte sind abstrakte Daten, die auf unterschiedlichen Skalen abgebildet werden können. Um eine schnelle Auswertung der erzeugten Visualisierungen zu ermöglichen, muss das eingesetzte Mapping eine Abbildung auf visuelle Variablen realisieren, die dem späteren Betrachter geläufig sind. Nach [17] stellen Metaphern eine gute Möglichkeit dar, um diese Ansprüche zu realisieren. Dabei muss die spätere Anwendungsumgebung genau untersucht werden, um Missverständnisse und mögliche Fehlinterpretationen zu vermeiden.

Wie im Kapitel über das Finden von guter Visualisierungen gezeigt wurde, stellt dieser Schritt eine große Herausforderung dar. Die im Folgenden präsentierten Visualisierungen werden später im Kontext eines wissenschaftlichen Knochensimulations-Workflow eingesetzt und stellen somit Spezialfälle und Interpretationen der sechs Datenqualitätsdimensionen durch den Autor und seinen Betreuer dar. Das grundsätzliche Vorgehen kann jedoch als Muster für das Finden weiterer Visualisierungen in anderen Einsatzgebieten dienen.

3.9.2.1 Genauigkeit (Accuracy)

Für die Visualisierung von *Genauigkeit* bieten sich verschiedene nützliche Assoziationen und Metaphern an. In dieser Arbeit wird das Symbol der *Zielscheibe* gewählt, da diese je nach eingesetzter Datenqualitätsskala leicht verändert werden kann. Das bedeutet, je nach Skala und Ansprüchen des Benutzers an die Genauigkeit der visuellen Repräsentation, lassen sich die Ringe der Zielscheibe variieren. Zudem bietet sie eine einfache Möglichkeit einen gegebenen Schwellenwert einzuzeichnen. Er ermöglicht dem Betrachter, dank der Fähigkeit des menschlichen Auges relative Positionierungen gut einschätzen zu können, leicht einen Soll-Ist-Vergleich durchzuführen ohne die ge-

nauen Zahlenwerte vergleichen zu müssen. Zusätzlich werden die Farben Rot für *niedrige Genauigkeit* und Grün für *hohe Genauigkeit* als Verstärkung des visuellen Eindrucks eingesetzt. Abbildung 3-22 und Abbildung 3-23 zeigen Beispielvisualisierungen der Dimension *Genauigkeit.* Jeweils mit einem Schwellenwert (blauer Kreis). Die einzelnen Datenwerte werden als schwarzer Kreis dargestellt, wobei wie im Sport, die äußere Kante den erreichten Wert anzeigt. Der genaue Wert kann zur leichteren und exakten Analyse zusätzlich unter dem jeweiligen Bild angezeigt werden.

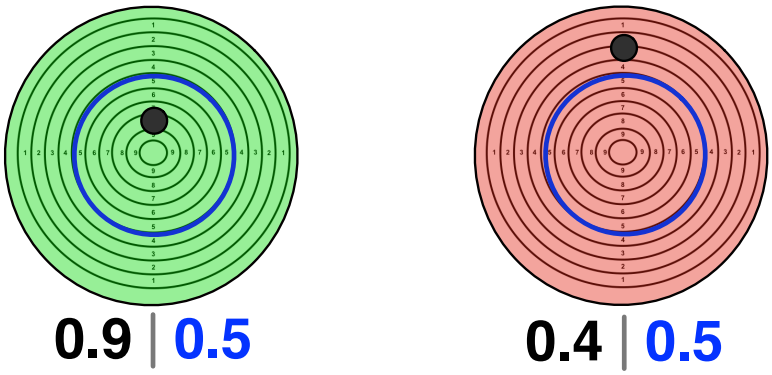


Abbildung 3-22: Genauigkeit als Zielscheibe auf der Skala von 0 bis 1 und einem Schwellenwert von 0,5

Ist die eingesetzte Skala, bzw. der exakte Wert nicht von Bedeutung, dann kann die Zielscheibe leicht modifiziert und wie in Abbildung 3-23 zu sehen eingesetzt werden. Das Bild ähnelt jetzt eher einem Fadenkreuz, das anzeigt, ob sich das Ziel, der gewünschte Datenqualitätswert, im Sucher befindet und somit erreicht wird oder nicht.

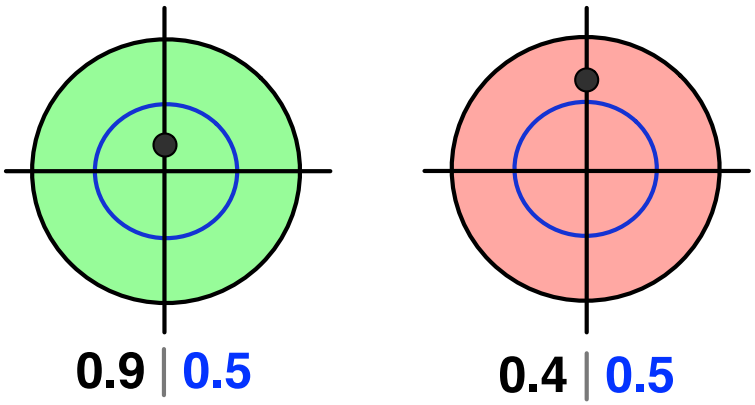


Abbildung 3-23: Genauigkeit als Fadenkreuz ohne Visualisierung der Skala und mit Schwellenwert von 0,5

3.9.2.2 Rechtzeitigkeit (Timeliness)

Auch für eine grafische Repräsentation der Datenqualitätsdimension *Rechtzeitigkeit* gibt es eine Reihe von passenden Metaphern und Symbole, die eingesetzt werden können.

Wie schon bei der Dimension *Genauigkeit* wird hier ein Symbol entwickelt, das vom Benutzer sofort erkannt und leicht verstanden wird und außerdem skalenunabhängig und mit unterschiedlichsten Anforderungen eingesetzt werden kann. Da die Uhr schon immer ein Symbol für die Zeit war, wird sie auch hier eingesetzt. Sie kombiniert alle geforderten Ansprüche an eine gute Visualisierung.
Die erste Variante einer Uhr, die wohl jedem im Zusammenhang mit Rechtzeitigkeit in den Sinn kommt, ist der *Wecker*. Er steht wie kein anderes Symbol für das *nicht verpassen eines Ereignisses* und bietet durch ein großes Ziffernblatt genug Platz um zusätzliche Informationen darzustellen.

Abbildung 3-24 zeigt eine mögliche Variante. Die Skala von 0 bis 1 wird dabei auf das Ziffernblatt abgetragen. Der erreichte Datenqualitätswert wird mit einem schwarzen Pfeil angezeigt, der auf die entsprechende Ziffer zeigt. So lässt sich auch leicht der Schwellenwert durch einen blauen Strich zeigen, der ebenfalls auf die entsprechende Ziffer zeigt und so einen leichten Soll-Ist-Vergleich ermöglicht. Zusätzlich zeigen die Farben Grün und Rot eine *gute Rechtzeitigkeit* und eine *schlechte Rechtzeitigkeit* an. Auch der Animationseffekt im rechten Bild verdeutlicht das *zu späte Eintreffen der Datenwerte* symbolisch.

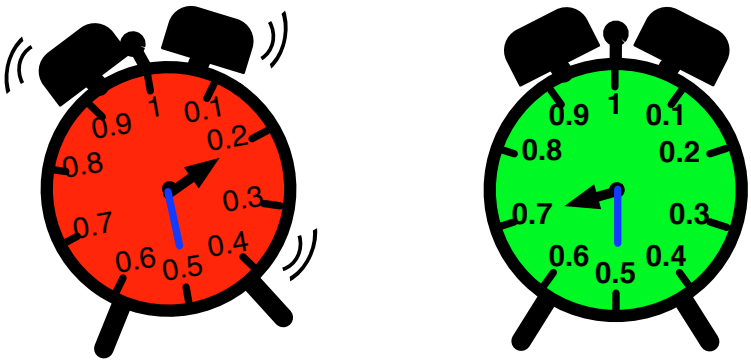


Abbildung 3-24: Rechtzeitigkeit als Wecker auf der Skala von 0 bis 1 und Schwellenwert von 0,5

Ist die genaue Skala für den Betrachter uninteressant, lässt sich der Wecker leicht, wie in Abbildung 3-25 zu sehen modifizieren. Hier wird nur der Schwellenwert an die symbolische *12* gesetzt und der Datenqualitätswert entweder davor, bei *guter Rechtzeitigkeit*, oder dahinter, bei *schlechter Rechtzeitigkeit*, gezeichnet. Der Abstand des schwarzen Pfeils zu *12 Uhr* und damit zum Schwellenwert spielt dabei keine Rolle.

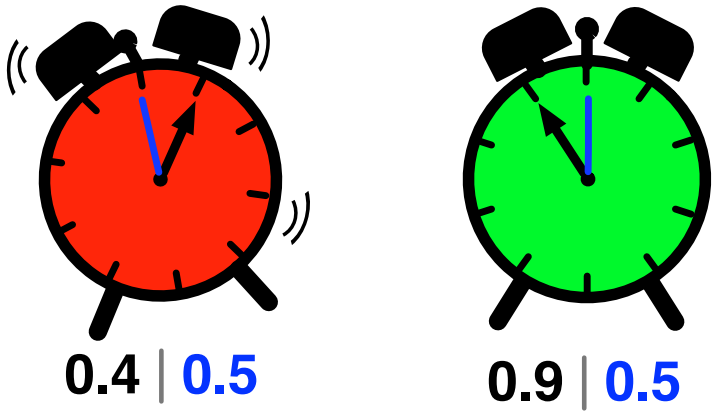


Abbildung 3-25 Rechtzeitigkeit als Wecker ohne Visualisierung der Skala und Schwellenwert von 0,5

Eine weitere Metapher, welche abgeleitet von einer Uhr im Zusammenhang mit *Rechtzeitigkeit* eingesetzt werden kann, ist die *Sanduhr*. Im Gegensatz zum Wecker bietet diese keine gute Möglichkeit einer exakten Abbildung der eingesetzten Skala. Ihre Stärke liegt jedoch darin, dass der Benutzer sofort erkennt, wann eine *gute Rechtzeitigkeit* und wann eine *schlechte Rechtzeitigkeit* erreicht wird. Sie kann also vor allem für einen schnellen Soll-Ist-Vergleich eingesetzt werden.

Abbildung 3-26 zeigt zwei Beispiele. Der Datenqualitätswert wird hier durch Sand repräsentiert. Dieser ist bei *guter Rechtzeitigkeit* oberhalb der Verengung und bei *schlechter Rechtzeitigkeit* darunter. Die Verengung wiederum repräsentiert den Schwellenwert, der mindestens erreicht werden muss für eine *gute Rechtzeitigkeit*. Es können somit zwei Bilder entstehen. Bei *guter Rechtzeitigkeit* ist der Sand oberhalb und bei *schlechter Rechtzeitigkeit* darunter. Wie schon in den Visualisierungen zuvor, werden auch hier die Farben Rot und Grün zur Verdeutlichung eingesetzt und unterstützen dabei die Wahrnehmung.

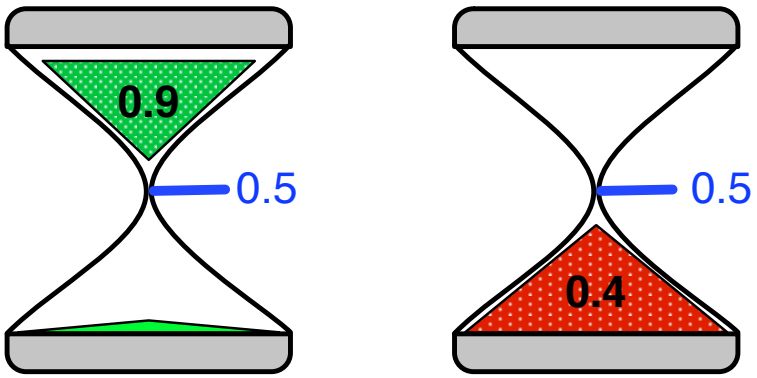


Abbildung 3-26: Rechtzeitigkeit als Sanduhr ohne Visualisierung der Skala und Schwellenwert von 0,5

3.9.2.3 Vollständigkeit (Completeness)

Für die Dimension *Vollständigkeit* werden in diesem Abschnitt wieder zwei mögliche Visualisierungen gezeigt, die sich im praktischen Versuch als nützlich herausgestellt haben. Als erstes folgt hier eine Abbildung auf einen Zylinder. Dieser ist durch seine vielen Gestaltungsmöglichkeiten besonders gut geeignet Zahlenwerte und die dazugehörige Skala abzubilden. Abbildung 3-27 zeigt zwei Beispiele. Links *gute Vollständigkeit* und rechts *schlechte Vollständigkeit*. Dabei werden die einzelnen Skalenwerte als waagerechte Linien horizontal gestapelt. Die *Vollständigkeit* wird durch die entsprechende Füllung des Zylinders symbolisiert und mit Hilfe einer etwas dickeren Linie in Kombination mit dem Zahlenwert abgetragen. Die übrigen Linien sollen den Abstand bis zur vollständigen Füllung verdeutlichen. Diese Darstellung erlaubt es zudem den Schwellenwert, hier als blaue Linie, einzuzeichnen und so einen effektiven und effizienten Soll-Ist-Vergleich durchzuführen. Zusätzlich werden die Farben Grün für *gute Vollständigkeit* und Rot für *schlechte Vollständigkeit* eingesetzt.

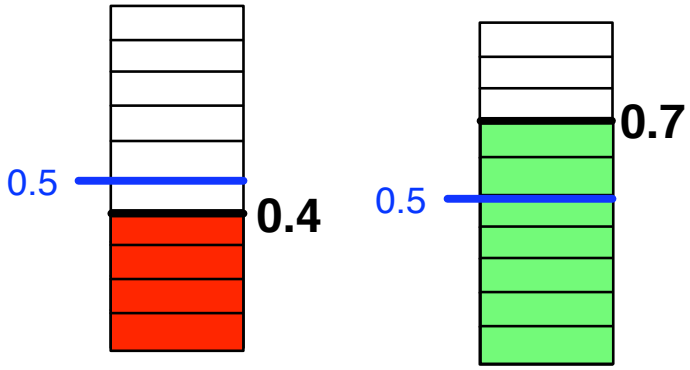


Abbildung 3-27: Vollständigkeit Säule mit Skala und mit Schwellenwert von 0,5

Ist die genaue Skala eher uninteressant, können für eine Visualisierung der Dimension *Vollständigkeit* die Werte, ähnlich einem Kuchendiagram, auf einem Kreis abgetragen werden. Der Unterschied besteht darin, dass die fehlende Werte nicht mit eingezeichnet werden und ein Lücke entsteht. Dank den Fähigkeit des menschlichen Sehsystems *Lücken* zu füllen, entsteht auf diese Weise eine expressive Darstellung. Auch hier wird der Schwellenwert als blaue Linie eingezeichnet und hilft somit bei der Bewertung des Datenwertes. Für die genaue Untersuchung kann der tatsächlich erreichte Datenqualitätswert unter das Bild oder, wie in den anderen Visualisierungen auch, direkt ins Bild eingetragen werden. Die Farben Rot und Grün werden wieder für die Verdeutlichung genützt.

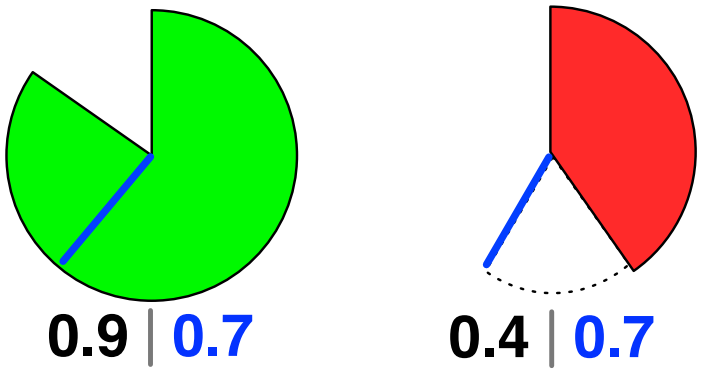


Abbildung 3-28: Vollständigkeit ohne Skala und mit Schwellenwert von 0,7

3.9.2.4 Konsistenz (Consistency)

Bisher waren die vorgestellten Visualisierungen intuitiv und vertraut. Mit der Datenqualitätsdimension *Konsistenz* werden die meisten Menschen jedoch auf Anhieb keine passende Metapher oder ein zugehöriges Bild verbinden. Um auch hier eine Visualisierung zu finden, welche den oben genannten Ansprüchen genügt, wird zuerst etwas näher auf den Begriff Konsistenz eingegangen.
Die Definition des Begriffes *Konsistenz* beschreibt sie in der Wissenschaft als „*Grad und Art des Zusammenhalts eines Stoffes*“ und in der Logik als „*strenger gedanklicher Zusammenhalt*“ [37]. Konsistenz beschreibt demnach den Zusammenhalt mehrerer Elemente innerhalb einer bestimmten Menge. Aus diesem Verständnis kann die folgende Visualisierung abgeleitet werden.

Ähnlich der Visualisierung von *Genauigkeit* bildet die *Zielscheibe* als Metapher die Ausgangsform. Darauf kann leicht die verwendete Skala abgebildet werden. Der Schwellenwert kann ebenfalls einfach eingezeichnet werden. Auf die Zielscheibe werden vier Punkte eingezeichnet. Alle verteilt auf dem selben Ring und mit gleichem Abstand zueinander. Der Ring repräsentiert dabei den erreichten Datenqualitätswert. Das bedeutet, bei einer *guten Konsistenz* liegen die Punkte dicht bei einander und bei einer schlechten Konsistenz weiter auseinander. Die Farben Grün und Rot werden, wie immer, zur Kennzeichnung von *guter Konsistenz* und *schlechter Konsistenz* als verstärkendes Mittel eingesetzt. Abbildung 3-29 zeigt ein Beispiel.

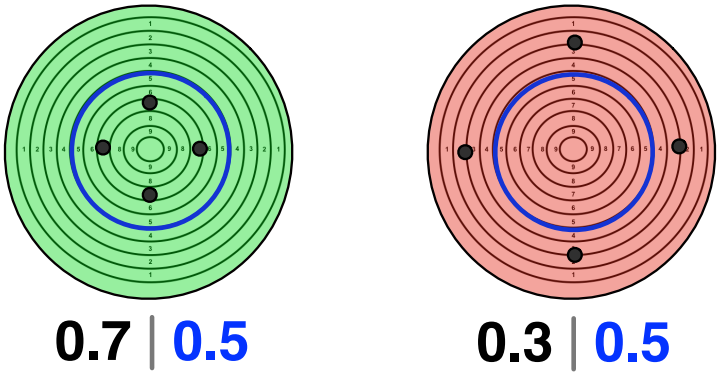


Abbildung 3-29: Konsistenz Zielscheibe mit Skala und mit Schwellenwert von 0,5

Ist die betrachtete Skala beim *Filtering* aus der Visualisierungspipeline ausgeschlossen worden, das heißt nur der berechnete Datenqualitätswert ist von Bedeutung und nicht die genaue Zuordnung, so kann die Zielscheibe wie in Abbildung 3-30 zu sehen entsprechend angepasst werden. Hier werden die einzelnen Punkte nach einem speziellen Algorithmus im Kreis verteilt. Wobei ein enger Zusammenschluss als *gute Konsistenz* und eine weitgehend freie Verteilung als *schlechte Konsistenz* zu werten sind. Der blaue Kreis zeigt wieder den Schwellenwert an. Jedoch ohne genauen Wert um eine binäre Bewertung, gut oder schlecht, geben zu können. Liegen alle eingezeichneten Punkte darin, dann ist es eine *gute Konsistenz*, liegen einige außerhalb eine *schlechte Konsistenz.* Mit der Anzahl von innen und außen liegenden Punkte kann so auch eine Einschätzung der Konsistenz gegeben werden. Um trotz fehlender Skala eine wissenschaftliche Untersuchung zu ermöglichen, werden die exakten Datenqualitätswerte unterhalb des Bildes geschrieben.

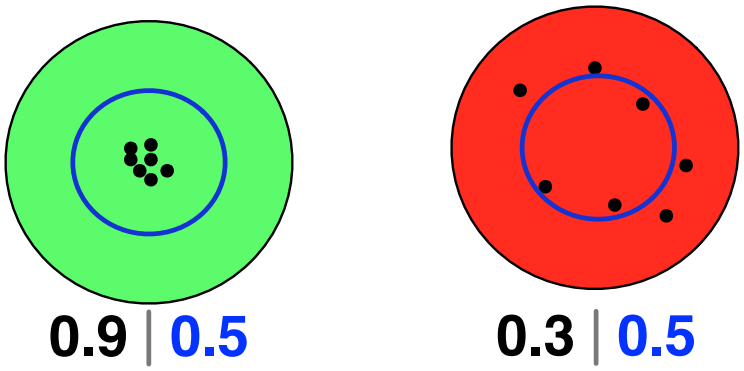


Abbildung 3-30: Konsistenz ohne Skala und Schwellenwert

3.9.2.5 Aktualität (Currency)

Für die Dimension *Aktualität* gibt es erneut keine intuitive und vertraute Metapher die sich als Basissymbol für die Visualisierung eignet. Ein dennoch weit verbreitetes Symbol, das vor allem bei Internetbrowsern häufig eingesetzt wird, ist ein Kreis mit einem

Pfeil am Ende. Dieses Symbol wird im Folgenden als Basissymbol verwendet, da es viel Platz für Informationen und eine Reihe von Anpassungsmöglichkeiten bietet. Dabei wird, je nach erreichtem Datenqualitätswert, der Außenkreis mehr oder weniger farbig gezeichnet. Die errechnete Aktualität wird an das Ende der farbigen Markierung des Außenkreises geschrieben. Zusammen mit dem eingezeichneten Schwellenwert lässt sich so leicht ein Soll-Ist-Vergleich durchführen. Zusätzlich entscheidet der Schwellenwert, wann die Füllung des Außenkreises grün gefärbt wird und wann rot.
Je nach Anforderung ist es zudem möglich die Skala auf der Datenqualitätswert liegt mit einzuzeichnen.

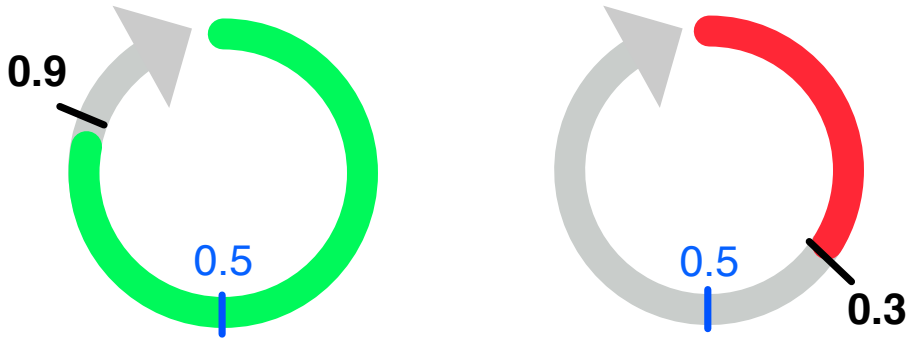


Abbildung 3-31: Aktualität ohne Skala und mit Schwellenwert von 0,5

3.9.2.6 Schwankungsfreudigkeit (Volatility)

Für die Datenqualitätsdimension *Schwankungsfreudigkeit* gibt es mehrere passende Metaphern, welche jeweils unterschiedliche Möglichkeiten der Anpassung erlauben. Als erstes wird im folgenden eine Visualisierung beschrieben, die sich an der Anzeige für Aktienkurse orientiert. Abbildung 3-32 zeigt zwei Beispiele. Bei *guter Schwankungsfreudigkeit* wird eine Sinuskurve gezeichnet die sich flach und gleichmäßig um eine waagerechte Linie windet. Bei *schlechter Schwankungsfreudigkeit* hingegen, wird eine zitternde, unregelmäßige Kurve gezeichnet die sich ebenfalls entlang einer waagerechten Linie bewegt. Der genaue Datenwert wird unterhalb des Bildes geschrieben. Die Farben Grün und Rot werden wie bisher unterstützend eingesetzt.

Diese Form der Visualisierung hat aber neben der leichten Erkennung und Zuordnung der Werte zwei Nachteile. Zum einen kann die Skala, auf der sich der Datenqualitätswert befindet nicht mit abgebildet werden und zum anderen kann der Schwellenwert, welcher entscheidet ob es sich um eine *gute* oder *schlechte Schwankungsfreudigkeit* handelt, ebenfalls nicht eingezeichnet werden.
Werden diese Faktoren außer acht gelassen, so bieten diese Metapher aber eine effektive und leichte Einordung und Bewertung des Sachverhaltes.

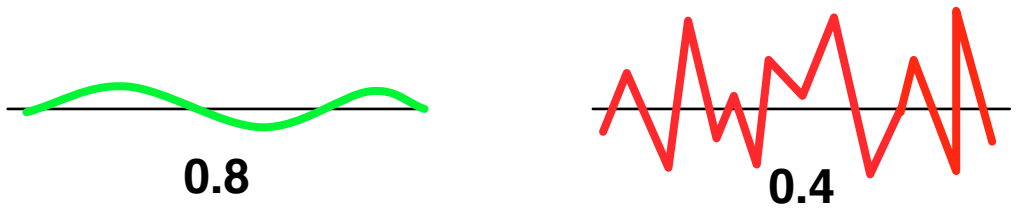


Abbildung 3-32: Schwankungsfreudigkeit ohne Skala und Schwellenwert

Ist das Anzeigen des Schwellenwertes und der Skala von Bedeutung, lassen sich basierend auf dem Symbol des *Pendels* weitere Visualisierungen entwickeln, welche die oben genannten Anforderungen erfüllen. Abbildung 3-33 zeigt zwei Beispiele. Die Grafik besteht aus einem Halbkreis, welcher die maximale Pendelbewegung repräsentiert, einem Pfeil, der den aktuellen Datenqualitätswert anzeigt, einer blauen Linie, welche den Schwellenwert zum Vergleich anzeigt und dem genauen Wert, der wieder unterhalb des Bildes angezeigt wird. Zudem werden wieder die Farben Rot und Grün als verstärkendes Mittel eingesetzt.
Diese Form der Visualisierung bietet viel Raum für spezifische Anpassungen. Beispielsweise ließe sich die Skala leicht einzeichnen oder der Schwellenwert entfernen.

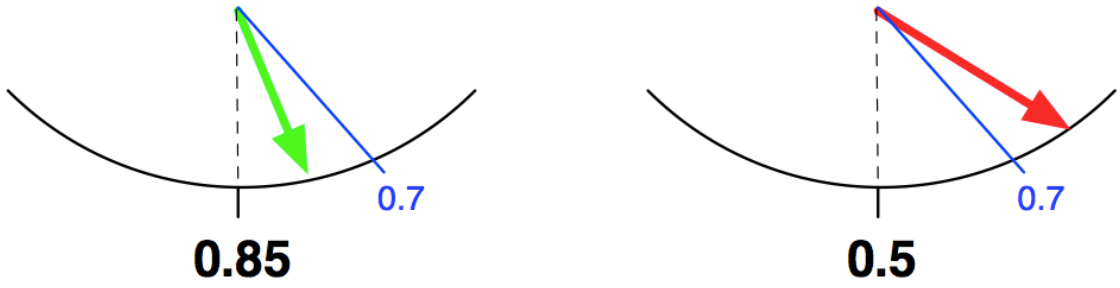


Abbildung 3-33: Schwankungsfreudigkeit als Pendel ohne Skala und Schwellenwert von 0,7

3.9.2.7 Allgemeine Diagramme

Die in den voran gegangenen Abschnitten vorgestellten Visualisierungen stellen Spezialfälle dar, die sich vor allem zur Darstellung einzelner Werte ohne Bezug zu einander eignen. In vielen Fällen stehen die verschiedenen Datenqualitätswerte jedoch in Beziehung zu einander, wie beispielsweise die *Rechtzeitigkeit* und *Genauigkeit*. Wenn die Simulation hohe Ansprüche an die *Genauigkeit* der einzelnen Datenwerte stellt, so kann es vorkommen, dass Komponenten länger für eine Berechnung benötigen und somit die Ergebnisse nicht innerhalb eines geforderten Zeitraums der nächsten Komponente zur Verfügung stellen. Um solche Beziehungen visuell auszudrücken zu können, müssen die Datenqualitätswerte auf einer einheitlichen Skala mit einheitlich definierten visuellen

Attributen abgebildet werden. Nur so ist eine effektive Bearbeitung durch den Benutzer gewährleistet.
Auch wenn einzelne *Services* im Bezug auf ihre Datenqualität über einen Zeitraum betrachtet werden sollen, bieten sich die einfachen Standarddiagramme, wie Kurven- und Liniendiagramme aus Kapitel 3.7.2.1 an.

4 Anforderungen an das Visualisierungsframework

Dieses Kapitel beschreibt die Anforderungen an das Visualisierungsframework, welches zur Unterstützung der wissenschaftlichen Auswertung von Datenqualitätswerten in Simulation-Workflows eingesetzt wird.

Im ersten Teil werden zunächst die allgemeinen Anforderungen formuliert. Die Struktur der nachfolgenden Kapitel orientiert sich anschließend an den beiden Benutzergruppen des Visualisierungsframeworks, „*Wissenschaftler*“ und „*Programmierer*“. Dazu werden im zweiten Teil die Anforderungen der Wissenschaftler, als direkte Anwender eines WebServices auf Basis des Visualisierungsframeworks, gezeigt.
Im dritten Abschnitt werden die Anforderungen seitens der Entwickler von Erweiterungen für das Visualisierungsframework formuliert.

Alle Anforderungen werden für das spätere Referenzieren durch das Symbol $\Re$ (*Requirement*) nummeriert.

4.1 Allgemeine Anforderungen an das Framework

Dieses Kapitel beschreibt zunächst die allgemeinen Anforderungen an das Visualisierungsframework.

4.1.1 Wiederverwendbarkeit ($\Re$1)

Das Framework soll so entworfen werden, dass es leicht von einer Simulation-Workflow-Umgebung zu einer anderen portiert werden kann.

4.1.2 Anbindung an das JDQCF ($\Re$2)

Das Framework soll das JDQCF (siehe Kapitel 2.3.3) als Datenquelle für die Visualisierungen verwenden. So können die durch das JDQCF generierten Datenqualitätswerte verarbeitet werden.

4.1.3 Anbindung an externe Datenquellen ($\Re$3)

Simulationsdaten können auf einem externen Server liegen. Um diese Daten trotzdem verarbeiten zu können, soll das Framework die Möglichkeit bieten, referenzierte Daten zu laden und in dem Visualisierungsprozess zu berücksichtigen. Beispielsweise können diese Rohdaten durch einen *Data as a Service (DaaS)* [40] zur Verfügung gestellt werden. Das JDQCF übergibt dann lediglich die Adresse unter welche die Daten breitgestellt werden.

4.1.4 Unterstützung mehrere Simulationen (ℜ4)

Das Framework soll mehrere laufende Simulationen unterscheiden können. Somit kann ein Wissenschaftler jede seiner Simulationen gleichzeitig überwachen.

4.1.5 Verarbeitung unterschiedlicher Datenstrukturen (ℜ5)

Das Framework wird in unterschiedlichsten Fachbereichen eingesetzt. Damit verbunden können unterschiedliche Daten entstehen. Um die größtmögliche Flexibilität zu gewährleisten, soll das Framework unterschiedliche Datenstrukturen verarbeiten können.

4.2 Anforderungen aus Sicht der Wissenschaftler

Dieses Kapitel beschreibt die Anforderungen aus Sicht der Wissenschaftler. Diese haben als Endbenutzer vor allem Anforderungen an die Benutzerverwaltung und die generierten Visualisierungen.

4.2.1 Anforderungen an die Benutzerverwaltung

Dieser Abschnitt beschreibt die verschiedenen Anforderungen der Wissenschaftler an die Benutzerverwaltung der Visualisierungsframeworks.

4.2.1.1 Verwaltung von mehreren Benutzern (ℜ6)

Das Framework soll die Verwaltung von mehreren gleichzeitig angemeldeten Wissenschaftlern unterstützen.

4.2.1.2 Autorisierung des Benutzers (ℜ7)

Um die Zugriffssicherheit auf die laufende Simulation und deren Daten zu gewährleisten, soll das Framework die Möglichkeit bieten, eine simulationsabhängige Benutzerautorisierung einzubinden. Es soll sichergestellt werden, dass nur berechtigte Personen die Daten verarbeiten können.

4.2.1.3 Unterstützung bei Steuerung der Simulation durch den Benutzer (ℜ8)

Das Framework soll eine Schnittstelle für die Steuerung einer laufenden Simulation bereitstellen. Das bedeutet, dass es dem Wissenschaftler die grafischen Hilfsmittel dazu bereit stellt, beispielsweise einen *„Simulation Abbrechen“-Button.* Dabei soll das Visualisierungsframework diese Aktion nicht implementieren. Es soll die konkrete Aktion, zum

Beispiel das *Drücken* dieses Buttons, über ein Schnittstelle an eine externe Stelle weiterleiten, an der sie entsprechend verarbeitet wird.

4.2.1.4 Verwendung eines rollenbasierten Systems zur Regelung der Benutzerinteraktionen (ℜ9)

In Simulation-Workflows gibt es spezielle Rollenverteilungen. So gibt es im Allgemeinen eine Person die für die Simulation verantwortlich ist (*Simulation Owner*) und somit alle Rechte für die Interaktion mit dieser besitzt. Neben ihr gibt es meist noch weitere Wissenschaftler mit unterschiedlichen Rollen und Rechten an der Simulation.
Diese Struktur soll sich in dem Framework wiederspiegeln, so dass unterschiedliche Rollen, ihren Rechten entsprechende Interaktionsmöglichkeiten besitzen. Beispielsweise besitzt der *Simulation Owner* alle Rechte an der Simulation und darf in alle Stufen der Visualisierungspipeline oder in die laufende Simulation eingreifen. Andere Wissenschaftler, mit weniger Rechten, können diese Steuerungsmöglichkeiten nur eingeschränkt oder gar nicht nutzen.

4.2.2 Anforderungen an die Visualisierungskomponente

Dieses Kapitel beschreibt die Anforderungen an die Visualisierungskomponente des Frameworks.

4.2.2.1 Berücksichtigung der Rolle des Wissenschaftlers bei der Visualisierung (ℜ10)

Das Framework soll die Rolle eines Wissenschaftler in einer Simulation bei der Generierung der Visualisierungen berücksichtigen. Das bedeutet, dass unterschiedliche Rollen zu unterschiedlichen Visulisierungen führen können. Beispielsweise werden für den *Simulation Owner,* neben den Datenqualitätswerten, zusätzlich die eigentlichen Simulationsdaten visualisiert. Bei anderen Wissenschaftlern mit weniger Rechten werden hingegen nur die Datenqualitätswerte zur Überwachung angezeigt.

4.2.2.2 Generierung expressiver, effektiver und angemessener Datenqualitätsvisualisierungen (ℜ11)

Das Framework soll aus den empfangenen Daten Visualisierungen generieren können, welche die in Kapitel 3.4 gezeigten Anforderungen erfüllen. Dazu gehören insbesondere das Generieren expressiver, effektiver und angemessener Datenqualitätsvisualisierungen.

4.2.2.3 Gerüst für die unterschiedlichen Stufen der Visualisierungspipeline (ℜ12)

Wie in Kapitel 3.3.4 gezeigt, können die einzelnen Stufen der Visualisierungspipeline auf verschiedenen Rechnern und durch unterschiedliche Programmiersprachen realisiert werden. Das Framework soll die unterschiedlichen Varianten zur Verteilung dieser Stufen unterstützen.

4.2.2.4 Unterstützung unterschiedlicher Ausgabegeräte (ℜ13)

Die generierten Visualisierungen sollen auf unterschiedlichen Anzeigegeräten präsentiert werden können. Das bedeutet, dass das Visualisierungsframework die unterschiedlichen Geräteeigenschaften bei der Generierung der Visualisierungen berücksichtigt.

4.2.2.5 Steuerung der Visualisierungspipeline durch den Benutzer (ℜ14)

Das Framework soll dem Benutzer die Möglichkeit geben, die Generierung der Visualisierungen interaktiv zu beeinflussen. Beispielsweise soll dieser die Möglichkeit haben zu entscheiden ob er eine Übersicht aller Datenqualitätsvisualisierungen mit oder ohne den dazugehörigen Simulationsdaten dargestellt bekommt.

4.2.3 Anforderungen an das Verteilen der Daten

Neben der Benutzersteuerung und Visualisierung ist das Verteilen der generierten Visualisierungen von Bedeutung.

4.2.3.1 Unterstützung von unterschiedlichen Kommunikationsprotokollen (ℜ15)

Je nach gewähltem Anzeigegerät können unterschiedliche Protokolle für das Versenden der Daten notwendig sein. Hierzu soll das Framework unterschiedliche Versandarten wie *SOAP with Attachment* Nachrichten oder das Ablegen der Visualisierungen auf einem Server unterstützen.

4.3 Anforderungen aus Sicht der Programmierer

Dieses Kapitel beschreibt die Anforderungen an das Visualisierungsframework aus Sicht der Entwickler. Diese gliedern sich in funktionale- und nichtfunktionale Anforderungen.

4.3.1 Funktionale Anforderungen

In diesem Abschnitt werden die funktionalen Anforderungen der Programmierer an das Visualisierungsframework formuliert.

4.3.1.1 Das Framework soll einen einfachen Rahmen für die Entwicklung von Erweiterungen bereitstellen (ℜ16)

Das Framework soll einen einfachen Rahmen für das Erweitern bereitstellen. So sollen Entwickler ihre Algorithmen leicht einbinden können, ohne das gesamte Framework anpassen zu müssen.

4.3.1.2 Trennung der Schnittstellen zur Benutzerautorisierung und der Visualisierungskomponente (ℜ17)

Da es sich bei der Entwicklung der Benutzerautorisierung und der Visualisierungskomponenten um unterschiedliche Problemstellungen handelt, soll das Framework diesen zweigeteilten Charakter in seiner Architektur und seinen Schnittstellenbeschreibungen berücksichtigen.

4.3.2 Nichtfunktonale Anforderung

Dieser Abschnitt zeigt die nichtfunktionale Anforderung der Entwickler an das Visualisierungsframework.

4.3.2.1 Lose Kopplung der Komponenten (ℜ18)

Der flexible Einsatz der Frameworks soll sich durch eine lose Kopplung der einzelnen Komponenten leicht realisieren lassen. So sollen einzelne Komponenten leicht an neue Bedingungen anpassbar oder austauschbar sein.

5 Konzeptioneller Entwurf des Java Data Quality Visualization Framework

Ziel dieser Arbeit ist es die Wissenschaftler bei der Überwachung von laufenden Simulationen durch die Visualisierung der Datenqualität zu unterstützen. In diesem Kapitel wird dazu der konzeptionelle Entwurf des *Java Data Quality Visualization Framework* (*JDQVisF*) erarbeitet. Es werden zunächst die Architektur und die entsprechenden Komponenten entworfen und schließlich zu einem Gesamtsystem zusammengefügt.

Für das leichtere Verständnis und zur Vermeidung von Verwechslungen werden im folgenden das Framework und eine Implementierung auf Basis dieses Frameworks gleichgesetzt und unter dem Begriff *JDQVisF* zusammengefasst.
Ein Wissenschaftler mit seinem Anzeigegerät wird unter dem Begriff *JDQVisClient* verwendet.
Ein Programmierer der Visualisierungen realisiert, wird als *Visualisierer* bezeichnet.
Das Symbol $\Re$ verweist auf die jeweilig Anforderung aus Kapitel 4.

Zudem werden die einzelnen Komponenten in allen Abbildungen mit den selben Farben kodiert. Dadurch können die zum Teil komplexen Zusammenhänge der Komponenten leichter verstanden werden und der Gesamtüberblick bewahrt werden.

Alle im Folgenden vorgestellten Konzepte und Architekturentscheidungen entstanden durch Rücksprachen mit dem Betreuer.

5.1 Erweiterung des bisherigen Simulationskontextes

Um die Anforderungen an das JDQVisF umsetzen zu können, ist es notwendig den bisherigen Simulationskontext um eine *SimulationId* und *UserId* zu erweitern.

In einer Simulation kann es mehrere Wissenschaftler bzw. *JDQVisClienten* mit unterschiedlichen Rollen und Rechten geben [25]. So gibt es typischerweise einen Wissenschaftler (*Simulation-Owner*) der die Verantwortung für die Simulation trägt und somit alle Rechte an ihr besitzt. Es kann jedoch zusätzliche Mitarbeiter (*Domänen-Spezialisten* oder *Hilfswissenschaftler*) geben, welche weniger Rechte an der gesamten Simulation besitzen und nur für ein Teilgebiet zuständig sind. Es entsteht durch diesem Zusammenhang eine 1:n Simulation-Wissenschaftler-Beziehung.

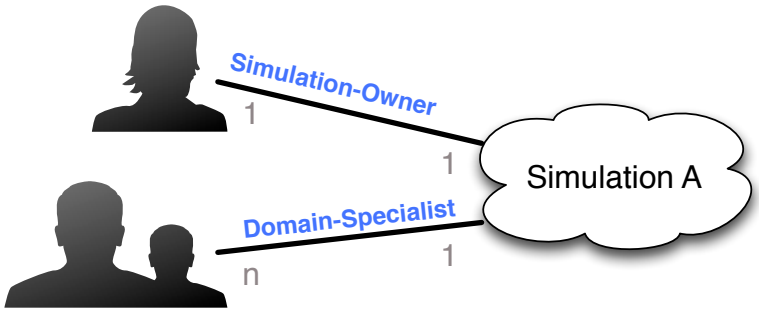


Abbildung 5-1: Simulation-Wissenschaftler-Beziehung. Zu einer Simulation kann es mehrere Wissenschaftler mit unterschiedlichen Rollen geben

Auf der anderen Seite kann ein Wissenschaftler mit verschiedenen Rollen an mehreren Simulationen arbeiten. Es entsteht eine 1:m Wissenschaftler-Simulation-Beziehung.

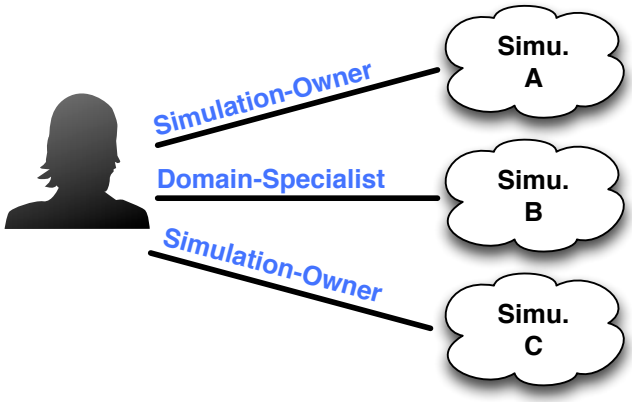


Abbildung 5-2: Wissenschaftler-Simulation-Beziehung. Ein Wissenschaftler kann mehrere Simulationen mit unterschiedlichen Rollen betreuen

Kombiniert man diese beiden Zusammenhänge, ergibt sich eine m:n Wissenschaftler-Simulation-Beziehung.

Damit das JDQVisF diese Beziehung korrekt auflösen und verarbeiten kann, benötigt es sowohl für die Simulation als auch für den Wissenschaftler eine Identifikationsmöglichkeit. Aus diesem Grund wird der bisherige Simulationskontext um eine *SimulationId* und einer *UserId* erweitert. Dies ermöglicht das Einbinden von mehreren Simulationen in das *JDQVisF* (ℜ4) und die Verarbeitung mehrerer Benutzern mit unterschiedlichen Rollen (ℜ6 und ℜ9). Abbildung 5-3 zeigt den Zusammenhang.

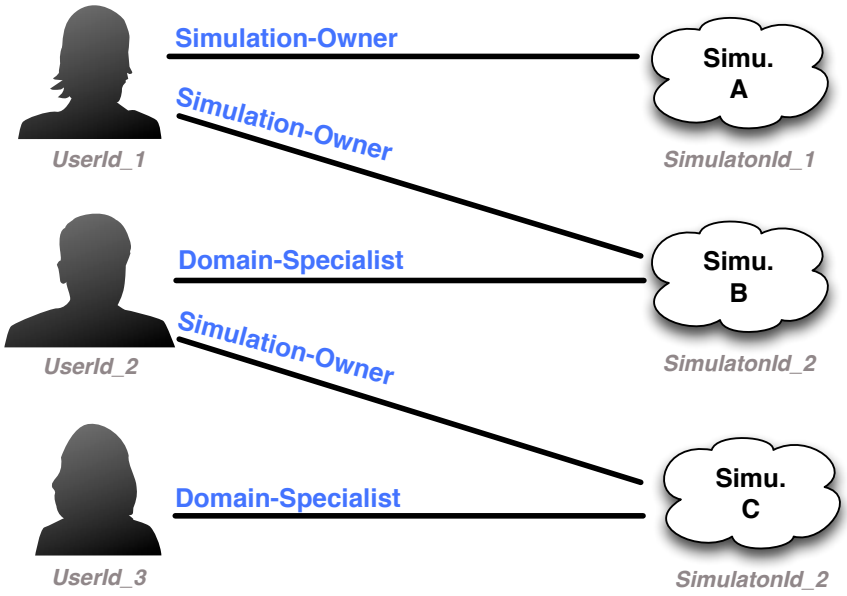


Abbildung 5-3: Beispielhafte Wissenschaftler-Simulation-Beziehungen mit UserIds und SimulationIds zur Identifikation. Ein Wissenschaftler kann an mehrere Simulationen arbeiten und eine Simulation kann von mehreren Wissenschaftlern in unterschiedlichen Rollen betreut werden

Wie das JDQVisF die *SimulationId* und *UserId* genau verwendet, wird in den nachfolgenden Kapiteln bei der Beschreibung der Gesamtarchitektur und der einzelnen Komponenten gezeigt.

5.2 Grundsätzliche Architektur

Um die in Kapitel 4 gezeigten Anforderungen zu erfüllen und den unterschiedlichen Benutzergruppen des JDQVisF gerecht zu werden, wird die Architektur aus zwei verschiedenen Sichtweisen heraus aufgebaut. Zum einen aus Sicht der Programmierer des Frameworks und somit als direkter Benutzer. Zum anderen aus Sicht der Wissenschaftler als Endbenutzer des JDQVisF.

Neben den funktionalen Anforderungen stehen vor allem Flexibilität, Einfachheit, Robustheit und Wiederverwendbarkeit der Komponenten im Vordergrund. Wobei Flexibilität bedeutet, dass das JDQVisF betriebssystemunabhängig eingesetzt und leicht an neue Anforderungen angepasst werden kann. Einfachheit bezieht sich sowohl auf die Implementierung, als auch auf das Verwenden des JDQVisF.

Als grundsätzliche Architekturentscheidung wird eine Client-Server-Architektur (siehe Abbildung 5-4) gewählt. Diese bringt sowohl aus Sicht der Wissenschaftler (*JDQVisClient*) als auch aus Sicht der Programmierer (*Visualizer*) viele Vorteile mit sich, die in den folgenden Abschnitten gezeigt werden.

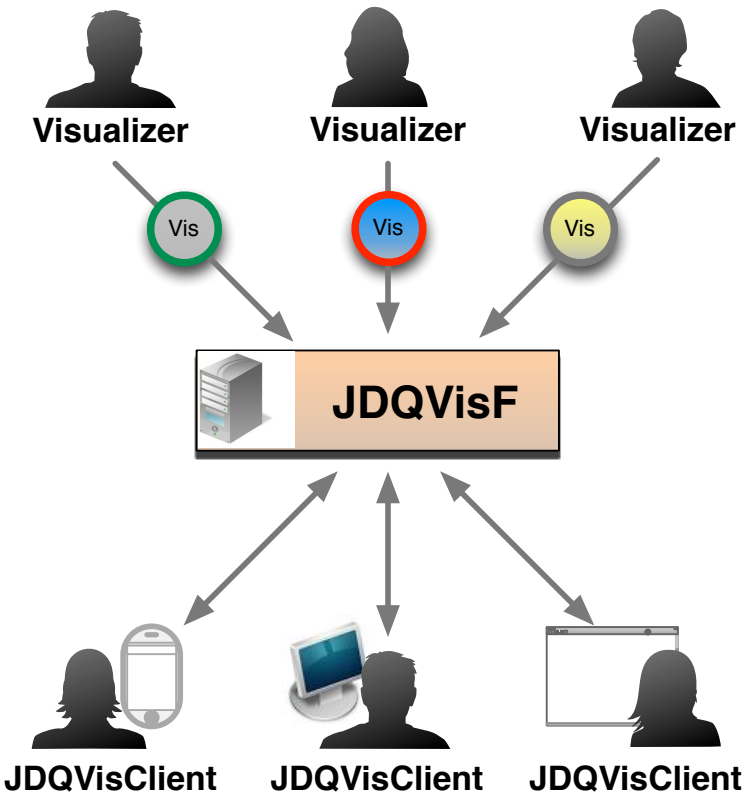


Abbildung 5-4: Client-Server Architektur mit Unterscheidung der Anwender des JDQVisF. Oben sind die Entwickler, die ihre Algorithmen in das JDQVisF einpflegen, unten die JDQVisClients

Den *JDQVisClients* bietet das JDQVisF als WebService eine zentrale Stelle für die Anmeldung und entkoppelt sie dadurch vom restlichen Simulation-Workflow.
Ein Vorteil der Server-Komponente, ist die Einführung einer zentralen Benutzersteuerung. Die verschiedenen Client-Programme müssen dadurch nicht die Zugriffsrechte des jeweiligen Wissenschaftler beachten, was eine vereinfachte Implementierung ermöglicht. Das JDQVisF bietet eine gemeinsame Komponente für die Konfiguration und Wartung der Benutzersteuerung, was das Hinzufügen neuer Wissenschaftler zu einer Simulation oder das Anpassen der Rechte eines Wissenschaftlers an einer Simulation vereinfacht.
Durch die Entkopplung der Clientprogramme von den Simulation-Workflows, lassen sich diese simulationsunabhängig einsetzten (ℜ5). Die jeweiligen Autorisierungsfunktionen werden im JDQVisF zusammengefasst und ermöglichen dadurch eine simulationsabhängige Anmeldung jedes JDQVisClients ohne lokale Überprüfung.

Die Architektur unterstützt neben den unterschiedlichen JDQVisClients auch mehrerer Simulationen. Ein Wissenschaftler muss eine neue Simulation nur einmal beim JDQVisF registrieren und kann anschließend beliebig viele Anzeigegeräte für deren Überwachung einsetzen, ohne diese anpassen zu müssen.

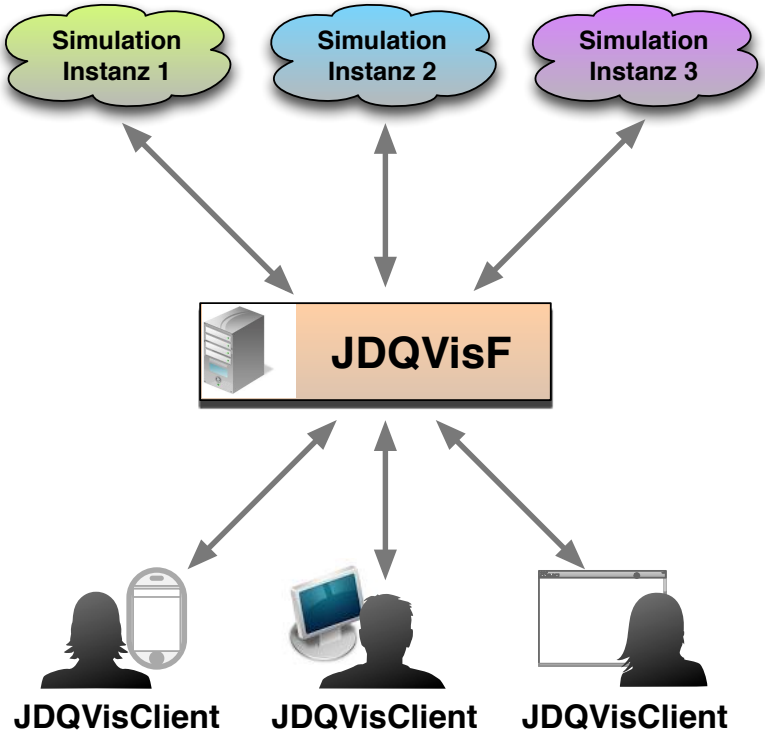


Abbildung 5-5: JDQVisF als zentrale Komponente zwischen verschiedenen Simulationen und JDQVisClients

Ein weiterer Grund der zu dieser Client-Server-Architektur führt, ist die Notwendigkeit für die Berücksichtigung der Rolle eines Wissenschaftlers in einer Simulation (ℜ9). Das

JDQVisF berücksichtigt diesen Zusammenhang bei der Generierung von Visualisierungen, so dass eine rollenbasierte Visualisierung auf Clientseite entfällt.
Durch das Auslagern des rollenbasierten Rechtesystems auf Serverseite, wird zudem das kontrollierte Eingreifen in die Simulation durch den Wissenschaftler ermöglicht (ℜ14). Das Clientprogramm muss die Rechte des aktuell angemeldeten Wissenschaftler nicht kennen, was zu einer weiteren Vereinfachung führt.

Allgemein entlastet diese Architektur das Anzeigegerät durch die Verarbeitung der Rohdaten und die Generierung der Visualisierungen auf Serverseite. Das JDQVisF bietet den Wissenschaftlern die Möglichkeit, sowohl leistungsschwache als auch leistungsstarke Anzeigegeräte zu verwenden (ℜ13). So kann beispielsweise ein Tablet-Computer komplexere Visualisierungen verarbeiten und das Rendering lokal ausführen. In diesem Fall würde das JDQVisF nur das *Filtern* und *Mappen* der Daten übernehmen und das berechnete Geometriemodell an das Anzeigegerät versenden. Der Tablet-Computer übernimmt anschließend lokal das *Rendering*. Bei einem schwachen Anzeigegeräten, etwa einem Internetbrowser, kann das JDQVisF alle Schritte der Visualisierungspipeline übernehmen und fertige Bilddateien zur Anzeige bereitstellen.
Das JDQVisF unterstützt somit alle Möglichkeiten zur Verteilung der Schritte der Visualisierungspipeline (ℜ12).

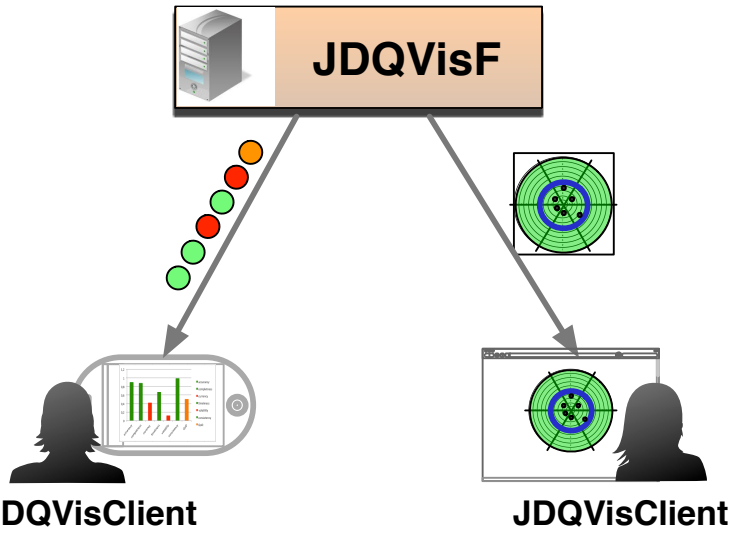


Abbildung 5-6: Unterstützung unterschiedlicher Anzeigegeräte durch das JDQVisF: (links) lokales Rendering des Visualisierungsmodells. (rechts) anzeige des fertigen Bildes

Durch das Auslagern der Visualisierungskomponente vom Clientgerät auf das JDQVisF können auch leicht neue Anzeigegeräte verwendet werden, ohne dass neue Visualisierungsalgorithmen auf Clientseite implementiert werden müssen. Es können bereits vorhandene Visualisierungsalgorithmen des JDQVisF genutzt werden. Vorausgesetzt sie passen zu dem neuen Anzeigegerät. Wird für ein Anzeigegerät ein spezieller Visualisierungsalgorithmus benötigt, kann dieser zum JDQVisF hinzugefügt werden. Ist das nicht erwünscht, so übernimmt das JDQVisF lediglich die Anbindung an den Simulation-

Workflow und die Benutzersteuerung und leitet die empfangenen Daten an das Gerät weiter. Dadurch erleichtert diese Architektur eine Endgeräteunabhängigkeit, da die Verarbeitungslogik durch den Server übernommen werden kann. Zusätzlich ermöglicht diese Client-Server-Architektur das Generieren der selben Visualisierungen für Anzeigegeräte mit unterschiedlichen Betriebssystemen aber ähnlichen Geräteeigenschaften. So können beispielsweise auf einem Android-Tablet die gleichen komplexen Visualisierungen angezeigt werden wie auf einem iPad.

Aus Sicht der Visualisierer bietet diese Architektur und damit die Visualisierung auf Serverseite, ebenfalls mehrere Vorteile. Durch die Skalierbarkeit des Servers lassen sich selbst für leistungsschwache Geräte schöne und komplexe Visualisierungen generieren. Der Server kann das rechenintensive *Mapping* und *Rendering* übernehmen. Dadurch ist der Visualisierer nicht durch die fehlende Rechenleistung der Anzeigegeräte eingeschränkt.
Neben der Skalierbarkeit bietet diese Client-Server-Architektur den Visualisierern auch eine gemeinsame Stelle, an der er seine neuen Visualisierungsalgorithmen einbinden kann. Etwa wenn eine neue Simulation überwacht werden soll oder eine neue Rolle in einer Simulation hinzugefügt wird. Er ist von den Anzeigegeräten entkoppelt und muss bei einer Änderung nicht jedes verwendete Gerät anpassen. Muss ein Visualisierungsalgorithmus angepasst werden, so bietet das JDQVisF eine zentrale Komponente für die Wartung und die Pflege an.

Ein weiterer Vorteil dieser Architektur ist die Möglichkeit, Daten in einen zentralen *Cache* abzulegen. So können bei Bedarf zeitliche Verläufe visualisiert werden oder die Daten nach Simulationsende ausgewertet werden.

Bevor in den nächsten Kapiteln der genaue Aufbau der JDQVisF und die einzelnen Komponenten im Detail erarbeitet werden, wird im nächsten Abschnitt zunächst die Einordnung in den Simulation-Workflow gezeigt.

5.2.1 Einordnung des JDQVisF in den Simulation-Workflow

Eine Visualisierung hilft dem Wissenschaftler bei der Auswertung abstrakter Datenmengen, wie sie bei Simulationen entstehen. Sie bildet damit eine Schnittstelle zwischen Datenerzeuger, hier die Simulation, und Datenverarbeitung durch den Wissenschaftler. Dieser verbindende Charakter spiegelt sich bei der Einordnung des JDQVisF in den Simulation-Workflow wieder. Es gruppiert sich logisch zwischen diesen beiden Komponenten. Abbildung 5-7 zeigt den Zusammenhang:

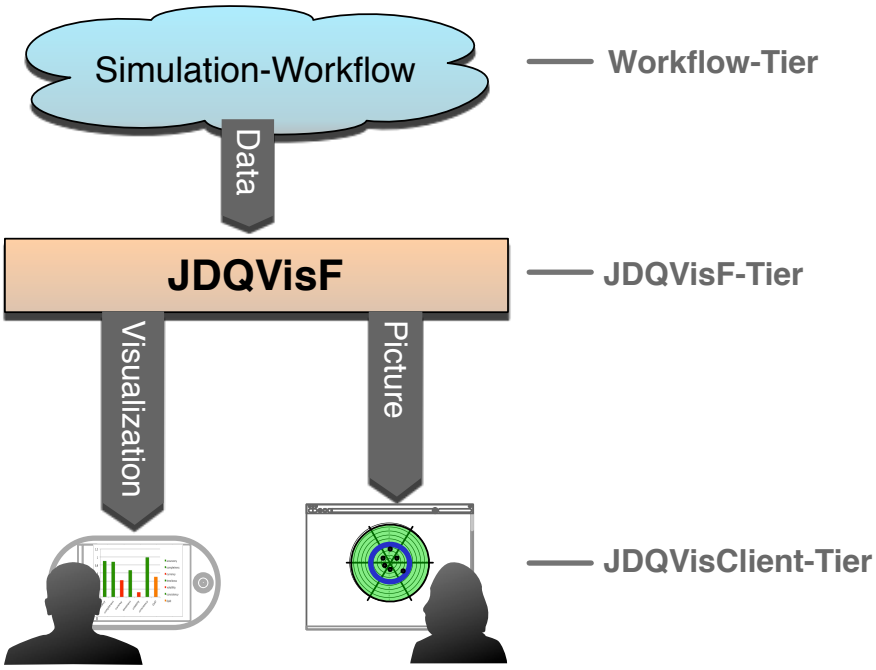


Abbildung 5-7: Einordung des JDQVisF zwischen Simulation-Workflow und Wissenschaftler

Durch diese Einordnung bildet sich eine logisch getrennte Dreischichtenarchitektur. Der Simulation-Workflow erzeugt Daten, die mit Hilfe des JDQVisF gerätespezifisch visualisiert werden und schließlich auf dem Gerät des Wissenschaftlers dargestellt werden. Auf der anderen Seite melden sich die Wissenschaftler beim JDQVisF an, um sich für Simulationen zu registrieren.

5.3 Struktureller Aufbau des JDQVisF

Grundsätzlich wird der Aufgabenbereich des JDQVisF in zwei Kategorien eingeteilt. Zum einen in Wissenschaftler- und Simulationsverwaltung und zum anderen in die Steuerung der Visualisierungspipeline. Aus diesem Grund wird die Architektur des JDQVisF in die zwei Kernkomponenten *JDQVisController* und *VisualizationMediator,* die jeweils einen Aufgabenbereich übernehmen, aufgeteilt. Der JDQVisController übernimmt die Benutzerverwaltung, die Anbindung an das JDQCF und das Weiterleiten von Steuerungsbefehlen. Der VisualizationMediator ist unabhängig von der Benutzersteuerung und übernimmt das Visualisieren und Versenden der Datenqualitätswerte.

Diese Trennung ermöglicht die Aufteilung der Schnittstellen nach ihrem Einsatzgebiet und vereinfacht die Wartung der Benutzersteuerung und das Verwalten der Simulationen durch die Wissenschaftler auf der einen Seite und die Wartung der Visualisierungspipeline durch die Visualisierer auf der anderen (ℜ17). Ein Wissenschaftler kann die Benutzersteuerung anpassen, ohne die Visualisierungskomponente zu beeinflussen. Die Visualisierer können neue Visulisierungsalgorithmen einbinden ohne die Benutzersteuerung anpassen zu müssen.

Abbildung 5-8 zeigt den strukturellen Aufbau des JDQVisF mit Anbindung an das JDQCF und einer Komponente für die Steuerung der Simulation (siehe Kapitel 5.4.1.1).

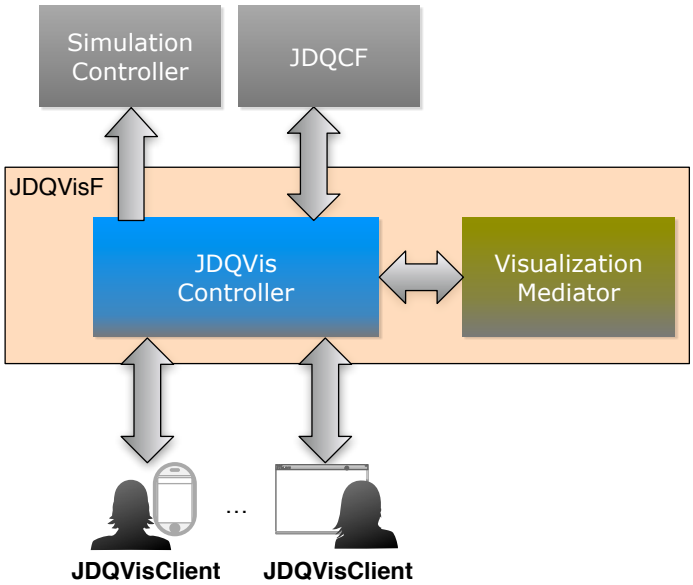


Abbildung 5-8: Struktureller Aufbau des JDQVisF mit Anbindung an das JDQCF und einem SimulationController. Die Pfeile deuten den Datenaustausch zwischen den Komponenten an

5.3.1 Plug-In Architektur

Das JDQVisF soll mehrere gleichzeitig laufende Simulationen unterstützen (ℜ4). Dabei können die Ansprüche an die Benutzerautorisierung oder an die Visualisierungsalgorithmen variieren. Ein Wissenschaftler kann an einer Simulation A andere Rechte besitzen als an einer Simulation B. Des weiteren sollen leicht neue Simulationen und Visualisierungsalgorithmen zum JDQVisF hinzugefügt werden können (ℜ13). Aus diesen Gründen können, weder der *JDQVisController* noch ein *VisualizationMediator,* aus abgeschlossenen Komponenten bestehen, die alle Simulationen und Rechte berücksichtigen. Möchte ein Wissenschaftler beispielsweise eine neue Simulation hinzufügen oder ein Visualisierer einen neuen Visualisierungsalgorithmus, müsste er die komplette Komponente verändern, was die Wahrscheinlichkeit von Fehlern erhöht und die Wartung erschwert. Die Anforderung an eine lose Kopplung der einzelnen Komponenten wird dadurch erfüllt (ℜ18).
Aus diesen Gründen wird für die Realisierung des JDQVisController und des VisualizationMediator eine Plug-In Architektur verwendet. Dabei bilden diese die beiden Kernkomponenten, die durch spezielle Schnittstellen erweitert werden können. Das ermöglicht sowohl das leichte Hinzufügen neuer Simulationen oder Rollen, als auch den Austausch von funktionalen Komponenten wie *Filter*, *Visualisierer*, *Verteiler, SimulationController* oder *Benutzerautorisierung* durch die Entwickler und bietet damit ein hohes Maß an Flexibilität (ℜ18).

Ein weiterer Grund für diese Architekturentscheidung ist die Möglichkeit, durch die Auslagerung der funktionalen Erweiterungen, neue Anforderungen seitens der Wissenschaftler oder neue Visualisierungsalgorithmen durch die Visualisierer zur Laufzeit in das JDQVisF einzubinden. Auf diese Weise werden andere laufende Simulationen nicht beeinflusst.

Eine Erweiterung wird im folgenden als *Plug-In* bezeichnet und beschreibt eine funktionale Komponente des JDQVisF. Beispielsweise sind Filter- oder Autorisierungskomponenten *Plug-Ins* die vom JDQVisF bei Bedarf geladen und an entsprechender Stelle aufgerufen werden.

Eine solche Plug-In-Architektur ermöglicht zudem die leichte Wiederverwendung des gesamten JDQVisF oder einzelner Komponenten (ℜ1).

5.4 Komponenten des JDQVisF

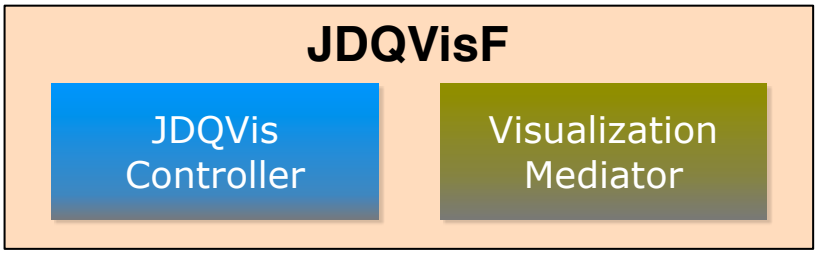


Abbildung 5-9: Interne Aufteilung des JDQVisF in die zwei Hauptkomponenten JDQVisController und VisualizationMediator

Die Architektur des JDQVisF besteht aus zwei Kernkomponenten, die zum einen für die Simulations- und Benutzersteuerung und zum anderen für die Steuerung der Visualisierungspipeline zuständig sind. Der *JDQVisController* steuert und verwaltet die verschiedenen *JDQVisClients*, koordiniert die Registrierungen am JDQCF, leitet Steuerungsbefehle an den Simulation-Workflow weiter, dient zur Realisierung der Zugriffsicherheit und verwaltet alle *VisualizationMediatoren*. Ein *VisualizationMediator* steuert die Visualisierungspipeline und versendet die generierten Visualisierungen.

Diese zweigeteilte Architektur entsteht aus den unterschiedlichen Anforderungen der Plug-In-Entwickler und der Wissenschaftler. Möchte ein Entwickler eine neue Visualisierung für das JDQVisF realisieren muss er lediglich die entsprechende Schnittstelle des *VisualizationMediator* implementieren. Durch die Trennung ist er dabei völlig unabhängig vom restlichen Aufbau des JDQVisF und kann sich ganz auf die für ihn wichtigen Stellen konzentrieren. Auf der anderen Seite meldet sich ein *JDQVisClient* beim *JDQVisController* an einer zentralen Stelle an und kann durch dessen Schnittstellen den gesamten Visualisierungsprozess nach seinen Wünschen anpassen (ℜ17). Ein direkten Eingriff in die Visualisierungsalgorithmen ist dazu nicht nötig.

Die Kommunikation zwischen dem *JDQVisController* und einem *VisualizationMediator* wird in Kapitel 0 gezeigt.

Im Folgenden werden die beiden Teilarchitekturen *JDQVisController* und *VisualizationMediator* des JDQVisF vorgestellt. Dazu wird jeweils die Architektur und die entsprechenden Plug-Ins detailliert erarbeitet.

5.4.1 Architektur des JDQVisController

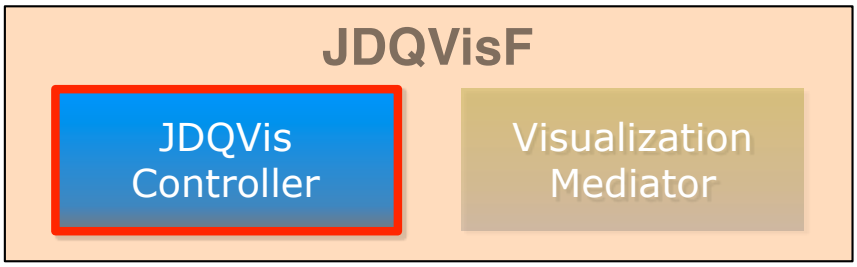


Abbildung 5-10: Komponenten des JDQVisF mit hervorgehobenem JDQVisController

Der *JDQVisController* ist die Hauptkomponente des *JDQVisF*. Er bildet eine Schicht zwischen *JDQVisClient* und Simulation-Workflow. Bei ihm können sich *JDQVisClienten* registrieren, die Visualisierungspipeline beeinflussen und Steuerungsanfragen ihrer Simulation stellen. Der *JDQVisController* realisiert die Kommunikation zwischen *JDQVisClient* und *JDQVisF*.

Der *JDQVisController* fungiert dabei als eine Kernkomponente, die über verschiedene Schnittstellen funktional erweitert werden kann (Abbildung 5-11).
Diese Architekturentscheidung begründet sich hauptsächlich durch die Anforderung an das JDQVisF, mehrere Simulationen zu unterstützen (ℜ4). Die gewählte Plug-In-Architektur bietet den Vorteil, unterschiedliche Anforderungen in unterschiedlichen Simulationsumgebungen bezüglich der Benutzerautorisierung (*UserAuthorizerInterface*) oder Simulationssteuerung (*SimulationControllerInterface*) zu unterstützten. Es müssen lediglich die entsprechenden Plug-Ins implementiert und in das JDQVisF eingebunden werden (siehe Kapitel 5.3.1).

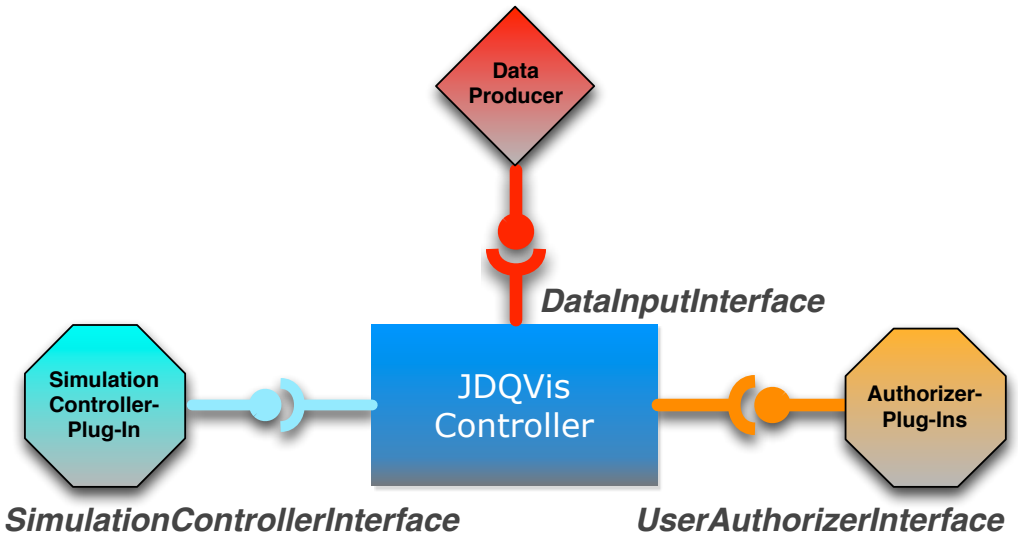


Abbildung 5-11: Kennzeichnung der Schnittstellen des JDQVisController. Die Symbole kennzeichnen den unterschiedlichen Charakter der Schnittstellen. Die Farben dienen zur Codierung der unterschiedlichen Schnittstellen

Die wesentlichen Aufgaben des JDQVisController sind die richtigen Plug-Ins für jeden *JDQVisClienten* zu laden, diese zu koordinieren und ihm zur Verfügung zu stellen.

Das Registrieren am JDQCF und das Empfangen der Datenqualitätswerte wird nicht in Plug-Ins ausgelagert. Diese Entscheidung begründet sich dadurch, dass das JDQCF mehrere Simulationen unterstützen werden kann und somit nur einmalige in das JDQVisF eingebunden werden muss. Damit auch externe Programme Rohdaten an das für das Visualisieren bereitstellen können, bietet der JDQVisController eine Schnittstelle für den Empfang von Datenwerten (*DataInputInterface*) an. Diese Schnittstelle wird in der technischen Umsetzung des JDQVisF in Kapitel 6.2.3 genauer beschrieben und wird an dieser Stelle nur für das allgemeine Verständnis erwähnt.

Die Unterstützung eines rollenbasierten Rechtesystems des Wissenschaftler wird durch die gewählte Plug-In-Architektur ebenfalls umgesetzt. Für jede Rolle, Simulation und jedes Anzeigegerät, können unterschiedliche Plug-Ins für die Verarbeitung geladen werden. Das bedeutet insbesondere auch, dass ein *JDQVisClient* mehrere Simulationen überwachen kann und jeweils andere Rechte an ihnen besitzt. Er bekommt also durch das Laden verschiedenen Benutzerautorisierungs-Plug-Ins unterschiedliche Rollen zugeteilt, die für die weiteren Visualisierungsschritte berücksichtigt werden. (ℜ9, ℜ10)

Für die Zuordnung der unterschiedlichen Daten, die bei mehreren parallel laufenden Simulationen und gleichzeitig angemeldeten *JDQVisClients* entstehen, werden Namespaces (siehe Kapitel 5.4.3.1) eingesetzt. Jeder *JDQVisClient* besitzt zu jeder Simulation einen eigenen Namespace in dem alle relevanten Daten abgelegt werden. Dieser wird vom *JDQVisController* für jeden angemeldeten *JDQVisClient* durch seine *UserId* einzigartig generiert. Eine Verwechslung von Wissenschaftlern wird dadurch ausgeschlossen und die Datenintegrität sichergestellt. Diese Architekturentscheidung erhöht somit die allgemeine Zugriffsicherheit. Nichtautorisierte Benutzer oder Wissenschaftler mit weniger Rechten an einer Simulation, können durch dieses *Sandbox-Prinzip* nicht auf Daten anderer angemeldeter Wissenschaftler zugreifen.

In den folgen Abschnitten werden die Erweiterungsschnittstellen für die Benutzerautorisierung und Simulationssteuerung vorgestellt. Zudem wird der Empfang der Rohdaten durch das JDQCF gezeigt.

5.4.1.1 Erweiterungen des JDQVisController

Der JDQVisController kann durch die verschiedenen Ansprüche an ihn, die Benutzerautorisierung und Simulationssteuerung nicht in einer abgeschlossenen Komponente umsetzen.
Aus diesem Grund bietet das JDQVisF über die beiden Schnittstellen *UserAuthorizerInterface* und *SimulationControllerInterface* die Möglichkeit simulationsabhängige Authori-

zer-Plug-Ins und SimulationController-Plug-Ins einzubinden. Diese werden bei Bedarf durch den *JDQVisController* geladen und ausgeführt.

Diese konzeptionelle Trennung der Schnittstellen zur Benutzerautorisierung und Steuerung der Simulation vereinfacht die unterschiedlichen Ansprüche der verschiedenen Benutzergruppen des JDQVisF zu realisieren. Auf der einen Seite können Wissenschaftler durch das Authorizer-Plug-In simulationsabhängig neue Mitarbeiter hinzufügen, ihre Rechte verändern oder komplett aus der Simulation entfernen, wodurch ein rollenbasiertes Rechtesystem für den Wissenschaftler leicht umzusetzen ist (ℜ9). Auf der anderen Seite bietet das SimulationController-Plug-In eine zentrale Stelle, die den Zugriff auf die Simulation durch definierte Regeln steuern kann (ℜ14).
Aus Sicht der Entwickler bietet diese Trennung den Vorteil, dass er jeweils eine definierte Schnittstelle für die Realisierung eines Authorizer- oder SimulationController-Plug-Ins zur Verfügung gestellt bekommt. Soll ein neues Plug-In eingebunden werden, muss lediglich die entsprechende Schnittstelle implementiert und im entsprechenden Namespace (siehe Kapitel 0) registriert werden. Eine direkte Manipulation des JDQVisF ist dadurch nicht erforderlich. Diese Architektur basiert durch die Trennung der Plug-Ins vom Rest des JDQVisF auf einer losen Kopplung, was ihre Einfachheit und Flexibilität erhöht, was wiederum ihre Portabilität und Wiederverwendbarkeit ermöglicht (ℜ18).

Die folgenden Unterkapitel beschreiben die Erweiterungen für die Benutzerautorisierung und Simulationssteuerung sowie die Anbindung an das JDQCF im Detail.

5.4.1.1.1 Benutzerautorisierung durch ein Authorizer-Plug-In

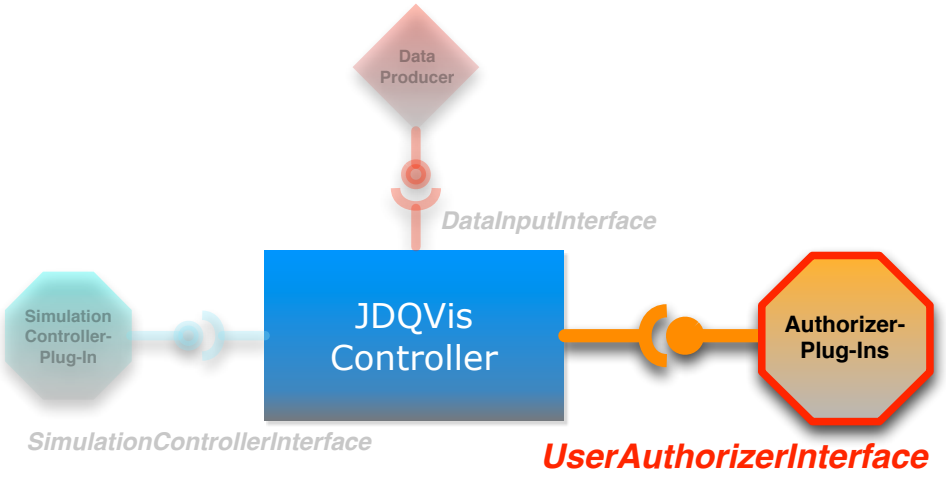


Abbildung 5-12: Authorizer-Plug-In des JDQVisController

Ein *Authorizer-Plug-In* ist eine Erweiterung des *JDQVisController* für die Autorisierung von *JDQVisClients* (ℜ7). Diese Komponente des JDQVisF bietet den *JDQVisClienten* eine zentrale Stelle für die Anmeldung zur Überwachung von Simulationen.

Ein *Authorizer-Plug-In* bekommt die Anmeldedaten, bestehend aus Benutzername und Passwort, als Eingabe und liefert eine *JDQVisUser*-Objekt als Ausgabe. Ein *JDQVisUser*-Objekt besteht aus einer *UserId* und einer *Role*. Eine *UserId* dient dem *JDQVisController* als Referenz auf den angemeldeten Wissenschaftler und kann aus einer beliebigen Zahlen- und Buchstabenkombination bestehen. Die *Role* spiegelt die Rolle des Wissenschaftlers in der Simulation. So kann vom *JDQVisController* und vom *VisualizationMediator* unterschieden werden, welche Rechte der Wissenschaftler an der Simulation besitzt. Das JDQVisF setzt durch diese Architektur eine simulationsabhängige Benutzerautorisierung um (ℜ7, ℜ9).

Ein Benutzerautorisierungs-Plug-In ist für genau eine Simulation verantwortlich. Das bedeutet, dass für unterschiedliche Simulationen, unterschiedliche Benutzerautorisierungs-Plug-Ins ausgeführt werden müssen. Indem der Entwickler verschiedene Plug-Ins implementiert, kann er für jede Simulation festlegen, welche Wissenschaftler sich für eine Simulation anmelden können und welche Rechte sie an dieser Simulation haben.

UserAuthorizationInterface

Der *JDQVisController* bietet über das *UserAuthorizationInterface* die Möglichkeit neue Benutzerautorisierungen in das JDQVisF einzubinden. Dazu muss ein Benutzerautorisierungs-Plug-In die *getUser*-Methode implementieren. Diese bekommt als Parameter den Benutzernamen und Passwort für die angeforderte Simulation. (ℜ16)

Eine technische Umsetzung eines *UserAuthorizationInterface* wird in Kapitel 6.2.1.1 gezeigt.

5.4.1.1.2 Simulationssteuerung durch ein SimulationController-Plug-In

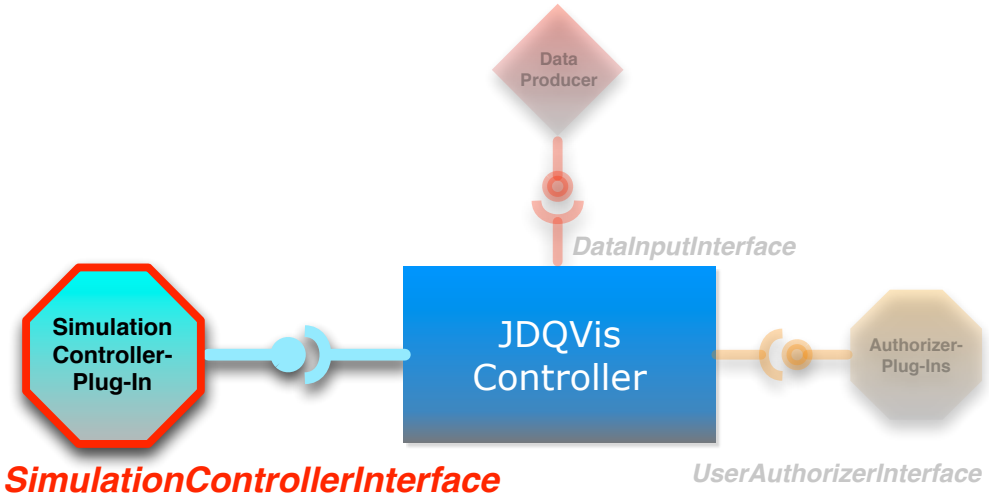


Abbildung 5-13: SimulationController-Plug-In des JDQVisController

Ein SimulationController-Plug-In ist eine Erweiterung des *JDQVisController* für die Weiterleitung von Steuerungsbefehlen an die Simulation. Es bietet dem Wissenschaftler die Möglichkeit in eine laufende Simulation einzugreifen. Durch diese Plug-In Architektur lassen sich simulationsabhängige, komplexe Steuerungen realisieren. Beim JDQVis-Controller angemeldete Wissenschaftler können nur in die Simulationen eingreifen für die sie registriert sind. Durch diese *Sandbox-Architektur* wird die Integrität der Simulationsdaten durch das JDQVisF gewährleistet (ℜ8).

Da der Fokus des JDQVisF auf dem Visualisieren von Datenqualität liegt, wird diese Komponente lediglich als eine Schnittstelle für das Weiterleiten der Benutzereingaben verwendet. Der JDQVisController ruft beim Empfang eines Steuerungsbefehls für die Simulation das für die entsprechende *SimulationId* registrierte Plug-In auf.

Aus Sicht der Simulationssteuerung bietet diese Lösung einen einfachen Weg für das Verarbeiten von Steuerungsbefehlen, da sie sich nur über eine Schnittstelle am JDQVisF registrieren muss.

Ein *SimulationController*-Plug-In bekommt eine UserId und den Steuerbefehl als Eingabe und gibt einen Verarbeitungsbericht in Textform als Rückgabe zurück. Die UserId dient dem SimulationController-Plug-In als Referenz auf den Wissenschaftler der den Steuerbefehl an das JDQVisF gesendet hat. Ist ein Wissenschaftler für eine SimulationId beim JDQVisController angemeldet, kann er diese mit Hilfe des Plug-Ins steuern. Wichtige Anwendungsfälle könnten das vorzeitige Abbrechen der Simulation oder das Auswechseln von Workflow-Komponenten bei schlechter Datenqualität sein. Durch das Übergeben der *UserId* kann das SimulationController-Plug-In entscheiden, ob der Wissenschaftler die nötigen Rechte für den übergebenen Steuerbefehl besitzt.

SimulationControllerInterface

Der *JDQVisController* bietet über das *SimulationControllerInterface* eine Schnittstelle für die Anbindung von SimulationController-Plug-Ins.

Die Umsetzung eines SimulationControllerInterface wird in Kapitel 6.2.1.1 gezeigt.

5.4.1.2 Empfang der Rohdaten für das JDQVisF

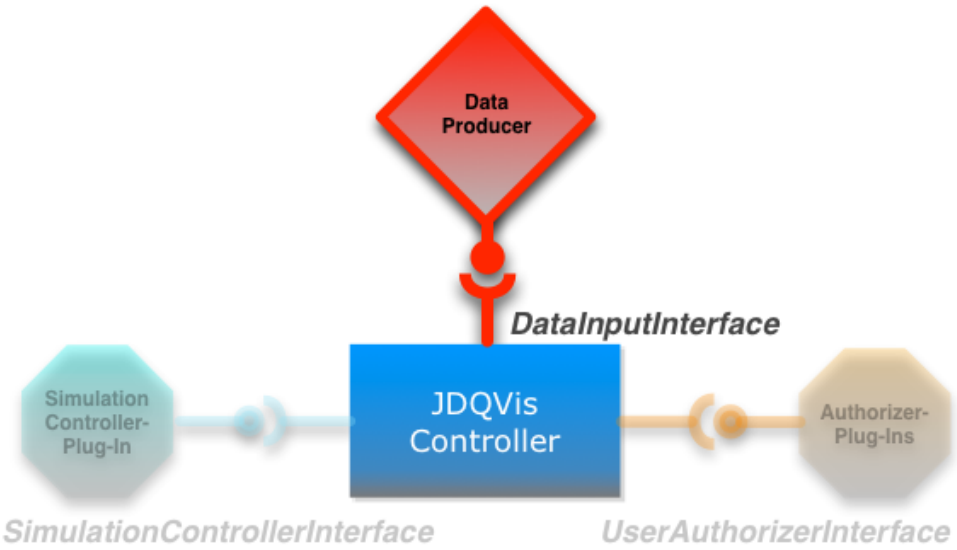


Abbildung 5-14: Empfang der Rohdaten für das JDQVisF

Um Datenqualitätsvisualisierungen erstellen zu können benötigt das JDQVisF Eingabedaten. Aus Gründen der Wiederverwendbarkeit des JDQVisF und Entkopplung vom JDQCF wird, im Gegensatz zu den oberen beiden Abschnitten, eine Architektur auf Basis von Namespaces (siehe Kapitel 5.4.3.1) vorgestellt.
Für jede Simulation gibt es einen bestimmten Namespace */ raw,* an dem die Rohdaten für die weitere Verarbeitung liegen. Das bedeutet, sollen neue Datenqualitätswerte durch das JDQVisF visualisiert werden, müssen sie in diesen Namespace abgelegt werden.

Diese Methode ermöglicht das Empfangen von Rohdaten, z.B. Datenqualitätswerten, ohne die direkte Anbindung an das JDQCF. Somit kann das JDQVisF auch Daten aus anderen Datenquellen (*DataProducer*) für die Eingabedaten verwenden.

Für das Überwachen der Namespaces besitzt der *JDQVisController* eine Hilfskomponente. Liegen neue Eingabedaten an, benachrichtig diese den *JDQVisController*. Diese Komponente trennt somit die Funktion Datenempfang von den übrigen Aufgaben des *JDQVisController* und führt dadurch zu einer besseren Struktur.
Der genaue Ablauf bei dem Empfang von Rohdaten wird in der technischen Umsetzung des JDQVisF in Kapitel 6.2.3 gezeigt.

Konzeptionelle Anbindung an das JDQCF (ℜ2)

Dieser Abschnitt beschreibt die konzeptionelle Anbindung des JDQVisF an das JDQCF. Für die genaue Beschreibung des JDQCF und dessen konzeptionellen Aufbau wird auf [1] verwiesen.

Um das JDQCF als Datenquelle nutzen zu können, muss das JDQVisF zwei Schritte realisieren. Im ersten Schritt muss es sich beim JDCQF über eine *Subscibe*-Nachricht für eine Simulation oder Metrik registrieren. Eine Metrik beschreibt dabei eine Datenqualitätsdimension. Das bedeutet, das JDQVisF muss sich für eine vollständige Überwachung, auf alle gewünschten Metriken registrieren. Im zweiten Schritt muss es für den Empfang von Datenqualitätswerten die *DataQualityReceiver*-Schnittstelle des *JDQCF* implementieren. Dieses enthält eine Methode, die für den Empfang der Datenqualitätswerte aufgerufen wird. Die berechneten Datenqualitätswerte werden durch das JDQCF an den *QoDReceiver* in serialisierter Form übermittelt. Dazu werden die *CalculationResult-Container* von der *DispatchingApi* in XML serialisiert und in den SOAP-Header der Response-Nachricht gepackt.

Abbildung 5-15 zeigt die Anbindung des JDQVisF an das JDQCF mit allen nötigen Schritten.

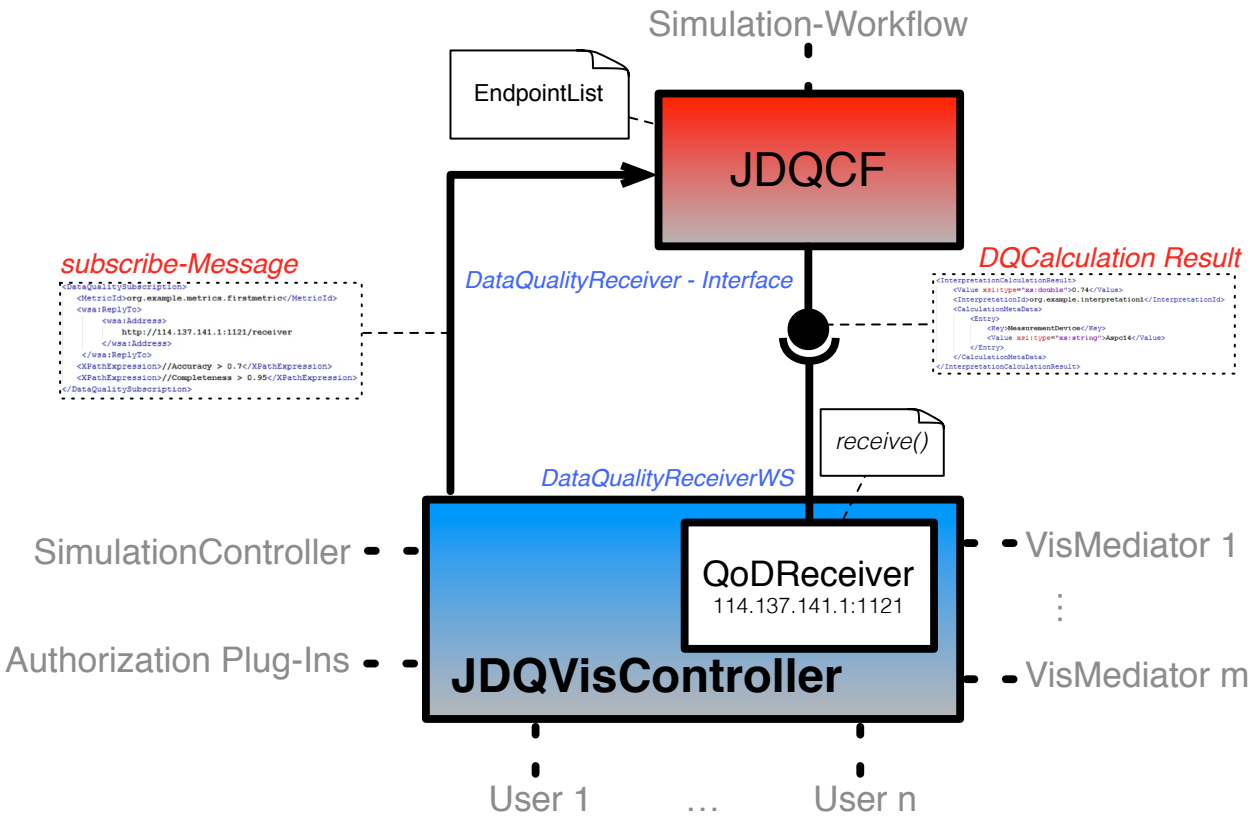


Abbildung 5-15: Anbindung des JDQVisController an das JDQCF

Da das *JDQVisF* das *JDQCF* als eine mögliche Datenquelle verwendet, werden durch den *QoDReceiver* des *JDQVisController* alle empfangenen Datenqualitätswerte in den entsprechenden Namespace in XML serialisiert.

5.4.2 Architektur des VisualizationMediator

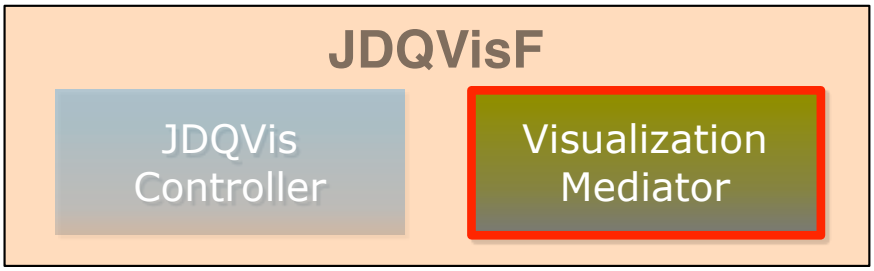


Abbildung 5-16: Komponenten des JDQVisF mit hervorgehobenen VisualizationMediator

Ein *VisualizationMediator* ist die zweite Hauptkomponente des JDQVisF und realisiert das Visualisieren der Datenqualitätswerte und die Steuerung der Visualisierungspipeline. Er besitzt eine ähnliche Architektur wie der *JDQVisController*. Der *VisualizationMediator* generiert für jeden, beim *JDQVisController* angemeldeten *JDQVisClient* und *SimulationId,* unterschiedliche Visualisierungen. Da das JDQVisF mehrere Simulationen und eine variierende Anzahl von Wissenschaftlern mit unterschiedlichen Rechten unterstützt, wäre der *VisualizationMediator* als eine abgeschlossene Visualisierungskomponente zu unflexibel, komplex, fehleranfällig und wartungsintensiv. Aus diesem Grund wird eine Plug-In-Architektur für den *VisualizationMediator* gewählt. Der *VisualizationMediator* ist eine Steuerungskomponente, welche die Generierung der Visualisierungen durch die Steuerung der Visualisierungspipeline übernimmt (ℜ5). Er wird vom JDQVisF benötigt, um alle Stufen zu koordinieren. Da ein hohes Maß an Flexibilität erreicht werden soll, werden die Schritte der Visualisierungspipeline durch verschiedene Plug-Ins realisiert.
Abbildung 5-17 zeigt die Schnittstellen des *VisualizationMediator*.

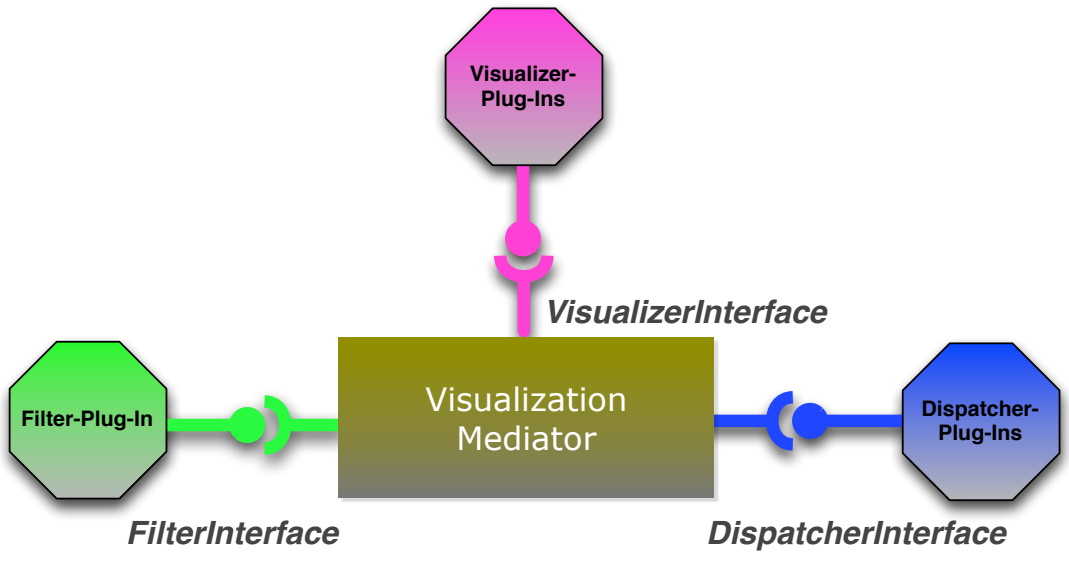


Abbildung 5-17: Schnittstellen des VisualizationMediator

Diese Plug-In Architektur bietet die Möglichkeit, die unterschiedlichen Eigenschaften der Anzeigegeräte in der Visualisierungspipeline zu berücksichtigen. Sie erlaubt eine flexible Zusammenstellung der einzusetzenden Plug-Ins und kann dadurch die unterschiedlichen Rechte der Wissenschaftler unterstützen. Zudem ermöglicht diese Architektur die unterschiedlichen Eigenschaften von verschiedenen Anzeigegeräten bei der Generierung der Visualisierungen zu berücksichtigen. So können beispielsweise für ein Tablet-Computer mit viel Rechenleistung und einem Smartphone mit wenig Rechenleistung unterschiedliche Visualisierungen für die gleichen Daten und Rollen generiert werden (ℜ13). Der *VisualizationMediator* lädt jeweils ein passendes *Filter*-, *Visualizer*- und *Dispatcher-Plug-In* und bietet somit ein Gerüst für die unterschiedlichen Stufen der Visualisierungspipeline (ℜ12).

Wie bei dem *JDQVisController* hat diese Plug-In-Architektur wichtige Eigenschaften, mit deren Hilfe die Anforderungen an das JDQVisF umgesetzt werden können. So besitzen die einzelnen Plug-Ins eine lose Kopplung untereinander. Das bedeutet, einzelne Plug-Ins können leicht ausgetauscht oder an neue Anforderungen angepasst werden. Dadurch wird die Wartung und Pflege vereinfacht und die Flexibilität und Wiederverwendbarkeit des ganzen JDQVisF oder einzelner Plug-Ins erhöht. Zudem ermöglicht sie eine flexible Zusammenstellung von *Filter*-, *Visualizer*- und *Dispatcher*-Plug-Ins. So kann beispielsweise ein Filter-Plug-In unabhängig von der späteren Visualisierung und einzig unter Berücksichtigung der Rolle und Simulation realisiert werden. Zudem ermöglicht diese Architektur die einzelnen Schritte der Visualisierungspipeline durch unterschiedliche Spezialisten zu realisieren. So kann beispielsweise ein Wissenschaftler durch das Implementieren eines Filter-Plug-Ins für jede Rolle festlegen, welche Daten für die Visualisierung freigegeben werden. Der Visualisierer hingegen muss sich keine Gedanken um die Herkunft der Daten machen und kann sich auf seinen Visualisierungsalgorithmus konzentrieren. Ein Dispatcher-Plug-In bekommt die generierten Visualisierungen als Eingabedaten und versendet sie an die gewünschte Adresse mit dem implementierten Protokoll und ist dabei komplett von der restlichen Visualisierungspipeline entkoppelt.

Zusätzlich hat diese Plug-In-Architektur den Vorteil, dass Plug-Ins zur Laufzeit eingebunden oder ausgetauscht werden können ohne das gesamte JDQVisF anzupassen und somit andere Simulationen und JDQVisClienten zu beeinträchtigen.

Um die gewünschten Visualisierungen zu generieren wählt der *VisualizationMediator* mit Hilfe eines Plug-In-Registers jeweils ein *Filter*-, ein *Visualizer*- und ein *Dispatcher-Plug-In* aus. Das Register enthält neben der Rolle des Wissenschaftlers und der *SimulationId* eine *DeviceId,* welche für die Auswahl der *Visualizer*- und *Dispatcher-Plug-Ins* mitentscheidend ist (ℜ10). Der genaue Aufbau dieses Registers wird in Kapitel 6.3.4 gezeigt.

Abbildung 5-18 beschreibt die Aufteilung der Visualisierungspipeline auf *Filter*- *Visualizer*- und *Dispatcher-Plug-Ins* mit Kennzeichnung des Datenflusses. Die Schritte *Map*-

ping und *Rendering* werden zu einem *Visualizer*-Plug-In zusammengefasst. Dies hat den Vorteil, dass die Implementierung von geräteabhängigen Visualisierungen vereinfacht wird. Für ein Anzeigegerät mit viel Rechenleistung kann beispielsweise nur das Mapping innerhalb des Visualizer-Plug-In realisiert werden und das Rendering lokal auf dem Gerät. Bei einem leistungsschwachen Gerät wiederum kann die Rechenleistung des Servers ausgenutzt werden, um komplexe Bilder zu generieren.

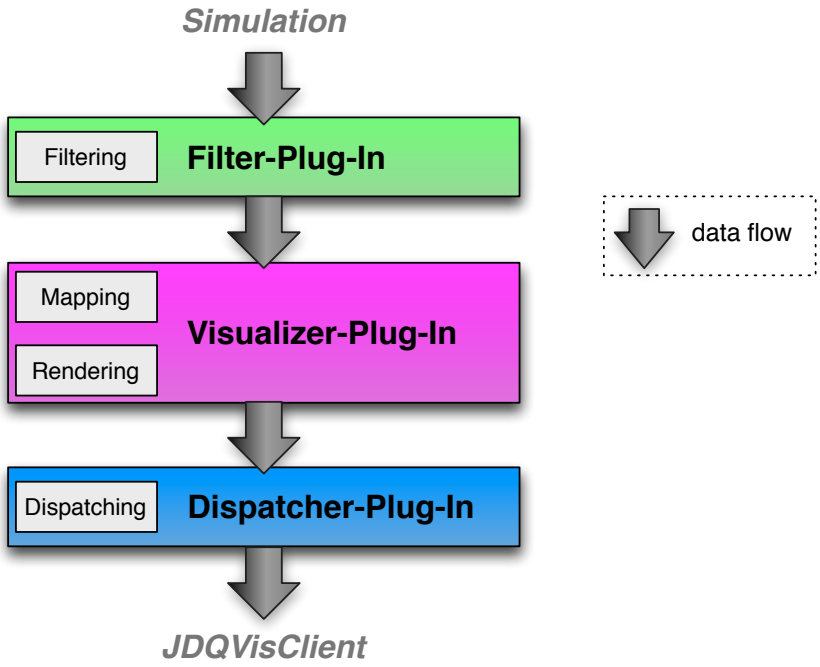


Abbildung 5-18: Aufteilung der Visualisierungspipeline auf Filter- Visualizer- und Dispatcher-Plug-In mit Kennzeichnung des Datenflusses

Die Einbindung der Plug-Ins für die Visualisierung in den *VisualizationMediator* und nicht an den *JDQVisController* begründet sich durch die in Kapitel 0 gezeigte Trennung des Aufgabenbereichs des JDQVisF. Er bietet zusätzlich den Visualisierern einen zentralen Punkt um ihre Visualizer-Plug-In einzubinden und erleichtert dadurch die Wartung und Pflege.

Für die Gewährleistung der Integrität der Daten, die bei den einzelnen Schritten entstehen, werden wie beim *JDQVisController* Namespaces (siehe Kapitel 5.4.3.2) verwendet. Das dadurch entstehende *Sandbox-Prinzip* verhindert die Veränderung der Daten durch die anderen geladenen Plug-Ins.

In den folgenden Abschnitten werden die Erweiterungen der Visualisierungspipeline des *VisualizationMediator* genauer beschrieben.

5.4.2.1 Erweiterungen des VisualizationMediators

Der VisualizationMediator kann seine Visualisierungspipeline wegen der oben gezeigten Anforderungen nicht in einer abgeschlossenen Komponente umsetzen. Aus diesem Grund bietet das JDQVisF den Entwicklern drei Schnittstellen an um ihre *Filter*-, *Visualizer*- und *Dispatcher-Plug-Ins* dem *VisualizationMediator* zur Verfügung zu stellen. Dazu gehören das *FilterInterface* für Filter-Plug-Ins, das *VisualizerInterface* für Visualizer-Plug-Ins und das *DispatcherInterface* für Dispatcher-Plug-Ins.

Dieses Kapitel beschreibt die konzeptionelle Architektur der Erweiterungen, ihre Funktionen und ihre Schnittstellen.

Die konzeptionelle Aufteilung der Visualisierungspipeline in diese drei Plug-Ins begründet sich hauptsächlich durch die daraus entstehende Flexibilität. Der *VisualizationMediator* kann zu jeder Rolle-Simulation-Anzeigegerät-Beziehung die passenden Plug-Ins zu einer Visualisierungspipeline kombinieren. Er nutzt die Wiederverwendbarkeit der einzelnen Plug-Ins und vermeidet dadurch redundante Implementierungen, was die Wartung und Pflege der einzelnen Plug-Ins vereinfacht.
In der ersten Stufe können die Wissenschaftler simulationsabhängige und rollenbasierte Filter entwickeln, ohne die restlichen, zum Teil komplizierten, Schritte der Visualisierungspipeline anpassen zu müssen. In der zweiten Stufe kann ein Visualisierer, der ein Visualizer-Plug-In entwickelt, die Eigenschaften eines Anzeigegerätes optimal ausnutzen. Er kann komplizierte Visualisierungsalgorithmen einbinden, ohne sich Gedanken um die Datenherkunft oder das Versenden der Visualisierungen machen zu müssen. In der letzten Stufe versendet ein Dispatcher-Plug-In Eingabedaten an eine bestimme Adresse, ohne das deren Herkunft berücksichtigt werden muss.

Um den jeweiligen Spezialisten einen einfache Rahmen für das Einbinden ihrer Plug-Ins bereitzustellen, bietet das JDQVisF drei Schnittstellen mit jeweils nur einer Methode an. Diese Architektur teilt dadurch die Plug-Ins nach ihrer jeweiligen Aufgabe und ermöglicht das Laden der richtigen Plug-Ins durch den *VisualizationMediator*. Möchte beispielsweise ein Visualisierer ein neues Visualizer-Plug-In für ein spezielles Anzeigegerät und eine bereits existierende Rolle und Simulation in das JDQVisF einbinden, so muss er nur eine Schnittstelle implementieren und das Plug-In im entsprechenden Namespace (siehe Kapitel 5.4.3.2) registrieren. Das gleiche gilt für die Filter- und Dispatcher-Plug-Ins.

Visualisierungsspezifikationen

Da alle Plug-Ins eine in sich abgeschlossene Komponente bilden, benötigten sie eine Möglichkeit für die Kommunikation mit dem *VisualizationMediator,* um beispielsweise den Ein- und Ausgabepfad der Daten zu bekommen. Zusätzlich benötigen Filter- und Visualizer-Plug-Ins eine Schnittstelle zu dem Wissenschaftler, über die dieser in die Visualisierungspipeline eingreifen kann. Aus diesem Grund werden für die drei Plug-Ins so genannte Visualisierungsspezifikationen angelegt. Sie enthalten beispielsweise den

Ein- und Ausgabepfad der Daten und im Fall von *Filter*- und *VisualizerSpecifications* ein Element für die Benutzeranfragen. Der Plug-In-Entwickler muss sich also nicht um die Beschaffung der Ein- und Ausgabepfade kümmern, was die Implementierung vereinfacht. Der *JDQVisController* passt die entsprechenden Spezifikationen bei der Anmeldung eines *JDQVisClienten* oder bei dem Empfang eines Steuerungsbefehls an. Eine persistente Auslagerung dieser Spezifikationen, zum Beispiel in eine XML-Datei, bietet im Gegensatz zu einer einfachen Parameterübergabe durch den *VisualizationMediator* an die Verarbeitungsmethoden der Plug-Ins, mehrere Vorteile. So können Wissenschaftler, beispielsweise zu Simulationsbeginn, festlegen, welche Daten während des Simulationsdurchganges visualisiert werden und auf welche Art. Sie müssen dazu einmalig ihre *Filter*- und *VisualizerSpecifications* mit Hilfe des *JDQVisController* anpassen. Diese werden dann bei jeder Anmeldung des Wissenschaftlers unabhängig vom Anzeigegerät berücksichtigt. Somit erhält er selbst bei Anzeigegeräte ohne Interaktionsmöglichkeiten genau die Visualisierungen, die er zuvor an einem anderen Anzeigegerät konfiguriert hat.

Im Folgenden werden die einzelnen Plug-Ins, die entsprechende Schnittstelle des JDQVisF und die Einbindung in das JDQVisF gezeigt.

5.4.2.1.1 Datenaufbereitung durch ein Filter-Plug-In

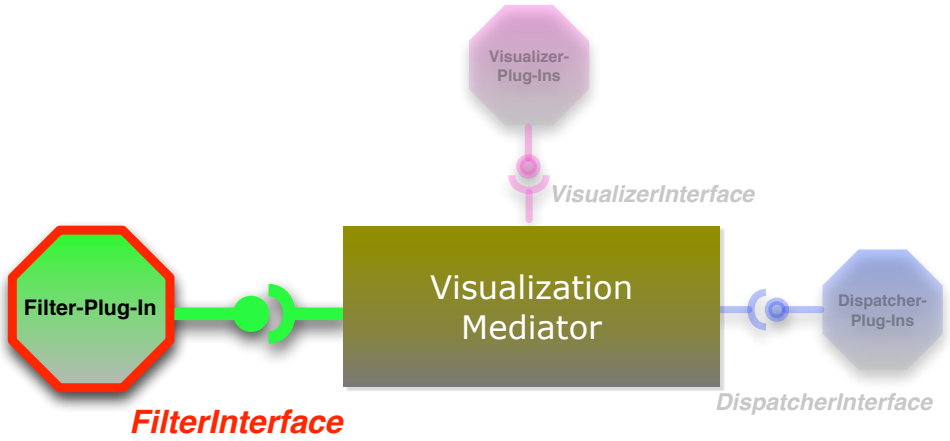


Abbildung 5-19: Filter-Plug-In des VisualizationMediator

Ein Filter-Plug-In ist eine Erweiterung für die erste Stufe der Visualisierungspipeline. Es bekommt die Rohdaten als Eingabe und liefert aufbereitete Daten als Ausgabe. Dabei ist ein Filter-Plug-In für eine SimulationId und eine bestimmte Rolle in dieser Simulation verantwortlich. Das bedeutet, dass für jede Rolle in einer Simulation unterschiedliche Filteroperationen ausgeführt werden können. Ein Wissenschaftler kann durch die Implementierung von unterschiedlichen Filter-Plug-Ins für alle Wissenschaftler festlegen, welche Daten die einzelnen Rollen sehen können und welche ihnen verborgen bleiben. Beispielsweise benötigt der Wissenschaftler dem die Simulation gehört alle Daten, während für Wissenschaftler mit weniger Rechten, nur die berechneten Datenqualitätswerte

sichtbar sind. Zudem lassen sich durch dadurch die Interaktionsmöglichkeiten des JDQVisClient an dem Filter-Plug-In kontrollieren. So kann das Filter-Plug-In anhand der Rolle unterscheiden, ob es eine Steuerungsanfrage bearbeitet oder nicht.

Das Filter-Plug-In bietet den Wissenschaftler eine einfache Möglichkeit die Rechte von Rollen anzupassen. Durch die gewählte Plug-In-Architektur können leicht weitere Rollen zu einer Simulation hinzugefügt, bearbeitet oder einzelne Rollen entfernt werden.

Da ein Filter entgegen des Namens zur allgemeinen Rohdatenaufbereitung verwendet wird, ist er insbesondere dafür verantwortlich, die originalen Simulationsdaten bei Bedarf zu laden und sie für die weiteren Verarbeitung zur Verfügung zu stellen. Dabei können die Simulationsdaten außerhalb des JDQVisF, beispielsweise bei einer *Data as a Service* (DaaS) liegen. Ein DaaS beschreibt eine Datenbank in der Cloud und wird in [31] und [40] genauer beschrieben. Es liegt in der Verantwortung des Plug-In Entwickler die Simulationsdaten zu dereferenzieren und zu laden. Die Adresse der Daten erhält ein Filter beispielsweise aus den Rohdaten des JDQCF. (ℜ3)

FilterSpecification

Ein Filter-Plug-In kann über eine so genannte *FilterSpecification* gesteuert werden. In ihr findet der Programmierer alle für ihn wichtigen Angaben. So wird in ihr neben der Rolle und der SimulationId der Ein- und Ausgabepfad der Daten und ein FilterRequest angegeben.

In Kapitel 6.3.5.1 wird die Umsetzung einer *FilterSpecification* durch ein XML-Dokument gezeigt.

FilterInterface

Der *VisualizationMediator* bietet über das *FilterInterface* die Möglichkeit, neue Filteralgorithmen in das JDQVisF einzubinden. Dazu muss ein Filter-Plug-In die *filterData*-Methode implementieren. Sie bekommt den Pfad zu einer passenden *FilterSpecification* als Parameter.

5.4.2.1.2 Visualisierung durch ein Visualizer-Plug-In

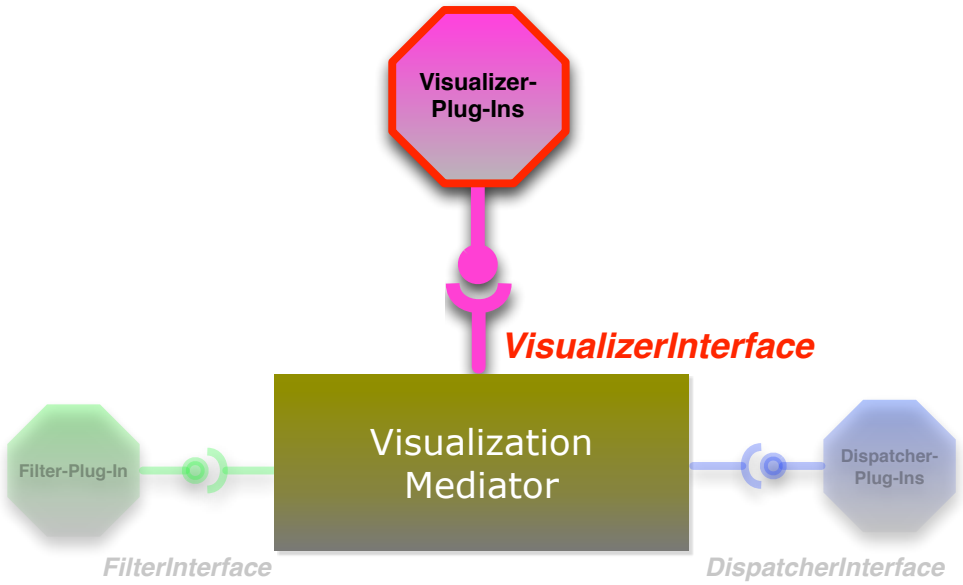


Abbildung 5-20: Visualizer-Plug-In des VisualizationMediator

Ein Visualizer-Plug-In fasst die zweite und dritte Stufe der Visualisierungspipeline zusammen und realisiert dadurch die Schritte *Mapping* und *Rendering*. Diese Zusammenlegung begründet sich durch die Unterstützung von unterschiedlichen Anzeigegeräten durch das JDQVisF. Sie ermöglicht es dem Visualisierer, optimal auf das Anzeigegerät und der Simulation abgestimmte Visualisierungen zu generieren. Dadurch kann, beispielsweise bei leistungsstarken Anzeigegeräten, das JDQVisF nur das Mapping durch ein *Visualizer-Plug-In* übernehmen. Das Rendering, also die tatsächliche Bildgenerierung, wird lokal auf dem Gerät ausgeführt und kann die individuellen Stärken des Gerätes vollständig ausnutzen. Bei leistungsschwachen Geräten kann der Visualisierer hingegen die Rechenleistung des Servers ausnutzen. So kann er komplexe Visualisierungen erstellen, welche auf Geräteseite nur noch angezeigt werden müssen.
Zudem ermöglicht diese Plug-In-Architektur dem VisualizationMediator eine einfache Zusammenstellung seiner Visualisierungspipeline, indem er für jede Simulation-Rolle-Anzeigegerät-Beziehung, das jeweils passende *Visualizer-Plug-In* auswählt (ℜ13). Durch die lose Kopplung der einzelnen Plug-Ins untereinander und insbesondere auch innerhalb der Visualisierungspipeline, können leicht neue Anzeigegeräte zur Visualisierung der Datenqualität hinzugefügt werden können. Ein Visualisierer muss lediglich ein neues *Visualizer-Plug-In* in das JDQVisF hinzufügen (siehe Kapitel 6.3.4.2) und einen entsprechenden *Client* auf dem Anzeigegerät installieren.

Aus Sicht der Plug-In-Entwickler bietet diese Architektur mehrere Vorteile. Sie können unabhängig der restlichen Komponenten des JDQVisF ihre Visualisierungen in dessen Visualisierungspipeline einbinden. Das vereinfacht insbesondere das Anpassen und Hinzufügen von Visualisierungsalgorithmen und erleichtert somit die Wartung. Die klare Trennung der Schritte *Filtern* und *Visualisieren* entkoppelt zusätzlich die Visualisierer von den Wissenschaftlern, die das Aufbereiten der Rohdaten übernehmen können. So

sind die Visualizer-Plug-In unabhängig von der Datenquelle realisierbar, was die Flexibilität erhöht.

Ein Visualizer-Plug-In bekommt die aufbereiteten Daten als Eingabe und liefert die generierten Visualisierungen als Ausgabe. Als Visualisierung werden sowohl einfache Bilder, zum Beispiel für einen Internetbrowser, wie auch komplexe Geometriemodelle für leistungsstärkere Ausgabegeräte unterstützt. Für die Umsetzung wählt der *VisualizationMediator* für das aktuelle Anzeigegeräte, das jeweils passende *Visualizer-Plug-Ins* aus.

Wie beim Filtern, muss der *VisualizationMediator* auch beim Visualisieren zwischen den einzelnen Simulationen und deren unterschiedlichen Rollen unterscheiden. Somit ist es möglich, dass unterschiedliche Rollen unterschiedliche Visualisierungen als Ausgabe erhalten (ℜ10). Beispielsweise kann ein Visualizer-Plug-In für die Rolle mit allen Rechten an der Simulation, zusätzlich zu den Datenqualitätswerten, auch die Simulationsdaten visualisieren. Rollen mit weniger Rechten bekommen hingegen nur die visualisierten Datenqualitätswerte angezeigt. Zudem lassen sich durch die Berücksichtigung der Rolle die Interaktionsmöglichkeiten zwischen JDQVisClient und *Visualizer-Plug-In* kontrollieren. So kann ein *Visualizer-Plug-In* für jede Rolle definieren, ob es die Steuerungsanfrage bearbeitet oder nicht.

Um unterschiedliche Visualisierungen für unterschiedliche Endgeräte zu erhalten, lädt der VisualizationMediator das passende Plug-In. Dieses findet er anhand eines Plug-In-Registers (siehe Kapitel 6.1) in der alle *Visualizer-Plug-Ins* mit einer *SimulationId*, *Role* und *DeviceId* registriert sind. Diese *DeviceId* kennzeichnen die Anzeigegeräte, die das Plug-In unterstützt. Jedes Visualizer-Plug-In, das vom *VisualizationMediator* geladen werden soll, muss mindestens eine *DeviceId* unterstützen.

VisualizerSpecification

Ein Visualizer-Plug-In wird durch eine *VisualizerSpecification* gesteuert. In ihr findet der Programmierer alle für ihn wichtigen Angaben. So wird in ihr neben der Rolle und der SimulationId auch der Ein- und Ausgabepfad der Daten und ein *VisualizationRequest* angegeben.
Für jedes SimulationId-User-Paar gibt es genaue eine *VisualizerSpecification*. Eine VisualizerSpecification auf Basis einer SimulationId-Role ist nicht möglich, da diese mögliche Benutzerinteraktionen beschreiben und somit für jeden *JDQVisClienten* einzigartig sein müssen, um andere *JDQVisClients* mit der selben Rolle nicht zu beeinflussen.

In Kapitel 6.3.5.2 wird die Umsetzung einer *VisualizerSpecification* durch ein XML-Dokument gezeigt.

VisualizerInterface

Das JDQVisF bietet über das *VisualizerInterface* die Möglichkeit neue Visualisierungsalgorithmen für den VisualizationMediator einzubinden. Dazu muss ein Visualizer-Plug-

In die visualize*Data*-Methode implementieren. Diese bekommt als Parameter den Pfad zu einer passenden *VisualizerSpecification*.

Der VisualizationMediator hat keinen Einfluss auf die tatsächliche Visualisierung der Daten und kann somit nicht garantieren, dass gute Visualisierungen (siehe Kapitel 3.4) generiert werden oder nicht. Die Verantwortung der Berücksichtigung der Anforderungen an eine Visualisierung liegt bei den Entwicklern der Visualizer-Plug-Ins. Für die Unterstützung wird zusätzlich ein Verweis auf diese Arbeit im Interface gegeben (ℜ11).

5.4.2.1.3 Verteilung der Visualisierungen durch ein Dispatcher-Plug-In

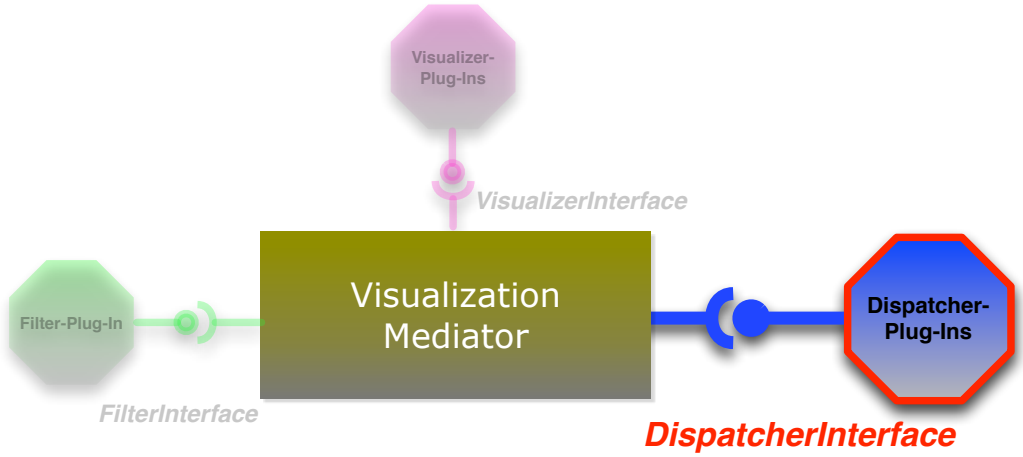


Abbildung 5-21: Dispatcher-Plug-In des VisualizationMediator

Ein *Dispatcher-Plug-In* ist eine Erweiterung für das Versenden der generierten Visualisierungen. Es bekommt dabei die Visualisierungen als Eingabe und verteilt diese an die angegebene Adresse. Wie in den zuvor vorgestellten Erweiterungen, wird auch hier die Plug-In Variante aus Gründen der Flexibilität gewählt. Diese bietet den Vorteil, dass leicht neue Versandarten eingebunden oder bestehende angepasst werden können.
Zudem wird das Versenden der Visualisierungen von den restlichen Komponenten des JDQVisF entkoppelt, was zu einer Vereinfachung der Wartung führt (ℜ16). Aus Sicht der Implementierer bietet diese lose Kopplung zusätzlich den Vorteil, dass sie nur eine Schnittstelle implementieren müssen um ihre Versandarten in das JDQVisF einzubinden. Anpassungen am JDQVisF sind nicht nötig (ℜ17, ℜ18).

Diese Plug-In Architektur ermöglicht dem JDQVisF unterschiedliche Versandprotokolle (z.B. SOAP) und -Arten (z.B. SOAP-HTTP-Binding) für das Verteilen der Daten zu unterstützen. Das Verwenden von *Dispatcher*-Plug-Ins bietet den Vorteil, dass der VisualizationMediator für jede Rolle und Anzeigegerät ein anderes Plug-In verwenden kann. Es lassen sich Versandarten einbinden, die genau auf ein Anzeigegerät und den Ansprüchen der Wissenschaftler abgestimmt sind. Beispielsweise können die Daten über Soap-Nachrichten an ein leistungsstarkes Anzeigegerät versendet werden. Eine andere

Möglichkeit ist die Daten auf einem Server abzulegen, bei dem sie über eine URL abgefragt werden können (ℜ15).

Neben dem reinen Verteilen der Daten bietet ein *Dispatcher-Plug-In* die Möglichkeit, Daten an eine Cache-Komponente zu versenden. Damit lassen sich auch Visualisierungen älterer Simulationsdaten generieren oder zeitliche Verläufe der Simulation abbilden.

DispatcherSpecification

Ein *Dispatcher-Plug-In* wird durch eine *DispatcherSpecification* gesteuert. In ihr findet der Programmierer alle wichtigen Angaben. So wird in ihr neben der Rolle und der SimulationId, der Ein- und Ausgabepfad der Daten, das Eingabeformat und einen optionalen Pfad für die Ablage in den Cache des JDQVisF angegeben.

In Kapitel 6.3.5.3 wird die Umsetzung einer *DispatcherSpecification* als XML-Dokument gezeigt.

DispatcherInterface

Der *VisualizationMediator* bietet über das *DispatcherInterface* die Möglichkeit neue Verteilungsalgorithmen in das JDQVisF einzubinden. Dazu muss ein Dispatcher-Plug-In die dispatch*Data*-Methode implementieren. Diese bekommt als Parameter den Pfad zu einer passenden *DispatcherSpecification*.

5.4.3 Namespaces des JDQVisF

Dieses Kapitel beschreibt die verschiedenen Namensräume (Namespaces) die das JDQVisF unterstützt. Sie werden von der vorgestellten Architektur benötigt, um für jede Simulation-Wissenschaftler-Anzeigegerät-Beziehung die richtigen Daten zu finden und zu verarbeiten. Sie realisieren dadurch das Sandbox-Prinzip des JDQVisF, das die Zugriffsicherheit und die Datenintegrität gewährleisten.

Für die Beschreibung der Namespaces werden *SimulationIds* und *UserIds* in einfachen Hochkomma geschrieben. Sie werden im konkreten Fall durch die echten Identifiatonen ersetzt. Tabelle 1 gibt eine Erklärung der Symbole, die in den folgenden Unterkapiteln verwendetet werden.

Tabelle 1: Erklärung der Symbole in den Beschreibungen der Namespaces des JDQVisF.

Symbol	Bedeutung
'simulationId'	Repräsentiert einen Platzhalter für eine konkrete *SimulationId.*
'userId'	Repräsentiert einen Platzhalter für eine konkrete *UserId.*
*	Beliebige Anzahl von Elementen inkl. 0.
+	Beliebige Anzahl von Elementen größer als 0.
!	Genau ein Element.
|	Genau ein Element der Auswahl.

5.4.3.1 Ressourcen und Datenhaltung

Dieses Kapitel beschreibt die verschiedenen Namespaces für die Ressourcen und Datenhaltung. Sie dienen zur Identifikation der Daten, die bei den verschiedenen Stufen der Visualisierungspipeline entstehen.

Abbildung 5-22 zeigt einen Überblick der Namespaces für die Ressourcen. Die Farben beziehen sich auf die der einzelnen Komponenten des JDQVisF und kennzeichnen das entsprechenden Verwendungsgebiet.

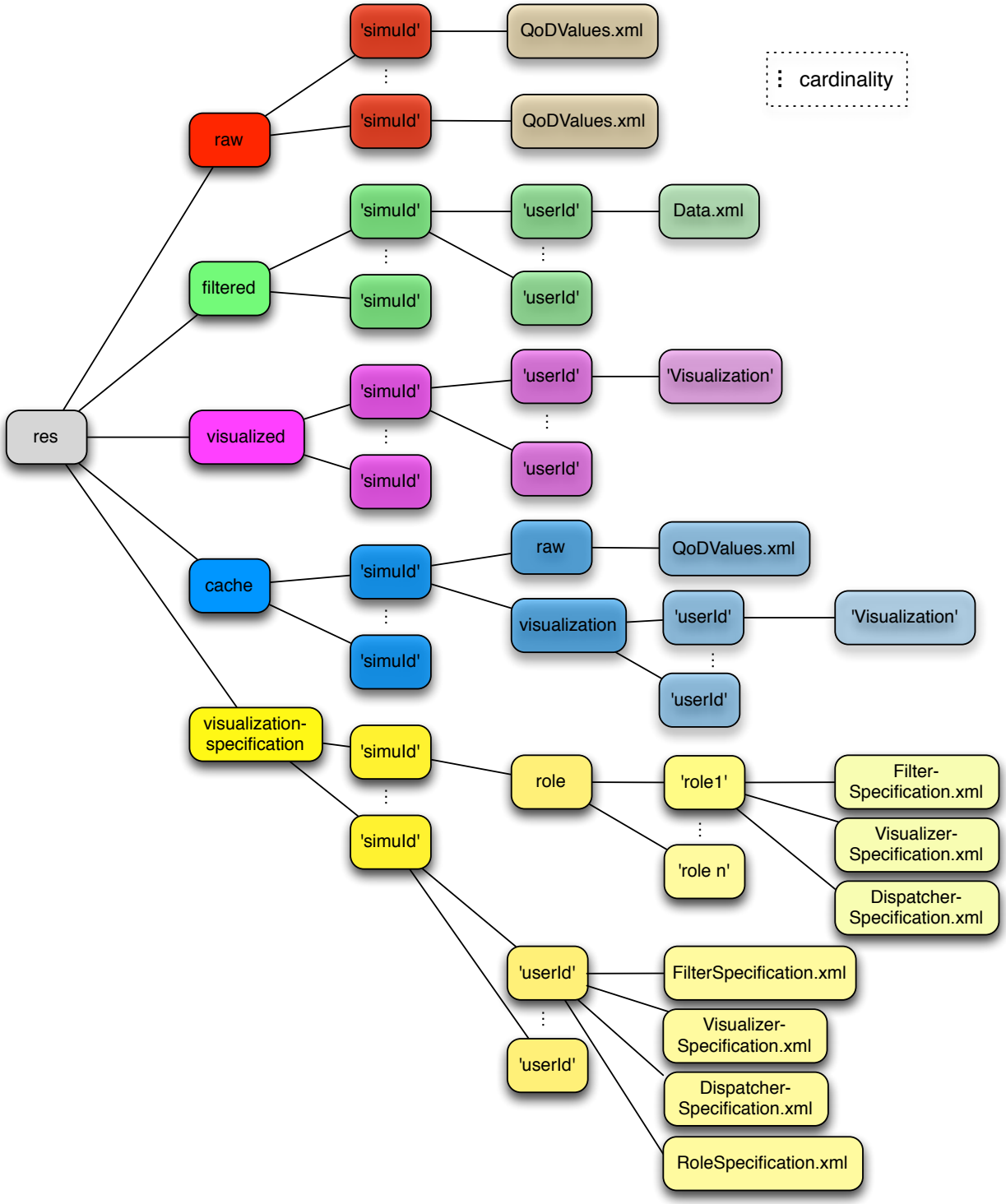


Abbildung 5-22: Übersicht über die Namespace-Hierarchie der Ressourcen und Datenhaltung des JDQVisF mit entsprechenden Kardinalitäten

/ res

Dieses Element enthält alle Ressourcen, die für die Visualisierung von Datenqualität und deren Steuerung benötigt werden. Es enthält genau ein *cache*-, ein *filtered*-, ein *raw*-, ein *visualizationspecification*- und ein *visualized*-Element.

/ res / raw

Dieses Element enthält alle, dem JDQVisF bekannten, Simulationen und dient zum Empfang der Rohdaten aus dem Simulation-Workflow, beispielsweise vom JDQCF. Es dient als Ausgangspunkt aller weiteren Schritte der Visualisierungspipeline.

*/ res / raw / 'simulationId' **

Dieses Element enthält die Rohdaten einer Simulation. Es besitzt nur ein weiteres Kindelement */QoDValues*! in dem die Rohdaten gespeichert sind. Rohdaten sind im Zusammenhang von Simulation-Workflows, die Datenqualitätswerte und eventuell zusätzlichen Simulationsdaten, welche visualisiert werden sollen. Sie dienen den *Filter-Plug-Ins* als Eingabedaten.

/ res / filtered

Dieses Element enthält alle, dem JDQVisF bekannten, Simulationen und dient als Container für die aufbereiteten Datenqualitäts- und Simulationsdaten. Es beinhaltet damit alle Daten, die in nach der ersten Stufe der Visualisierungspipeline, dem Filtern, entstehen.

*/ res / filtered / 'simulationId'**

Dieses Element enthält für die jeweilige Simulation alle am JDQVisF registrierten UserIds als Kindelemente. Somit können die unterschiedlichen Anforderungen und Rollen der einzelnen Wissenschaftler unterschieden werden.

/ res / filtered / 'simulationId' / ,userId'

Dieses Element repräsentiert die Simulation-Wissenschaftler-Beziehung und besitzt genau ein Kindelement */Data.* In diesem werden die aufbereiteten Datenqualitäts- und Simulationsdaten für den Wissenschaftler und die Simulation gespeichert. Damit ist es möglich, die unterschiedlichen Simulation-Wissenschaftler-Beziehungen umzusetzen. Je nach Simulation und Rolle werden unterschiedliche Daten generiert und in dem entsprechenden Namespace abgelegt. Die dort liegenden Daten dienen als Eingabe der *Visualizer-Plug-Ins*.

/ res / visualized

Dieses Element enthält alle, dem JDQVisF bekannten, Simulationen und dient als Container für die visualisierten Datenqualitäts- und Simulationsdaten. Es beinhaltet alle Daten, die in der zweiten Stufe der Visualisierungspipeline, dem Visualisieren, entstehen.

*/ res / visualized / 'simulationId'**

Dieses Element enthält für die jeweilige Simulation alle registrierten *UserIds* als Kindelemente. Somit können die unterschiedlichen Anforderungen und Rollen der einzelnen Wissenschaftler unterschieden werden.

/ res / visualized / ’simulationId’ / ,userId’

Dieses Element repräsentiert die Simulation-Wissenschaftler-Beziehung und besitzt die generierten Visualisierungen als Kindelemente. Damit ist es möglich die unterschiedlichen Simulation-Wissenschaftler-Beziehungen umzusetzen. Je nach Simulation und Rolle werden unterschiedliche Visulisierungen generiert und in dem entsprechenden Namespace abgelegt. Die dort liegenden Daten dienen als Eingabe der *Dispatcher-Plug-Ins*.

/ res / visualizationspecification

Dieses Element enthält alle, dem JDQVisF bekannten, Simulationen und dient als Container für alle *Visualisierungsspezifikationen*.

*/ res / visualizationspecification / ’simulationId’ **

Dieses Element dient zur Kennzeichnung der angeforderten Simulation. Es enthält für jeden registrierten Benutzer die jeweiligen Visualisierungsspezifikationen. Dazu enthält es zwei Kindelemente. /user enthält alle Spezifikationen von aktiven Benutzern. /role enthält dagegen für jede Rolle in der Simulation Muster-Visualisierungsspezifikationen.

*/ res / visualizationspecification / ’simulationId’ / user / ’userId’**

Dieses Element enthält für den Wissenschaftler mit entsprechender UserId die verschiedenen Visualisierungsspezifikationen für das Filtern (/*FilterSpecification*), Visualisieren (/*VisualizerSpecification*) und Versenden (/*DispatcherSpecification*) der Daten. Außerdem wird über das Element */RoleSpecification* die Rolle des Wissenschaftlers in der aktuellen Simulation festgelegt.

/ res / visualizationspecification / ’simulationId’ / user / ’userId’ / FilterSpecification!

Dieses Element enthält alle Anweisungen des JDQVisClient für das geladene *Filter-Plug-In*

/ res / visualizationspecification / ’simulationId’ / user / ’userId’ / VisualizerSpecification!

Dieses Element enthält alle Anweisungen für das geladene *Visualizer-Plug-In.*

/ res / visualizationspecification / ’simulationId’ / user / ’userId’ / DispatcherSpecification!

Dieses Element enthält alle Anweisungen für das geladene *Dispatcher-Plug-In.*

/ res / cache

Dieses Element enthält alle, dem JDQVisF bekannten, Simulationen und dient zum Speichern der Simulationsdaten und generierten Visualisierungen für spätere Auswertungen oder falls der Simulation-Workflow zum Anfragezeitpunkt keine neuen Daten sendet.

*/ res / cache / ’simulationId’**

Dieses Element enthält alle verarbeiteten Daten einer Simulation. Das bedeutet, dass alle Rohdaten und generierten Visualisierungen für jeden Benutzer (UserId)

gespeichert werden. Dazu besitzt es die beiden Kindelement */raw* und */visualizations,* wobei letzteres noch beliebig viele Kindelemente für die unterschiedlichen *UserId*s (*/'userId'*) besitzt.

5.4.3.2 Funktionale Erweiterungen und deren gemeinsames Register

Dieses Kapitel beschreibt die Namespaces für die Plug-Ins, welche die Benutzerautorisierung, die Steuerung der Simulation, das Filtern der Rohdaten, das Visualisieren der aufbereiteten Daten und das Versenden der generierten Visualisierungen realisieren.

Abbildung 5-23 zeigt den Aufbau der Namespaces für die Plug-In-Verwaltung des JDQVisF.

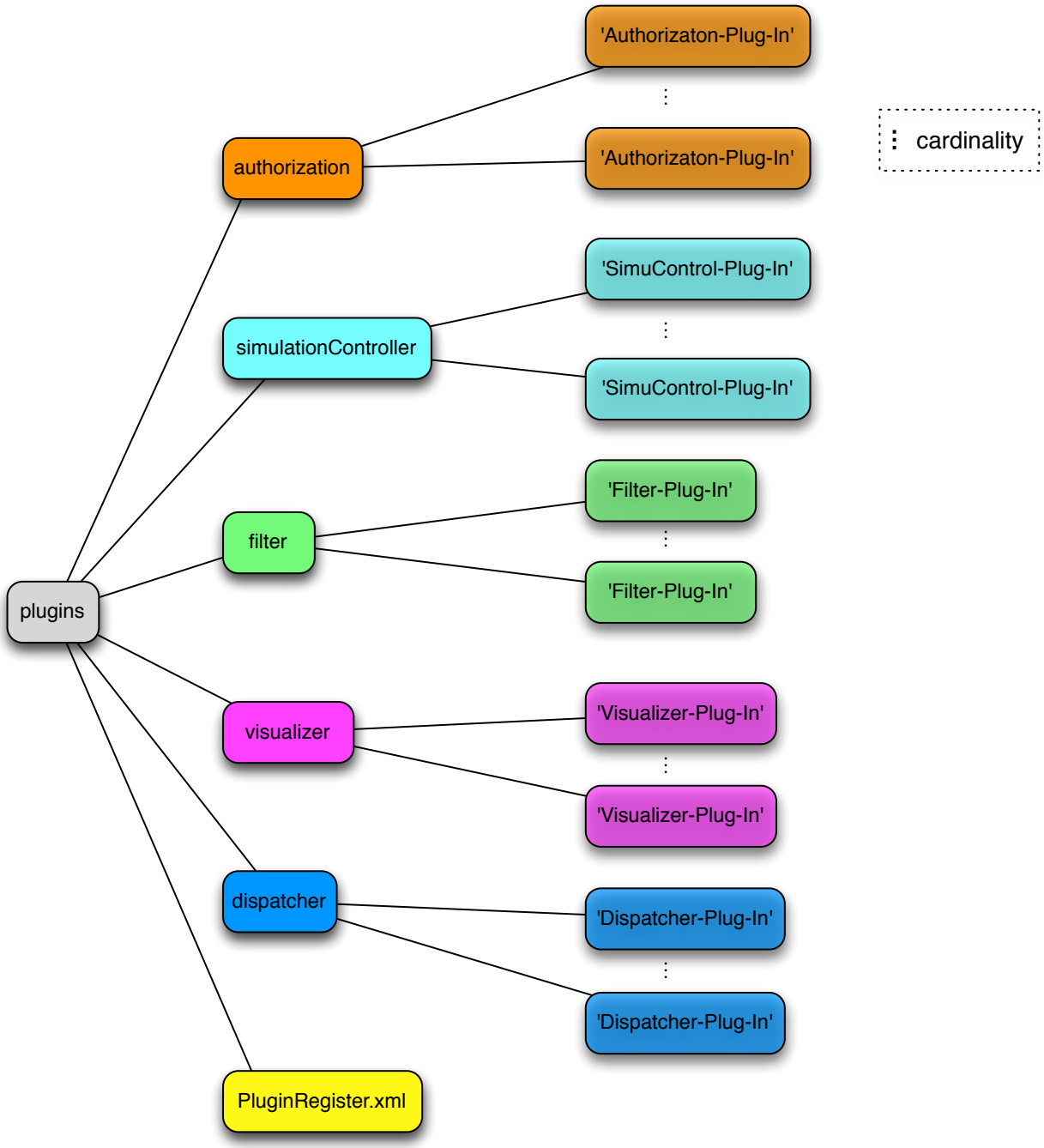


Abbildung 5-23: Übersicht über die Namespace-Hierarchie der Plug-Ins für das JDQVisF mit entsprechenden Kardinalitäten

/ plugins

Dieses Element ist das Wurzelelement aller Plug-Ins, die vom JDQVisF geladen werden sollen. Es enthält jeweils Kindelemente für Authorizer-Plug-Ins, Dispatcher-Plug-Ins, Filter-Plug-Ins und Visualizer-Plug-Ins.

/ plugins / PluginRegister

Dieses Element dient zur Registrierung der Plug-Ins. Hier müssen alle Erweiterungen, sei es ein Dispatcher-, Filter-, Autorisierungs- oder Visualizer-Plug-Ins, die vom JDQVisF verwendet werden sollen, eingetragen werden. Der genau Prozess der Plug-In-Registrierung wird in Kapitel 6.1 gezeigt.

/ plugins / (authorizer | simulationController | dispatcher | filter | visualizer)

Dieses Element enthält beliebig viele Kindelemente / *'Plug-In'* und ist der Ausgangspunkt aller Plug-Ins. Jedes Plug-In das eine Funktionalität realisiert, kann eines dieses Element als Elternknoten besitzen. Beispielsweise hat ein konkretes Filter-Plug-In: */ plugin / filter* als Elternknoten.

5.4.3.3 Schnittstellen

Um das JDQVisF durch Plug-Ins funktional zu erweitern, bietet es verschieden Schnittstellen an. Dieses Kapitel beschreibt die Namespaces der Schnittstellen (Interfaces) des JDQVisF. Wenn ein Entwickler ein neues Plug-In implementieren möchte, dass vom JDQVisF geladen werden kann, muss er eines der folgenden Interfaces implementieren.

Abbildung 5-24 zeigt die Namespacehierarchie der Interfaces die das JDQVisF anbietet.

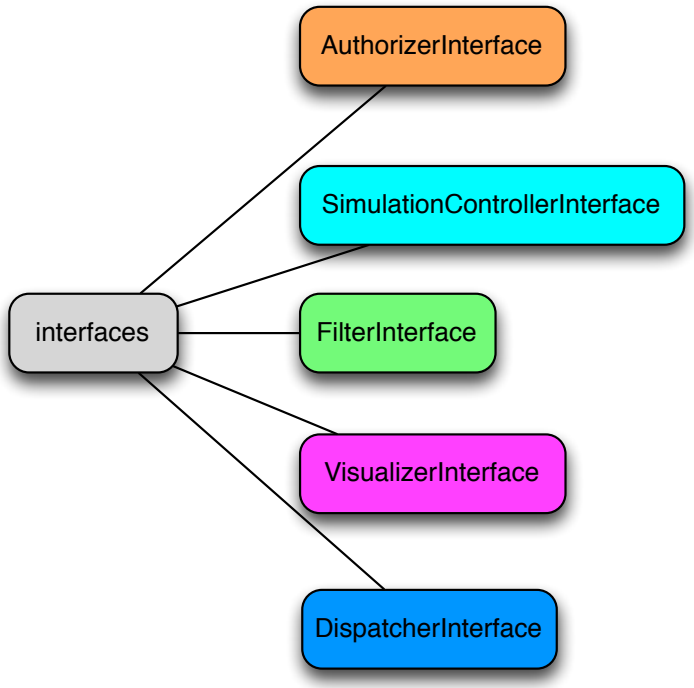


Abbildung 5-24: Übersicht über die Namespace-Hierarchie aller Interfaces des JDQVisF

/ interfaces

Dieses Element ist das Wurzelelement aller Interfaces die vom JDQVisF bereit gestellt werden. Es besitzt vier Kindelemente: /UserAuthorizerInterface, /SimulationControllerInterface /DispatcherInterface, /FilterInterface und /VisualizerInterface. Hier findet ein Plug-In-Entwickler das entsprechende Interface.

5.5 Beschreibung des Visualisierungsprozess

Dieser Abschnitt beschreibt den konzeptionellen Prozess von der Anmeldung beim *JDQVisController* bis zum Versenden der Visualisierungen durch ein Dispatcher-Plug-In.

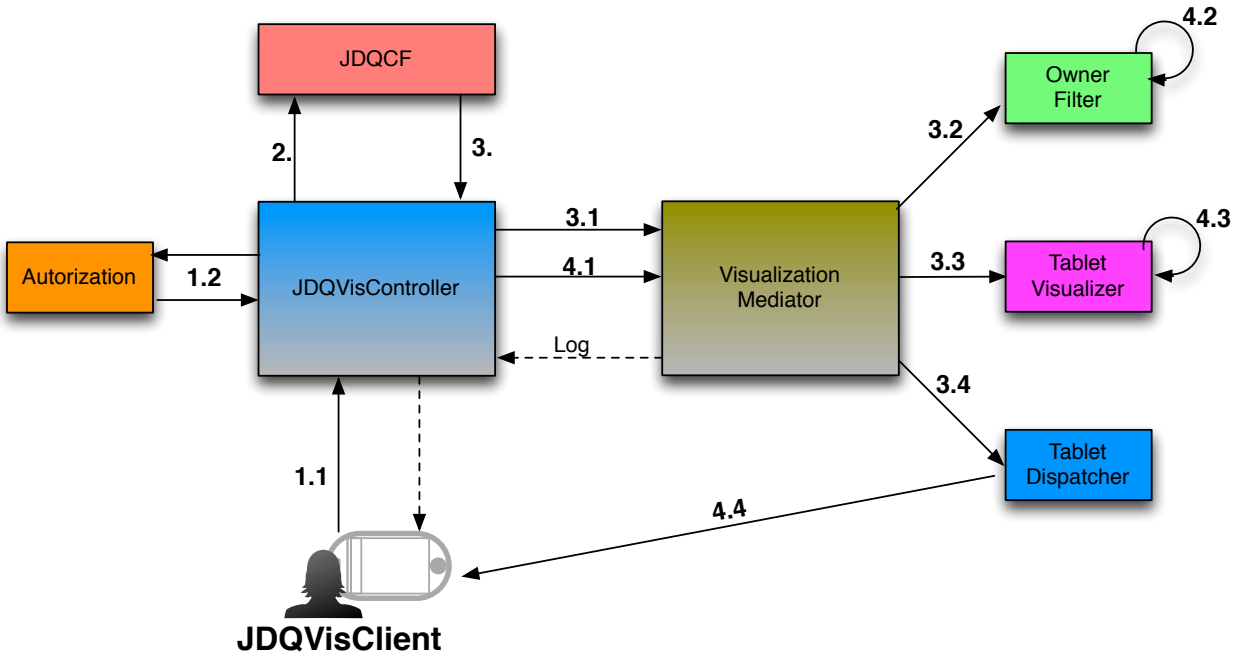


Abbildung 5-25: Konzeptioneller Ablauf vom Registrieren des JDQVisClients bis zum Versenden der Daten

Schritt 1 – Anmeldung beim JDQVisController

Möchte ein Wissenschaftler die Datenqualität einer Simulation mit Hilfe des *JDQVisF* überwachen, so muss er sich zunächst beim *JDQVisController* registrieren. Dazu benötigt der *JDQVisController* neben der *SimulationId* und *DeviceId* einen Benutzernamen und ein Passwort um sicher zu stellen, dass nur am System registrierte und somit berechtigte Personen Zugriff auf die angeforderten Simulationsdaten erhalten. Die Anmeldedaten werden über das *UserAuthorizerInterface* mit Hilfe eines entsprechenden Authorizer-Plug-Ins validiert.

Ist ein Benutzer erfolgreich am JDQVisController mit seiner *UserId* und *Role* registriert, werden daraus in Kombination mit der *SimulationId* die im vorangegangenen Kapitel vorgestellten Namensräume (*Namespaces*) generiert. Diese dienen im weiteren Visualisierungsprozess zur Identifikation der benötigten Daten.

Schritt 2 – Registrierung beim JDQCF

Erhält der *JDQVisController* bei der Anmeldung eines Benutzers eine inaktive *SimulationId*, so wird diese aktiv. Aktiv heißt, dass der *JDQVisController* sich beim JDQCF für generierte Datenqualitätswerte dieser Simulation als Endpunkt neu registriert. Für ein bereits aktive SimulationId ist dieser Schritt nicht notwendig, da das JDQVisF für diese Simulation beim JDQCF durch einer früheren Anmeldung registriert wurde.

Schritt 3 – Empfang der Datenqualitätswerten und Start der Visualisierung

Empfängt der *JDQVisController* über seinen *QoDReceiver* neue Datenqualitätswerte, so wird daraus die *SimulationId* gelesen. Im Anschluss wird für alle, für diese *SimulationId* angemeldeten, *JDQVisClienten* der Visualisierungsprozess gestartet. Dazu wird jeweils ein passender *VisualizationMediator* instanziiert. Dieser lädt dabei, die für die SimulationId-Role-DeviceId-Beziehung passenden, Plug-Ins und stellt sie für die Generierung der Visualisierung bereit.

Schritt 4 – Visualisierung der Datenqualitätswerte und Verteilung

Ist der *VisualizationMediator* instanziiert, startet der *JDQVisController* für jeden angemeldeten *JDQVisClient* dessen Visualisierungspipeline durch Aufruf der *visualize*-Methode des *VisualizatonMediator*. Diese führt nacheinander die Methoden der einzelnen Plug-Ins aus. Zuerst wird die *filterData*-Methode des geladenen Filter-Plug-Ins, anschließend die *visualizeData*-Methode des geladenen Visualizer-Plug-Ins und zum Schluss die *dispatchData*-Methode des geladenen Dispatcher-Plug-Ins ausgeführt.
Tritt bei den einzelnen Schritten ein Fehler auf, so wird der *JDQVisClient* über den *JDQVisController* darüber informiert. Ein typischer Fehler könnte dabei ein unauffindbares Plug-In sein. Tritt dieser Fehler auf, ist höchstwahrscheinlich ein fehlender oder fehlerhafter Eintrag im Plug-In-Register verantwortlich.

Wurden die Daten korrekt visualisiert, kann der *JDQVisClient* diese über das entsprechende Protokoll empfangen und anzeigen.

6 Technische Umsetzung des Java Data Quality Visualization Framework

Dieses Kapitel beschreibt die technische Umsetzung des JDQVisF. Dazu werden zunächst die Komponenten *JDQVisController* und *VisualizationMediator* beschrieben. Anschließend werden für jedes Plug-In Beispielimplementierungen gezeigt.

Das JDQVisF wird in der aktuellen Java-Version 1.7 implementiert und mit Hilfe von JAX-WS als WebService deklariert.

6.1 Aufbau und Struktur der PlugInRegister.xml

In den vorangegangenen Kapiteln wurde argumentiert warum das JDQVisF, genauer gesagt, seine beiden Hauptkomponenten *JDQVisController* und *VisualizationMediator,* durch Plug-In-Architekturen umgesetzt werden.

Nachdem ein Entwickler ein neues Plug-In für das JDQVisF implementiert hat, muss es für die Verwendung beim JDQVisF registriert werden. Aus diesem Grund besitzt das es ein Plug-In-Register (*PlugInRegister.xml*). Es wird als XML Datei realisiert, da so die Entwickler auf einfache Weise ihre Plug-Ins einbinden können.

Die *PlugInRegister.x*ml enthält eine Liste von Beschreibungen aller verwendeter Plug-Ins. Da es sich um eine einzelne Datei unter einem festen Verzeichnis *(/ plugins / PlugInRegister.xml*) handelt, können Entwickler ihre Plug-Ins einfach am JDQVisF registrieren. Das JDQVisF kann durch dieses Verfahren zur Laufzeit um neue Plug-Ins erweitert werden oder bestehende ausgetauscht und angepasst werden.

Die genauen Einträge innerhalb der *PlugInRegister.xml* werden in den Unterkapiteln der jeweiligen Komponenten gezeigt.

6.2 JDQVisController Klasse

Die Klasse *JDQVisController* ist der Ausgangspunkt für jede Interaktion mit dem JDQVisF und wird mit Hilfe von JAX-WS Annotationen als WebService ausgeschrieben (siehe Abbildung 6-1).

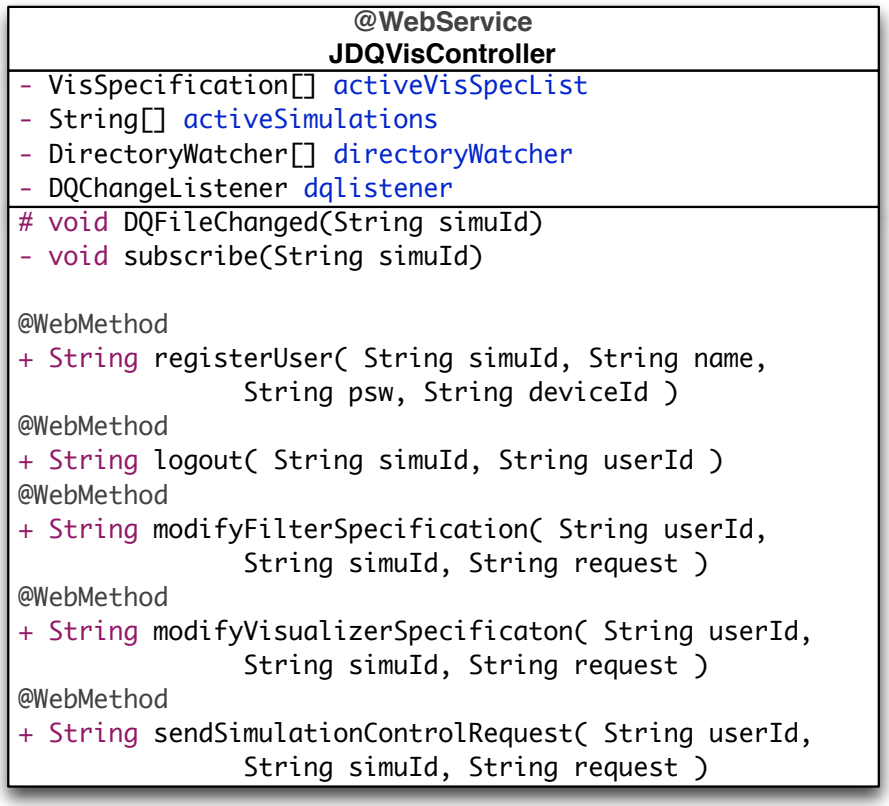


Abbildung 6-1: Klassendiagramm des JDQVisController

Der *JDQVisController* bietet dem *JDQVisClient* fünf Methoden für die Interaktion. Diese werden in den folgenden Abschnitten genauer beschrieben.

6.2.1 Benutzerregistrierung über die registerUser – Methode

Die *registerUser*-Methode ist der Einstiegspunkt für jeden *JDQVisClient*. Sie erwartet als Parameter die *SimulationId*, die Anmeldedaten für das Authorizer-Plug-In und die Kennzeichnung des gewünschten Anzeigegerätes über die *DeviceId*. Wurde der Benutzer erfolgreich durch das Authorizer-Plug-In überprüft, gibt diese Methode eine *UserId* zurück. Diese dient als Referenz und wird für alle weiteren Interaktionen mit dem *JDQVisController* benötigt.

6.2.1.1 AuthorizationInterface.jar

Das *AuthorizationInterface.jar* beschreibt die Schnittstelle des *JDQVisController* für das Einbinden der Authorizer-Plug-Ins. Es enthält lediglich ein Paket (*authorizationinterface*) mit einem Interface (*DQAuthorization*) darin. Dieses Interface bietet die Methode *authorize* an. Sie erwartet einen Benutzername und ein Passwort als Parameter.

Möchte der Entwickler ein Authorizer-Plug-In realisieren, so muss er das *DQAuthorizationInterface* in seiner *Library* einbinden und das Interface *DQAuthorization* implementieren.

Das User*AuthorizerInterface* ist unter dem in Kapitel 5.4.3.3 gezeigten Namespace */ interfaces / UserAuthorizerInterface.jar* des JDQVisF zu finden.

Damit der JDQVisController ein *Authorizer-Plug-In* verwenden kann, muss es unter der URL zu finden sein, die in dem in Kapitel 5.4.3.2 vorgestellten Namespace */ plugins / PlugInRegister.xml* registriert ist. Die URL kann auf eine beliebige Adresse zeigen. So ist es möglich auch externe Autorisierungsservices, die nicht in das JDQVisF eingebunden werden sollen, zu nutzen. Für eine leichtere Wartung, bietet das JDQVisF einen speziellen Namespace für alle Autorisierungs-Plug-Ins unter */ plugins / authorizer* an.

6.2.1.2 Registrierung eines Authorizer-Plug-In

Um ein Authorizer-Plug-In zu registrieren ist ein Eintrag in das PlugInRegister notwendig. Listing 2 zeigt den strukturellen Aufbau einer Registrierung im PlugInRegister.

```
<plugin type='dqauthorization'>
    <simulationId> ... </simulationId>+
    <url> ... </url>
    <classname> ... </classname>
</plugin>
```

Listing 2: Struktur der Registrierung eines Authorizer-Plug-Ins im PlugInRegister.xml

/ plugin
: Repräsentiert ein JDQVisF-Plug-In. Das *type*-Attribut *dqauthorization* identifiziert es als Autorisierungs-Plug-In.

/ plugin / simulationId+
: Beinhaltet jeweils die SimulationId für die dieses Plug-In geladen werden soll.

/ plugin / url
: Beinhaltet die URL an der das Plug-In zu finden ist. Das JDQVisF bietet dazu definierte Namespaces (siehe Kapitel 5.4.3.2) an. Prinzipiell kann die URL jedoch beliebig sein. So können auch externe Autorisierungsservices in das JDQVisF eingebunden werden. Dabei ist zu beachten, dass es sich um ein .jar-Archive handeln muss.

/ plugin / classname

Dieses Element identifiziert die Klasse die das *UserAuthorizerInterface* implementiert. Sie dient als Einstiegspunkt aller Authorizer-Plug-Ins und wird vom *JDQVis-Controller* bei der Benutzeranmeldung aufgerufen.

6.2.1.3 Ablauf der Benutzerregistrierung

Der JDQVisClient ruft die *registerUser*-Methode mit den Parametern: *SimulationId*, Benutzername, Passwort und *DeviceId* auf. Mit Hilfe des *GoF-Factory-Patterns* [41] wird daraufhin ein passender *DQAuthorizer* instanziiert. Dazu wählt die *AuthorizerFactory* Klasse mit Hilfe der *SimulationId* aus dem *PlugInRegister.xml* ein passendes Autorisierungs-Plug-In aus. Wird kein passendes gefunden, wird von der *AuthorizerFactory* Klasse nichts zurück gegeben und eine Fehlermeldung informiert den *JDQVisClient*.
Ist ein passendes Authorizer-Plug-In gefunden, wird eine neue Instanz der Klasse zurückgegeben, die innerhalb des Plug-Ins das *UserAuthorizerInterface* implementiert.
Anschließend ruft der JDQVisController die *authorize*-Methode des Authorizer-Plug-Ins auf und übergibt den Benutzername und das Passwort. Sind die Anmeldedaten korrekt, wird ein *JDQVisUser* Objekt an den *JDQVisController* zurückgegeben. Dieser leitet die darin enthaltende *UserId* an den *JDQVisClient* weiter. Andernfalls wird ein leeren String zurückgegeben.

Ist der Benutzer erfolgreich für eine *SimulationId* angemeldet, generiert der *JDQVisController* die entsprechenden Namespaces. Anschließend wird eine neue *VisSpecification* (Abbildung 6-2) aus den Anmeldedaten erstellt und zur *activeVisSpecList* des *JDQVis-Controller* hinzugefügt.

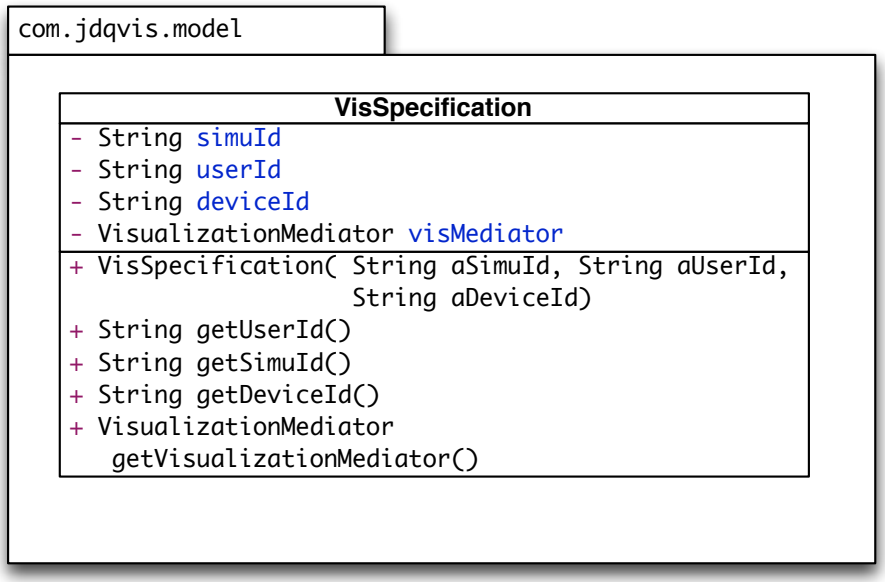


Abbildung 6-2: Klassendiagramm der VisSpecification-Klasse im Package com.jdqvis.model

Abbildung 6-3 fasst den Ablauf in einem Sequenzdiagramm zusammen.

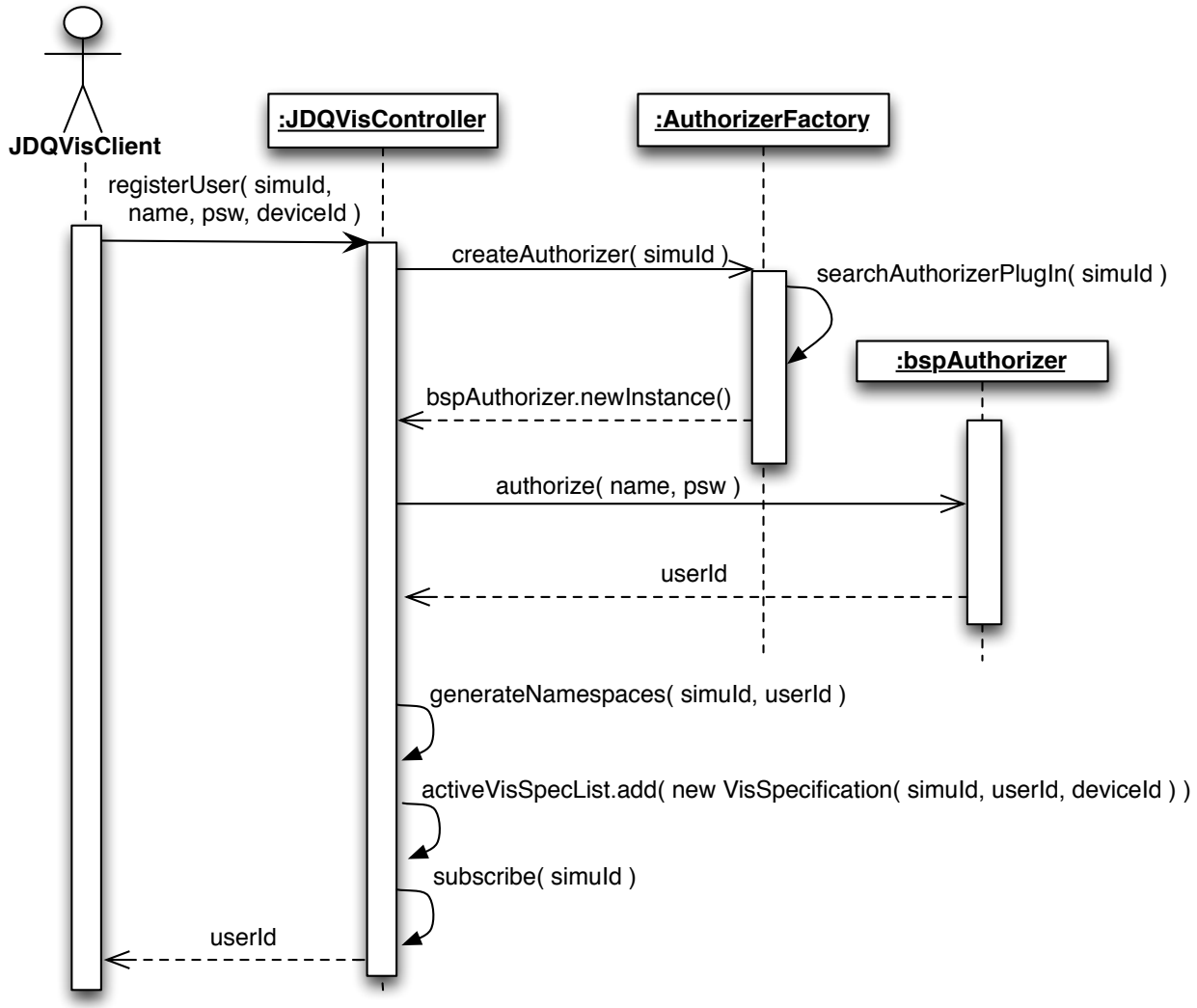


Abbildung 6-3: Sequenzdiagramm für die Benutzerregistrierung

6.2.2 Anbindung an das JDQCF über die subscribe-Methode

Das JDQCF bietet zwei Möglichkeiten für das Übermitteln von Ergebnisse von Datenqualitätsberechnungen an. Zum einen kann in einem Task ein externer Empfänger (hier der *QoDReceiver* des JDQVisF) angegeben werden. Zum anderen können Ergebnisse subskribiert werden. Das bedeutet, dass für laufenden Datenqualitätsberechnungen weitere externe Empfänger eingetragen werden.

Der JDQVisController realisiert einen solchen Subskribierungsauftrag in seiner *subscribe*-Methode. Diese wird aufgerufen, sobald sich ein neuer JDQVisClient ange-

meldet hat. Sie überprüft, ob die angeforderte *SimulationId* bereits durch einen anderen *JDQVisClient* angefordert wurde. Falls nein, sendet der *JDQVisController* eine subscribe-Nachricht über SOAP/HTTP-Binding an das JDQCF und gibt die Adresse seines QoDReceiver als so genannten *Endpoint* an. Ein *Endpoint* repräsentiert eine IP Adresse, an die das JDQCF die berechneten Datenqualitätswerte versendet.

Zu beachten ist, dass zum Zeitpunkt der Entstehung dieser Arbeit, das JDQCF die Registrierung für eine bestimmte SimulationId noch nicht unterstützt. Das JDQVisF muss daher alle Metriken von Hand beim JDQCF registrieren. Eine globale Überwachung der Simulation ist dadurch nur eingeschränkt möglich.

Abbildung 6-4 zeigt den Aufbau einer Subscribe-Nachricht an das JDQCF.

```
<DataQualitySubsciption>
    <MetrikId> Accuracy </MetrikId>
    <wsa:ReplyTo>
        <wsa:Address>
            http://example.JDQVisF.com
        </wsa:Address>
    </wsa:ReplyTo>
</DataQualitySubsciption>
```

Abbildung 6-4: Aufbau einer Subscribe-Nachricht an das JDQCF [1]

Für weiter Information sei an dieser Stelle an [1] verwiesen.

6.2.3 Schnittstelle für den Empfang von Rohdaten

Damit der Empfang von Rohdaten nicht auf das JDQCF beschränkt bleibt, bietet das JDQVisF über den Namespace aus Kapitel 5.4.3.1 */ res / raw / 'simulationId' / QoDValues.xml*. ein Verzeichnis für Rohdaten an. Dieses dient als Schnittstelle für den Empfang neuer Rohdaten. Ändert sich diese Datei, bemerkt es der *DQChangeListener* und ein neuer Visualisierungsdurchgang wird durch den *JDQVisController* gestartet.

Das JDQVisF bietet zwei Möglichkeiten für den Empfang neuer Rohdaten an. Zum einen durch die direkte Anbindung an das JDQCF durch den *QoDReceiver*, zum anderen bietet das JDQVisF durch die Klasse *DataReceiver* einen WebService für das Empfangen von Datenqualitätswerten an.

6.2.3.1 Funktionsweise des QoDReceiver

Der *QoDReceiver* ist eine Klasse des JDQVisF für das Empfangen von Datenqualitätswerten vom JDQCF. Sie implementiert dessen *DataQualityReceiverWebService*-Interface welches die *receive*-Methode enthält.

Hat das JDQCF neue Datenqualitätswerte errechnet, werden diese an den Service als SOAP-Nachricht übermittelt (Listing 3 zeigt einen Auszug). Der *QoDReceiver* empfängt diese Daten mit Hilfe der receive-Methode und schreibt sie in den entsprechenden Namespace aus Kapitel 5.4.3.1 */ res / raw / 'simulationId' / QoDValues.xml.*

```
<InterpretionCalculationResult>
    <Value> 0.95 </Value>
    <InterpretionId> Accuracy </InterpretionId>
</InterpretionCalculationResult>
```

Listing 3: Beispiel für ein InterpretionCalculationResult, das vom JDQCF an einen DataQualityReceiverWebServerices versendet wird

Für den genauen Ablauf des Versendens der Datenqualitätswerte und die Funktionsweise der *DispatchingAPI* des JDQCF, sei an dieser Stelle auf [1] verwiesen.

6.2.3.2 Funktionsweise des DataReceiver

Der *DataReceiver* ist eine Klasse die vom JDQVisF als WebService bereitgestellt wird. Sie dient für den Empfang neuer Datenqualitätswerte die nicht direkt vom JDQCF versendet werden. Möchte ein Datenerzeuger seine Daten mit Hilfe des JDQVisF visualisieren, so muss er zuerst einen *DataReceiver* instanziieren. Dazu benötigt er die *SimulationId* und drei Listen für die Datenqualitätsdimensionen, die dazu passenden Datenqualitätswerten und den Schwellenwerten der einzelnen Dimensionen.

Stehen neue Datenqualitätswerte bereit, so wird über die *writeDataToXml*-Methode des *DataReceiver* die neue Datenqualitätsdatei (siehe Kapitel 5.4.3.1) */ res / raw / 'simulationId' / QoDValues.xml* aus den Listen erstellt.

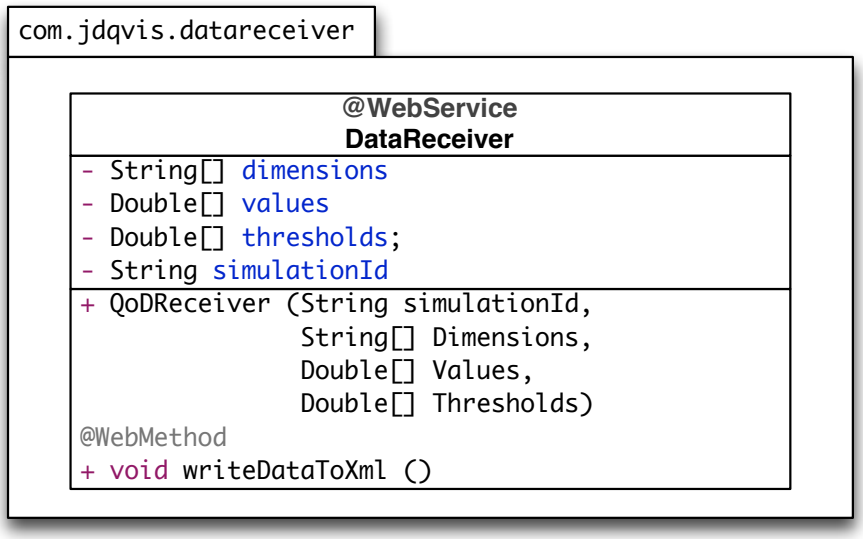


Abbildung 6-5: Klassendiagramm des DataReceiver

6.2.3.3 DirectoryWatcher und DQChangeListener

Ist der *JDQVisClient* erfolgreich am *JDQVisF* angemeldet, so instanziiert der *JDQVis-Controller* einen neuen *DirectoryWatcher* und einen *DQChangeListener* für den entsprechenden Namespace aus Kapitel 5.4.3.1 */ res / raw / 'simulationId' / QoDValues.xml*. Diese Klassen erkennen ob neue Rohdaten für die Visualisierung bereit stehen und benachrichtigen den *JDQVisController* über seine *DQFileChanged*-Methode. Diese startet daraufhin einen neuen Visualisierungsdurchgang. Sie durchsucht dabei die *activeVisSpecList* des *JDQVisController* nach aktiven *VisSpecifications*. Werden passende *VisSpecifications* gefunden, werden ihre VisualizationMediators instanziiert und deren *visualize*-Methode aufgerufen (siehe Kapitel 6.3.1 und 6.3.2).

6.2.4 Methoden für die Verarbeitung von Benutzerinteraktionen

Der JDQVisController bietet drei Methoden für die Verarbeitung von Benutzerinteraktionen an. Zwei für die Manipulation der *Filter-* und *VisualizerSpecification* und eine für das senden von Steuerbefehlen für die Simulation.

Für die Umsetzung der in Kapitel 6.5.3 beschriebene Beeinflussung der Visualisierungsspezifikationen bietet der JDQVisController die beiden Methoden *modifyFilterSpecification* und die *modifyVisualizerSpecification* an. Sie ermöglichen es dem *JDQVisClient,* alle Schritte der Visualisierungspipeline zu beeinflussen (siehe Kapitel 3.3.4). Da der grundsätzliche Ablauf bei beiden Methoden der selbe ist, wird dieser im Folgenden anhand der *modifyVisualizerSpecification* Methode erklärt.

6.2.4.1 Beeinflussung eine Visualizer-Plug-In

Mir Hilfe der *modifyVisualizerSpecification*-Methode des *JDQVisController* kann ein *JDQVisClient* den Visualisierungsschritt der Visualisierungspipeline beeinflussen. Dazu übergibt er seine *UserId,* die *SimulationId* und die Anfrage in Textform (siehe Kapitel 6.4.3). Ein Beispiel für eine Visualisierungsanfrage könnte „Zoom-In“ oder „showOverview“ sein. Der *JDQVisController* passt daraufhin die passende *VisualizerSpecification* an, indem er in das *request*-Element die erhaltene Anfrage schreibt.
Bei positiver Anpassung erhält der *JDQVisClient* eine Bestätigungsnachricht. Konnte die *VisualizerSpecification* nicht verändert werden, erhält er eine Fehlermeldung.

```
@WebMethod
public String modifyVisualizerSpecification(
            @WebParam(name='userId') String userId,
            @WebParam(name='simulationId') String simuId,
            @WebParam(name='request') String request
            )
```

Listing 4: Methodensignatur der modifyVisualizerSpecification-Methode des JDQVisController mit JAX-WS Annotationen

6.2.4.2 Steuerung der laufenden Simulation

Mit Hilfe der *sendSimulationControlRequest*-Methode des *JDQVisController,* kann ein *JDQVisClient* Steuerungsbefehle für eine Simulation senden. Dazu übergibt er seine *UserId*, die gewünschte *SimulationId* und den Steuerbefehl in Textform. Als Antwort kann ein Text vom SimulationController-Plug-In zurück gegeben werden.

```
@WebMethod
public String sendSimulationControllRequest(
            @WebParam(name='userId') String userId,
            @WebParam(name='simulationId') String simuId,
            @WebParam(name='request') String request
            )
```

Listing 5: Methodensignatur der sendSimulationControlRequest-Methode des JDQVisController mit JAX-WS Annotationen

Diese Methode wählt ein, zur *SimulationId* passendes, SimulationController-Plug-In über die *SimulationControllerFactory* Klasse aus. Diese implementiert das GoF-Factory-Pattern [41]. Die Klasse sucht in der *PlugInRegister.xml* nach einem simulationController-Plug-In das die übergebende *SimulationId* unterstützt. Ist ein passendes Plug-In gefunden, wird eine Instanz der Klasse zurück gegeben, die das *SimulationControllerInterface* implementiert. Anschließend wird die *processSimulationRequest*-Methode des Plug-Ins aufgerufen. Sie bekommt die *UserId* und den *Request* als Parameter übergeben. Als Antwort erwartet der *JDQVisController* ein Text den er dem *JDQVisClient* weiterleitet.

6.2.4.2.1 SimulationControllerInterface.jar

Der *JDQVisController* bietet über das *SimulationControllerInterface* die Möglichkeit, SimulationController-Plug-Ins in das JDQVisF einzubinden. Dazu muss es die *processSimulationRequest*-Methode implementieren. Diese bekommt als Parameter die *UserId*

und den Steuerbefehl in Textform, beispielsweise „*Abbruch*“. Als Rückgabe erwartet der *JDQVisController* einen Text.

Das *SimulationControllerInterface* ist unter dem in Kapitel 5.4.3.3 gezeigten Namespace */ interfaces / SimulationControllerInterface.jar* des JDQVisF zu finden.

6.2.4.2.2 Registrierung eines SimulationController-Plug-In

Um ein SimulationController-Plug-In zu registrieren, ist ein Eintrag in das PlugInRegister notwendig. Listing 6 zeigt den strukturellen Aufbau einer Registrierung im PlugInRegister.

```
<plugin type='simulationController'>
    <simulationId> example </simulationId>
    <url> ./plugins/simulationController/ExampleController.jar </url>
    <classname> controller.ExampleController </classname>
</plugin>
```

Listing 6: Struktur der Registrierung eines SimulationController-Plug-Ins im PlugInRegister.xml

/ plugin
: Repräsentiert ein JDQVisF-Plug-In. Das *type*-Attribut *simulationController* identifiziert es als SimulationController-Plug-In.

/ plugin / simulationId
: Beinhaltet die SimulationId für die dieses Plug-In geladen werden soll.

/ plugin / url
: Beinhaltet die URL an der das Plug-In zu finden ist. Das JDQVisF bietet dazu definierte Namespaces (siehe Kapitel 5.4.3.2) an. Prinzipiell kann die URL jedoch beliebig sein. So können auch externe *SimulationController* in das JDQVisF eingebunden werden. Dabei ist zu beachten, dass es sich um ein .jar-Archive handeln muss.

/ plugin / classname
: Dieses Element identifiziert die Klasse die das *SimulationControllerInterface* implementiert. Sie dient als Einstiegspunkt aller SimulationController-Plug-Ins und wird vom *JDQVisController* zur Verarbeitung von Steuerbefehlen aufgerufen.

6.3 Umsetzung des VisualizationMediator

Der *VisualizationMediator* ist für das Generieren der Visualisierungen zuständig. Er steuert die Visualisierungspipeline durch das Laden der richtigen *Filter*-, *Visualizer*- und *Dispatcher*-Plug-Ins.

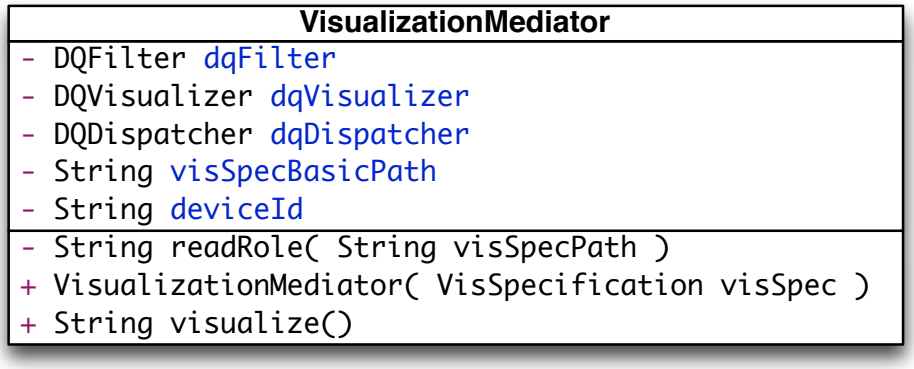


Abbildung 6-6: Klassendiagramm des VisualizationMediator

In den folgenden Abschnitten werden die einzelnen Methoden und ihr Zusammenspiel aufgezeigt.

6.3.1 Instanziierung eines VisualizationMediators

Empfängt das JDQVisF neue Datenqualitätswerte wird über die *DQFileChanged*-Methode des *JDQVisController* für jede aktive *VisSpecification* überprüft, ob sie die übergebene *SimulationId* als Attribut besitzt. Ist dies der Fall wird festgestellt, ob ihr *VisualizationMediator* schon instanziiert wurde. Wenn nein, wird ein neuer VisualizationMediator erstellt.
Der Konstruktor des VisualizationMediator erwartet das *VisSpecification*-Objekt. Zuerst generiert er aus dessen Attributen seinen *visSpeciBasicPath.* Dieser zeigt auf den in Kapitel 5.4.3.1 gezeigten Namespace */ res / visspecifications / 'simuId' / user / 'userId'* und dient als Referenz für alle Visualisierungs-Plug-Ins. Anschließend werden die entsprechenden Plug-Ins über die jeweiligen *Factory*-Klassen instanziiert.

Abbildung 6-7 zeigt ein Sequenzdiagramm der einzelnen Schritte. Der Übersicht halber, werden die jeweiligen Plug-In-Klassen nicht gezeigt.

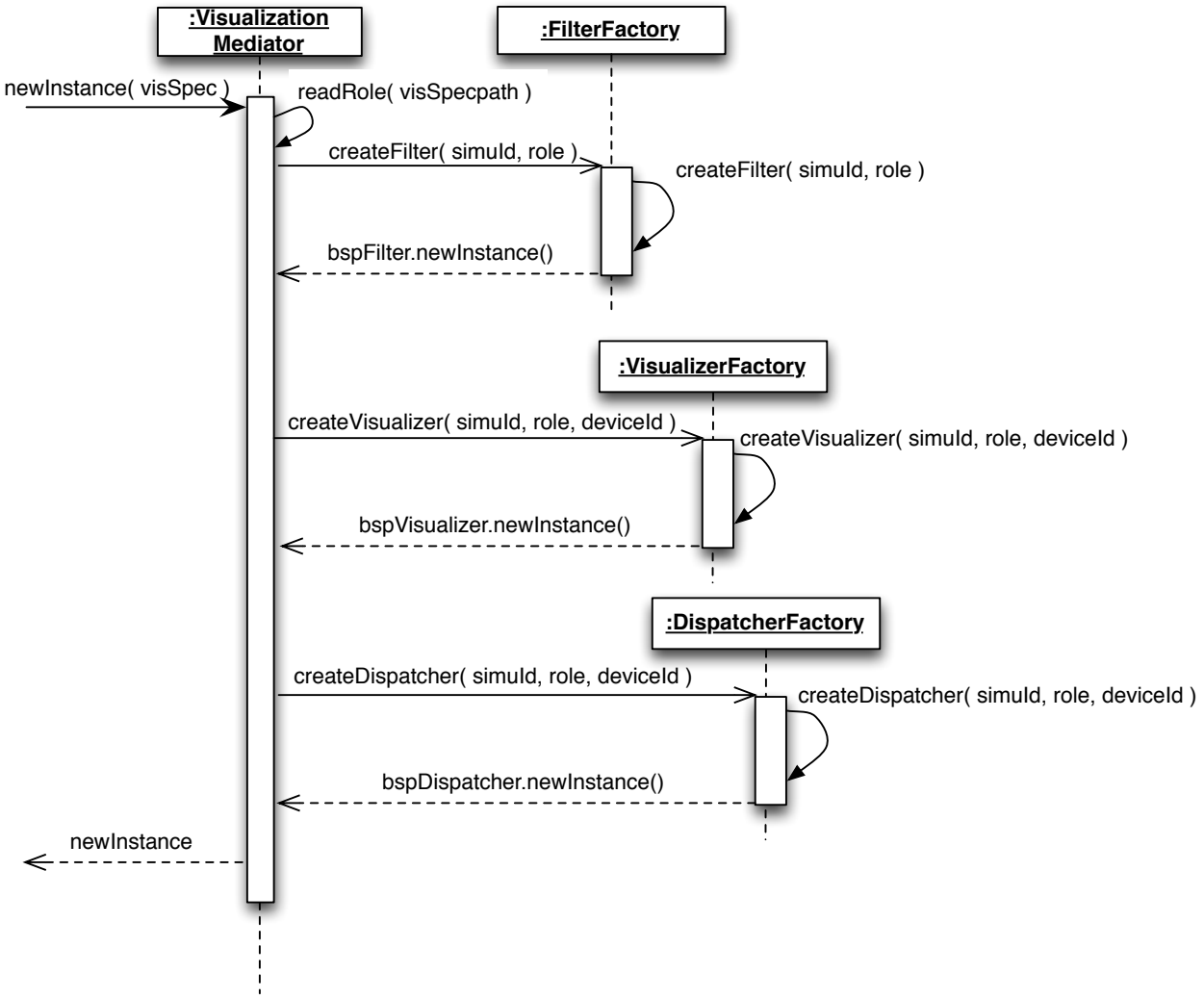


Abbildung 6-7: Ablauf der Instanziierung eines VisualizationMediator

6.3.2 Visualize-Methode des VisualizationMediators

Wurde ein *VisualizationMediator* erfolgreich instanziiert, kann seine *visualize*-Methode aufgerufen werden. Da alle nötigen Plug-Ins bereits in den Attributen des *Visualization-Mediators* bei der Instanziierung gesetzt wurden, erwartet diese Methode keine Parameter. Als Rückgabe liefert sie einen Logbericht. Dieser enthält Informationen über den Status der Visualisierung.

Abbildung 6-8 zeigt ein den Ablauf mit Hilfe eines Sequenzdiagramms.

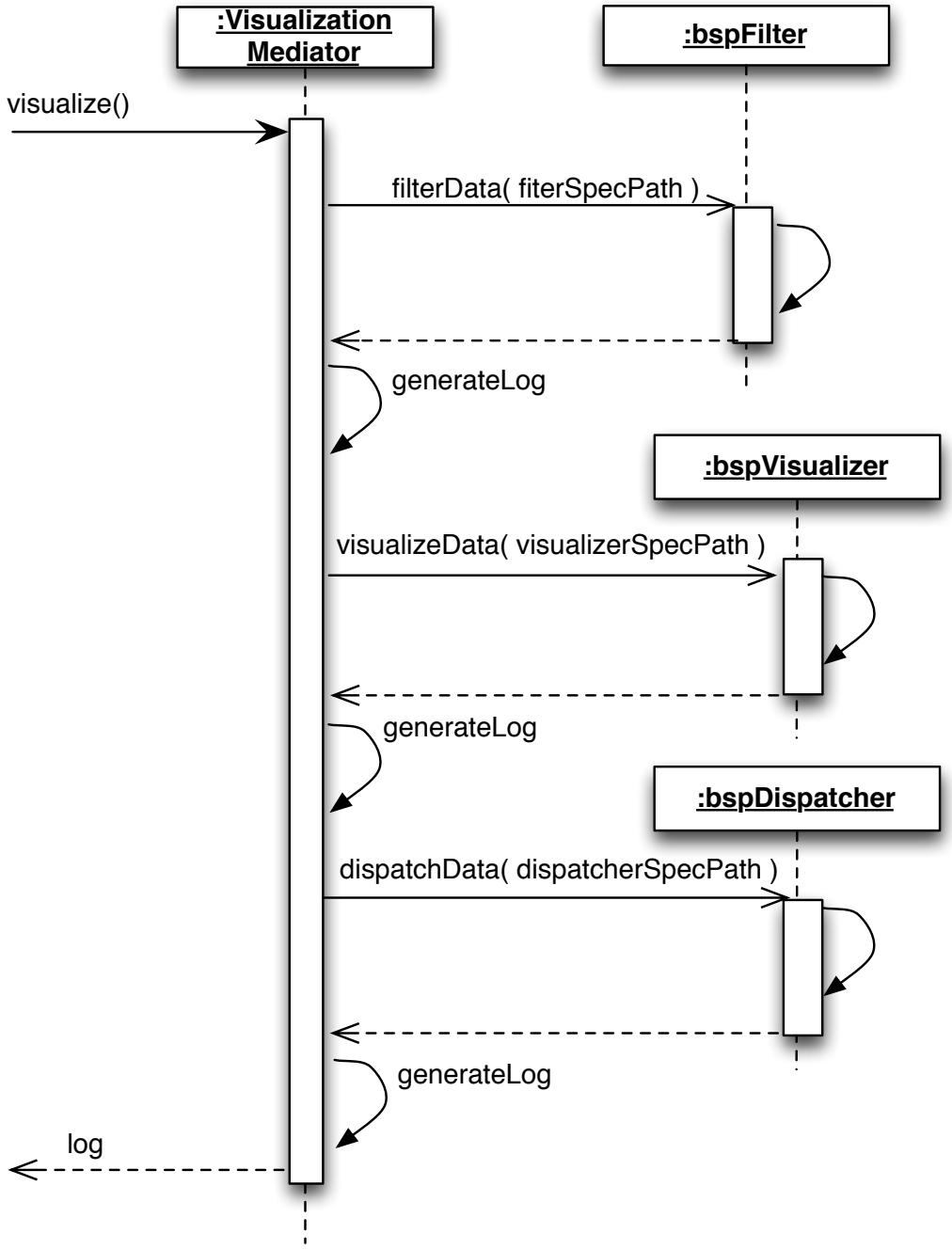


Abbildung 6-8: Ablauf der visualize-Methode des VisualizationMediator

6.3.3 Erweiterungsschnittstellen des VisualizationMediator

Dieser Abschnitt zeigt die Umsetzung der Schnittstellen die vom JDQVisF für das Einbinden neuer Visualisierungs-Plug-Ins bereit gestellt werden.

6.3.3.1 FilterInterface.jar

Das JDQVisF bietet über das *FilterInterface.jar* Entwicklern die Möglichkeit neue Filter-Plug-Ins einzubinden. Das *DQFilterInterface* enthält lediglich die *filterData*-Methode. Sie erwartet als Eingabe den Pfad unter dem die gewünschte *FilterSpecification* zu finden ist.

Das *FilterInterface* ist unter dem in Kapitel 5.4.3.3 gezeigten Namespace */ interfaces / FilterInterface.jar* des JDQVisF zu finden.

6.3.3.2 VisualizerInterface.jar

Das JDQVisF bietet über das *VisualizerInterface* die Möglichkeit neue Visualisierungsalgorithmen für den *VisualizationMediator* einzubinden. Das VisualizerInterface enthält die *visualizeData*-Methode. Diese bekommt als Parameter den Pfad zu einer passenden *VisualizerSpecification* und ist der Einstiegspunkt für jedes Visualizer-Plug-In.

Das *VisualizerInterface* ist unter dem in Kapitel 5.4.3.3 gezeigten Namespace */ interfaces / VisualizerInterface.jar* des JDQVisF zu finden.

6.3.3.3 DispatcherInterface.jar

Das DispatcherInterface des JDQVisF ermöglicht den Plug-In-Entwickler neue Versandarten für die berechneten Visualisierungen einzubinden. Es enthält das *DQDispatcher* Interface, welches wiederum die dispatchData-Methode enthält. Diese Methode ist er Einstiegspunkt für jedes Dispatcher-Plug-In.

Das *DispatcherInterface* ist unter dem in Kapitel 5.4.3.3 vorgestellten Namespace */ interfaces / DispatcherInterface.jar* des JDQVisF zu finden.

6.3.3.4 Hilfsklassen der Interfaces

Um das Realisieren der Interfaces für die Entwickler zu vereinfachen und um die Interoperabilität der verschiedenen Plug-Ins sicherzustellen, enthalten die vorgestellten Schnittstellen neben der reinen Interfacebeschreibung zusätzlich folgende Hilfsklassen:

DQReader – Er ließt mit seiner *readQoDValues*-Methode aus den Rohdaten alle wichtigen Daten aus. Dazu gehören: Die Skala auf der die Werte liegen und die Adresse bei der die originalen Simulationsdaten liegen. Zusätzlich bietet er eine Liste mit *DQCalculationResults* an. Dabei besteht ein *DQCalculationResult* immer aus einer Beschreibung, einem Wert und dem Schwellenwert. Also Beispielsweise „*Accuracy, 0.9, 0.5“*.

PropertiesReader – Er ließt mit seiner *readProperties*-Methode aus der Visualisierungsspezifikation alle Attribute und Elemente. Zu den Attributen gehören die SimulationId und die Rolle. Als Elemente werden beispielsweise der Ein- und Ausgabepfad der Daten und der *Request* des Wissenschaftlers zurück gegeben.

6.3.4 Registrierung der Visualisierungs-Plug-Ins

Damit die einzelnen Visualisierungs-Plug-Ins vom *VisualizationMediator* geladen werden können, müssen sie in der *PlugInRegister.xml* registriert werden. Die folgenden Abschnitte zeigen die jeweilige Struktur dieser Einträge.

6.3.4.1 Registrierung eines Filter-Plug-In

Damit der *VisualizationMediator* ein Filter-Plug-In verwenden kann, muss es zunächst in der Plug-In-Register.xml registriert werden. Listing 7 zeigt die Struktur der Registrierung eines Filters in der PlugInRegister.xml.

```
<plugin type='dqfilter'>
    <role> ... </role>+
    <simulationId> ... </simulationId>+
    <url> ... </url>
    <classname> ... </classname>
</plugin>
```

Listing 7: Struktur der Registrierung eines Filter-Plug-In in der PluginRegister.xml

/ plugin
: Repräsentiert ein JDQVisF-Plug-In. Das *type*-Attribut *dqfilter* identifiziert es als Filter-Plug-In das vom VisualizationMediator geladen werden kann.

/ plugin / role+
: Dieses Element identifiziert alle Rollen für die das Plug-In geladen werden soll.

/ plugin / simuId+
: Beinhaltet alle SimulationIds für die dieses Plug-In geladen werden soll.

/ plugin / url
: Beinhaltet die URL an der das Plug-In zu finden ist. Das JDQVisF bietet dazu definierte Namespaces (siehe Kapitel 5.4.3.2) an. Prinzipiell kann die URL jedoch beliebig sein. So können auch externe Filter-Plug-Ins in das JDQVisF eingebunden werden. Dabei ist zu beachten, dass es sich um ein .jar-Archive handeln muss.

/ plugin / classname
Dieses Element identifiziert die Klasse die das *FilterInterface* implementiert. Sie dient als Einstiegspunkt aller Filter-Plug-Ins und wird vom *VisualizationMediator* zu Begin des Visualisierungsprozesses geladen.

6.3.4.2 Registrierung eines Visualizer-Plug-In

Damit der *VisualizationMediator* ein Visualizer-Plug-In verwenden kann, muss es in der PlugInRegister.xml registriert sein. Listing 8 zeigt die Struktur der Registrierung eines Visualizer-Plug-In in der Register.xml.

```
<plugin type='dqvisualizer'>
    <role> ... </role>+
    <simulationId> ... </simulationId>+
    <deviceId> ... </deviceId>+
    <url> ... </url>
    <classname> ... </classname>
</plugin>
```

Listing 8: Struktur der Registrierung eines Visualizer-Plug-In in der PlugInRegister.xml

/ plugin
Repräsentiert ein JDQVisF-Plug-In. Das *type*-Attribut *dqvisualizer* identifiziert es als Visualizer-Plug-In das vom *VisualizationMediator* geladen werden kann.

/ plugin / role+
Dieses Element identifiziert alle Rollen für die das Plug-In geladen werden soll.

/ plugin / simuId+
Beinhaltet alle SimulationIds für die dieses Plug-In geladen werden soll.

/ plugin / deviceId
Beinhaltet alle Anzeigegeräte für die dieses Plug-In geladen werden soll.

/ plugin / url
Beinhaltet die URL an der das Plug-In zu finden ist. Das JDQVisF bietet dazu definierte Namespaces (siehe Kapitel 5.4.3.2) an. Prinzipiell kann die URL jedoch beliebig sein. So können auch externe Visualizer-Plug-Ins in das JDQVisF eingebunden werden. Dabei ist zu beachten, dass es sich um ein .jar-Archive handeln muss.

/ plugin / classname
Dieses Element identifiziert die Klasse die das *VisualizerInterface* implementiert. Sie dient als Einstiegspunkt aller Visualizer-Plug-Ins und wird vom *VisualizationMediator* nach dem Filtern der Daten geladen.

6.3.4.3 Registrierung eines Dispatcher-Plug-In

Damit der *VisualizationMediator* ein Dispatcher-Plug-In für das Versenden der generierten Bilder verwenden kann, muss diese in der PlugInRegister.xml registriert sein. Listing 9 zeigt die Struktur zur Registrierung eines Dispatcher-Plug-In in der PlugInRegister.xml.

```
<plugin type='dqdispatcher'>
    <role> ... </role>+
    <simulationId> ... </simulationId>+
    <deviceId> ... </deviceId>+
    <url> ... </url>
    <classname> ... </classname>
</plugin>
```

Listing 9: Struktur der Registrierung eines Dispatcher-Plug-In in der PlugInRegister.xml

/ plugin
: Repräsentiert ein JDQVisF-Plug-In. Das *type*-Attribut *dqdispatcher* identifiziert es als Dispatcher-Plug-In das vom *VisualizationMediator* geladen werden kann.

/ plugin / role+
: Dieses Element identifiziert alle Rollen für die das Plug-In geladen werden soll.

/ plugin / simuId+
: Beinhaltet alle SimulationIds für die dieses Plug-In geladen werden soll.

/ plugin / deviceId+
: Beinhaltet alle Anzeigegeräte für die dieses Plug-In geladen werden soll.

/ plugin / url
: Beinhaltet die URL an der das Plug-In zu finden ist. Das JDQVisF bietet dazu definierte Namespaces (siehe Kapitel 5.4.3.2) an. Prinzipiell kann die URL jedoch beliebig sein. So können auch externe Dispatcher-Plug-Ins in das JDQVisF eingebunden werden. Dabei ist zu beachten, dass es sich um ein .jar-Archive handeln muss.

/ plugin / classname
: Dieses Element identifiziert die Klasse die das *DispatcherInterface* implementiert. Sie dient als Einstiegspunkt aller Disaptcher-Plug-Ins und wird vom *VisualizationMediator* nach dem Visualisieren der Daten geladen.

6.3.5 Realisierung der Visualisierungsspezifikationen

Dieser Abschnitt zeigt die Realisierung der vorgestellten Visualisierungsspezifikationen.

6.3.5.1 FilterSpecification

Eine FilterSpecification liegt unter dem in Kapitel 5.4.3.1 vorgestellten Namespace */ res / visspecification / 'simuId' / user / 'userId' / FilterSpecification.xml.* Sie existiert für jedes SimulationId-UserId-Paar und enthält alle Elemente, die für das Aufbereiten der Rohdaten wichtig sind.

Listing 10 zeigt den Aufbau einer FilterSpecification.xml.

```
<filterDescription role="Owner" simulationId="'simuId'">
    <inputpath>./res/raw/'simuId'/QoDvalues.xml</inputpath>
    <outputpath>./res/filtered/'simuId'/'userId'/Data.xml</outputpath>
    <request>showAllData</request>
</filterDescription>
```

Listing 10: Aufbau der FilterSpecification.xml

/ filterDescription
: Wurzelelement jeder FilterSpecification. Als Attribute hat es die Rolle und die SimulationId für die das Plug-In verwendet wird.

/ filterDescription / inputpath
: Dieses Element repräsentiert den Ort an dem die Rohdaten liegen. Dabei kann es sich insbesondere um die Daten des JDQCF handeln.

/ filterDescription / outputpath
: Diese Element repräsentiert den Ort an dem die aufbereiteten Daten gespeichert werden müssen, damit sie in den weiteren Schritten des Visualisierungsprozess verwendet werden können.

/ filterDescription / filterRequest
: Dieses Element kann zur Manipulation des Filter-Plug-Ins verwendet werden (ℜ14). Es wird vom JDQVisController gesetzt, sobald er einen *FilterRequest* von einen *JDQVisClienten* empfangen hat. Ein *FilterRequest* kann dabei beliebige Anweisungen enthalten die von dem Filter-Plug-In umgesetzt werden können. Dabei kann das JDQVisF nicht garantieren, dass dieser Request auch verarbeitet wird. Es bietet über dieses Element lediglich die Schnittstelle zwischen Wissenschaftler und Filter-Plug-In an. Die Verantwortung der Bearbeitung liegt beim Filter-Plug-In.

6.3.5.2 VisualizerSpecification

Eine *VisualizerSpecification* liegt unter dem in Kapitel 5.4.3.1 gezeigten Namespace */ res / visspecification / 'simuId' / user / 'userId' / VisualizerSpecification.xml.* Sie existiert für jedes SimulationId-UserId-Paar und definiert alle Elemente, die für das Visualisieren der Daten wichtig sind.

Listing 11 zeigt den Aufbau einer VisualizerSpecification.xml in Pseudo-XML.

```
<visualizationDescription role="Owner" simulationId="'simuId'">
    <inputpath>./res/filtered/'simuId'/'userId'/Data.xml</inputpath>
    <outputpath>./res/visualized/'simuId'/'userId'/</outputpath>
    <request>Portrait</request>
</visualizationDescription>
```

Listing 11: Aufbau einer VisualizerSpecification.xml

/ visualizationDescription
: Wurzelelement jeder *VisualizerSpecification*. Als Attribute hat es die Rolle und die SimulationId für die das Plug-In verwendet wird.

/ visualizationDescription / inputpath
: Dieses Element repräsentiert den Ort an dem die Eingabedaten liegen. Dabei kann es sich insbesondere um die zuvor aufbereiteten Daten durch ein Filter-Plug-In handeln.

/ visualizationDescription / outputpath
: Diese Element repräsentiert den Ort an dem die Visualisierungen gespeichert werden müssen, damit sie von einem Dispatcher-Plug-In verwendet werden können.

/ visualizationDescription / visualizationRequest
: Dieses Element kann zur Manipulation des Visualizer-Plug-Ins verwendet werden (ℜ14). Es wird vom JDQVisController gesetzt, sobald er einen *VisualizationRequest* von einen *JDQVisClienten* empfangen hat. Ein *VisualizationRequest* kann dabei beliebige Anweisungen enthalten, die von dem Visualizer-Plug-In umgesetzt werden können. Dabei kann das JDQVisF nicht garantieren, dass dieser Request auch verarbeitet wird. Es bietet über dieses Element lediglich die Schnittstelle dafür an. Die Verantwortung liegt bei dem Plug-In-Entwickler.

6.3.5.3 DispatcherSpecification

Eine *DispatcherSpecification* liegt unter dem in Kapitel 5.4.3.1 vorgestellten Namespace */ res / visspecification / 'simuId' / user / 'userId' / DispatcherSpecification.xml.* Sie exis-

tiert für jedes SimulationId-UserId-Paar und definiert alle wichtigen Elemente, die für das Versenden der Daten wichtig sind.

Listing 12 zeigt den Aufbau einer DispatcherSpecification.

```
<dispatchDescription role="Owner" simulationId="'simuId'">
    <inputpath>./res/visualized/'simuId'/'userId'/</inputpath>
    <inputpathRaw>./res/raw/'simuId'/</inputpathRaw>
    <outputAddress>
        http://192.168.1.2/JDQVis/'simuId'/visualizations/'userId'/
    </outputAddress>
    <cachepathRaw>./cache/'simuId'/raw/</cachepathRaw>
    <cachepathVisualization>./cache/'simuId'/visualization/'userId'/</cache
</dispatchDescription>
```

Listing 12: Aufbau einer DispatcherSpecification.xml

/ dispatchDescription
: Wurzelelement jeder *DispatchSpecification*. Als Attribute hat es die Rolle und die SimulationId für die das Plug-In verwendet wird.

/ dispatchDescription / inputpath
: Dieses Element repräsentiert den Ort an dem die Visualisierungen als Eingabedaten liegen.

/ dispatchDescription / inputpathRaw
: Dieses Element repräsentiert den Ort an dem die Simulationsdaten als Eingabedaten liegen.

/ dispatchDescription / outputaddress
: Diese Element repräsentiert die Adresse an die das Dispatcher-Plug-In die Daten versenden soll.

/ dispatchDescription / cachpathRaw
: Dieses Element repräsentiert die Adresse des Caches an den die Simulationsdaten zusätzlich kopiert werden können.

/ dispatchDescription / cachpathVisualization
: Dieses Element repräsentiert die Adresse des Caches an den die Visualisierungen zusätzlich kopiert werden können.

6.4 Umsetzung der Plug-Ins

Dieses Kapitel beschreibt die Entwicklung von Plug-Ins zur Erweiterung des JDQVisF. Dazu wird jeweils ein Authorizer-, ein Filter-, ein Visualizer-, ein Dispatcher- und ein SimulationController-Plug-In beschrieben.

Bevor die einzelnen Implementierungen vorgestellt werden, wird zunächst der grundsätzliche Ablauf bei der Entwicklung eines Plug-Ins zur Erweiterung des JDQVisF gezeigt:

Schritt 1: Einbindung des *Interface*.jar – Für die Entwicklung eines neuen Plug-Ins ist es notwendig, das entsprechende *Interface*.jar in den *Java Build Path* einzubinden. Das *Interface*.jar findet der Entwickler in dem vorgestellten Namespace */ res / interfaces* (siehe Kapitel 5.4.3.3).

Schritt 2: Implementierung – Damit das JDQVisF das neue Plug-In verwenden kann, ist es notwendig die entsprechenden Interfaces zu Implementieren.

Schritt 3: Test – Da das JDQVisF keine Kontrolle über die Korrektheit der Plug-Ins hat, ist es sehr wichtig das neue Plug-In gewissenhaft zu testen um Fehler zur Laufzeit zu vermeiden.

Schritt 4: Einbindung in das JDQVisF – Nach dem erfolgreichen Testen, wird das neue Plug-In in den vorgestellten Namespace (siehe Kapitel 5.4.3.2) als lauffähiges .jar-Archiv exportiert. Lauffähig bedeutet in diesem Fall, dass alle vom Plug-In benötigten Ressourcen in dem .jar-Archiv enthalten sind.

Schritt 5: Registrierung in der PlugInRegister.xml – Damit das neue Plug-In zur Laufzeit vom JDQVisF geladen werden kann, muss es in der PlugInRegister.xml registriert werden. Dazu werden beispielsweise die URL, unter der das Plug-In gefunden werden kann, und eine *SimulationId,* für die das Plug-In verwendet werden soll, eingetragen. Die genauen Einträge wurden, für Authorizer-Plug-Ins in Kapitel 6.2.1.2, für SimulationController-Plug-Ins in Kapitel 6.2.4.2.2 und für Plug-Ins zur Erweiterung der Visualisierungspipeline im Kapitel 6.3.4, gezeigt.

6.4.1 Beispielimplementierung eines Authorizer-Plug-In

Ein Authorizer-Plug-In wird von JDQVisController zur Autorisierung eines Wissenschaftlers für eine Simulation geladen. Das bedeutet, dass für jede Simulation innerhalb des JDQVisF ein Authorizer-Plug-In existieren muss.

Das hier entwickelte Authorizer-Plug-In realisiert das AuthorizationInterface (siehe Kapitel 6.2.1.1) und implementiert die authorize-Methode. Diese bekommt als Eingabe einen

Benutzernamen und ein Passwort und gibt ein *JDQVisUser*-Objekt zurück. Sind die Anmeldedaten nicht korrekt, wird *null* zurück gegeben.
Zur Überprüfung der Benutzerdaten besitzt das Plug-In ein XML-Dokument. Es enthält alle an der Simulation beteiligten Wissenschaftler mit ihrer Rolle. Listing 13 zeigt einen Beispieleintrag.

```
<user name='aName' password='123'>
    <userId>aUserId</userId>
    <role>Owner</role>
</user>
```

Listing 13: Beispieleintrag in die Benutzerdatenbank des Authorizer-Plug-Ins

6.4.2 Beispielimplementierung eines Filter-Plug-In

Für die Datenaufbereitung können in die Visualisierungspipeline eines Visualization-Mediator Filter-Plug-Ins eingebunden werden. Dieser Abschnitt beschreibt eine Beispielimplementierung eines Filters, der das FilterInterface (siehe Kapitel6.3.3.1) realisiert. Um die vielseitigen Möglichkeiten eines Filter-Plug-Ins zu zeigen, wird ein Filter für einen Wissenschaftler mit allen Rechten gezeigt. Für die Überwachung der Simulation soll der Wissenschaftler neben den Datenqualitätsvisualisierungen, zusätzlich Visualisierungen der originalen Simulationsdaten angezeigt bekommen. Die Simulationsdaten liegen dabei auf einem externen Server und müssen durch das Filter-Plug-In vor der Visualisierung geladen und aufbereitet werden.

Der Einstiegspunkt des Filter-Plug-Ins ist die *filterData*-Methode. Diese bekommt als Parameter den Pfad zu einer passenden *FilterSpecification* (siehe Kapitel 6.3.5.1).

Abbildung 6-9 zeigt vereinfacht die einzelnen Schritte der Datenaufbereitung innerhalb des Filters als Sequenzdiagramm ohne technische Details.

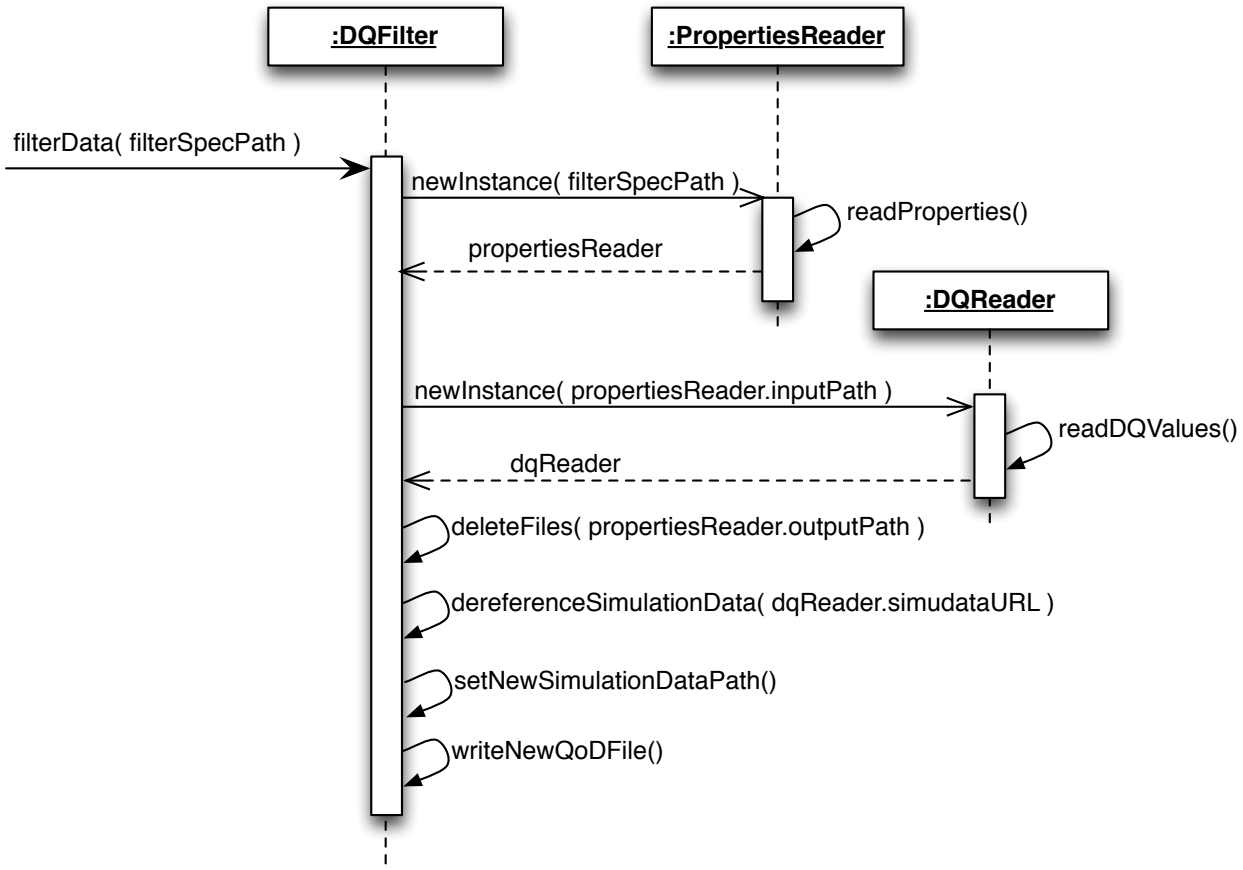


Abbildung 6-9: Sequenzdiagramm des Filter-Plug-Ins

Im ersten Schritt instanziiert der Filter einen *PropertiesReader* und einen *DQReader*. Diese stellen alle wichtige Informationen wie den Ein- und Ausgabepfad und die Referenz der Simulationsdaten bereit (siehe Kapitel 6.3.3.4).
Anschließend löscht das Filter-Plug-In alle alten Dateien innerhalb des Ausgabepfades (siehe Kapitel 5.4.3.1). Dadurch wird sicher gestellt, dass ein Visualizer-Plug-In nur die Daten erhält die es für die Visualisierung benötigt.
Im dritten Schritt lädt der Filter die referenzierten Simulationsdaten von einem Server und kopiert sie in den Namespace aus Kapitel 5.4.3.1. Die Adresse des Servers, bei dem die Simulationsdaten liegen, ist in den Rohdaten (*QoDValues.xml,* siehe Kapitel 6.2.3) enthalten und wird mit Hilfe des DQReader ausgelesen.
Anschließend wird die Adresse der Simulationsdaten auf den neuen lokalen Pfad des JDQVisF gesetzt und die modifizierte QoDValues.xml an den entsprechenden Namespace aus Kapitel 5.4.3.1 geschrieben.

Da es sein kann, dass zu einer Simulation keine externen Daten angegeben sind, werden hier die oben genannten Schritte für das Laden der Simulationsdaten übersprungen und nur die Datenqualitätswerte entsprechend verarbeitet.

6.4.3 Beispielimplementierung eines Visualizer-Plug-In

Ein Visualizer-Plug-In realisiert die zweite Stufe der Visualisierungspipeline eines *VisualizationMediator* und implementiert das *VisualizerInterface* (siehe Kapitel 6.3.3.2). In diesem Kapitel wird ein Visualizer-Plug-In vorgestellt, welches die Dataqualitätsvisualisierungen aus Kapitel 3.9.2 generiert und in Kombination mit den Visualisierungen der Simulationsdaten zu einem Gesamtbild zusammenfügt. Das beutetet, dass es die Schritte *Mapping* und *Rendering* der Visualisierungspipeline übernimmt und die fertigen Bilder für die Anzeigegeräten bereitstellt. Zusätzlich bietet das Visualizer-Plug-In einem *JDQVisClient* die Möglichkeit, die Bildgenerierung mit Hilfe eines *Requests* (siehe Kapitel 6.3.5.2) zu steuern. Der Wissenschaftler kann dadurch wählen ob er nur Datenqualitätsvisualisierungen im Querformat oder Datenqualitätsvisualisierungen zusammen mit Simulationsvisualisierungen im Hochformat angezeigt bekommt (siehe Appendix A und B).

Abbildung 6-10 zeigt ein vereinfachtes Sequenzdiagramm mit allen wichtigen Schritten für die Generierung der Visualisierungen. Dabei wird zwischen den beiden Alternativen *Landscape* und *Portrait*, gekennzeichnet durch *alt,* unterschieden. Technische Details werden für eine besseren Übersicht weggelassen.

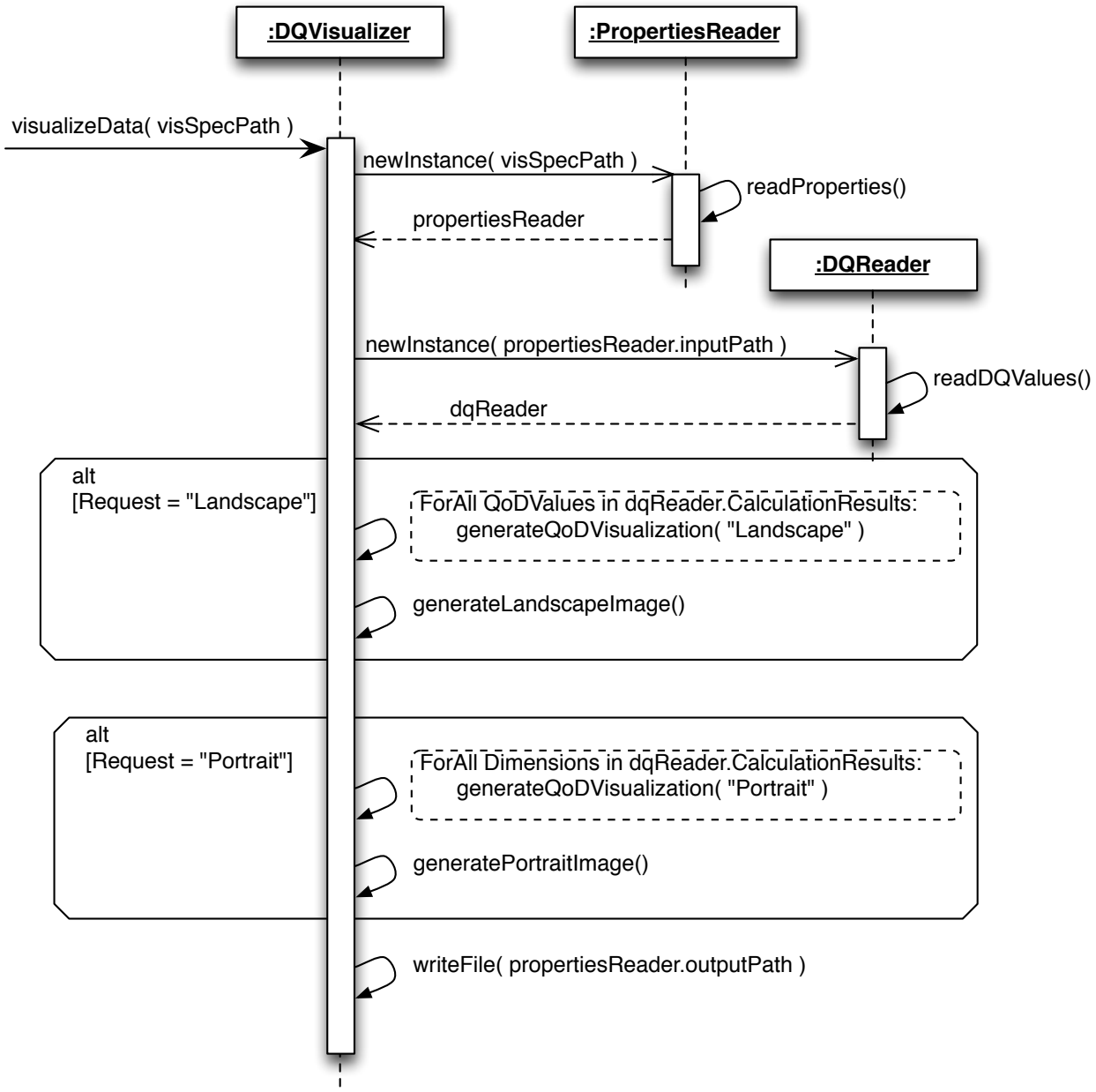


Abbildung 6-10: Vereinfachter Ablauf des Visualizer-Plug-In mit zwei alternativen Pfaden

Im ersten Schritt werden jeweils ein *PropertiesReader* und ein *DQReader* instanziiert. Sie stellen alle wichtigen Pfade und Daten für die Visualisierung bereit (siehe Kapitel 6.3.3.2). Anschließend wird je nach *Request* ein Bild, das entweder eine nur Datenqualitätsvisualisierungen oder Datenqualitätsvisualisierungen mit Simulationsvisualisierungen enthält, generiert und an den entsprechenden Namespace aus Kapitel 5.4.3.1 gespeichert.

Der folgende Abschnitt beschreibt beispielhaft das Visualisieren der einzelnen Datenqualitätsdimensionen anhand der Dimension *Rechtzeitigkeit* (siehe Kapitel 3.9.2.2). Der grundsätzliche Ablauf ist bei allen Dimensionen der gleiche.

Visualisierung der Dimension Rechtzeitigkeit

Die Visualisierung der Dimension *Rechtzeitigkeit* gliedert sich in zwei Teile. Zuerst wird ein Basisbild geladen. Je nachdem ob der Datenqualitätswert größer oder kleiner ist als der Schwellenwert, ist das ist ein Wecker mit grünem oder rotem Ziffernblatt. Abbildung 6-11 zeigt die beiden Basisbilder.

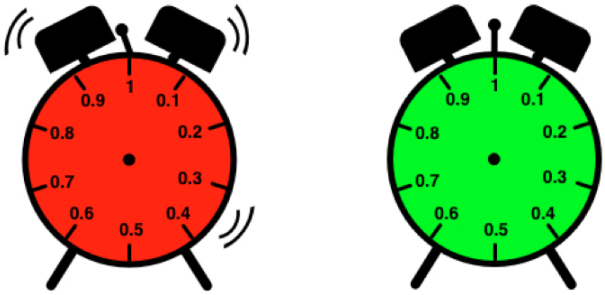


Abbildung 6-11: Basisbilder der Dimension Rechtzeitigkeit. Links schlechte, rechts gute Datenqualität

Anschließend werden der Datenqualitätswert und der Schwellenwertes als Zeiger in das Bild gezeichnet. Zum leichtern Erfassen werden die Werte zusätzlich unter dem Icon als Zahlenwert geschrieben (Abbildung 6-12).

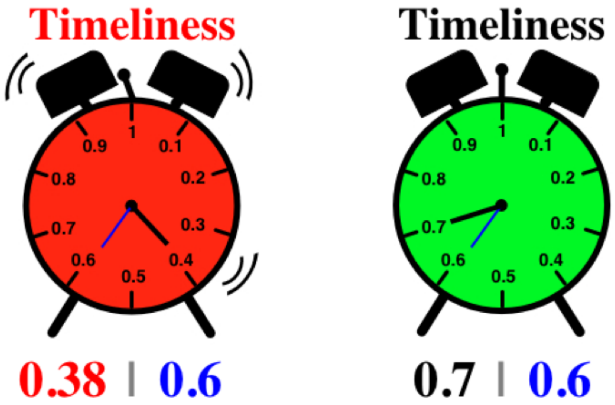


Abbildung 6-12: Visualisierung der Dimension Rechtzeitigkeit mit den eingefügten QoD- und Schwellenwerten. Links schlechte, rechts gute Datenqualität

6.4.4 Beispielimplementierung eines Dispatcher-Plug-In

Ein Dispatcher-Plug-In wird am Ende der Visualisierungspipeline vom *VisualizationMediator* geladen. Es versendet die generierten Visualisierungen an die angegebene Ad-

resse aus der *DispatcherSpezfication.xml* (Kapitel 6.3.5.3). Grundsätzlich können dabei alle möglichen Versandarten und -protokolle umgesetzt werden.

Als Beispielimplementierung wird ein Dispatcher-Plug-In gezeigt, welches die generierten Visualisierungen auf einen Server kopiert. Von diesem können die Visualisierungen anschließend über URL-Anfragen von den Anzeigegeräten geladen werden. Diese URLs sind für alle *UserIds* unterschiedlich, so dass jeder Wissenschaftler genau die Visualisierungen angezeigt bekommt, die für ihn generiert wurden.

Abbildung 6-13 zeigt einen vereinfachten Ablauf ohne technische Details.

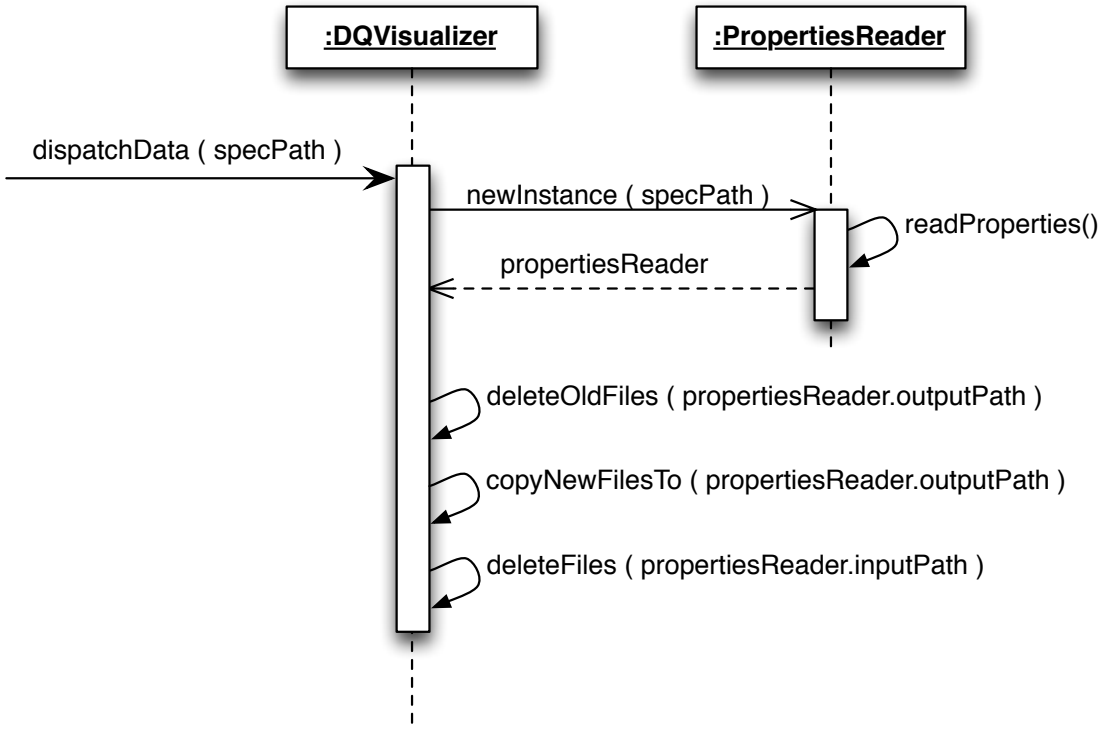


Abbildung 6-13: Vereinfachter Ablauf eines Dispatcher-Plug-Ins

Im ersten Schritt wird ein *PropertiesReader,* der die URL des Ausgabepfades enthält, instanziiert. Anschließend werden alle Dateien die sich im späteren Zielverzeichnis des Server befinden gelöscht. Dadurch wird sicher gestellt, dass dem Wissenschaftler immer die aktuellsten Visualisierungen angezeigt bekommt. Im nächsten Schritt werden alle Dateien die sich im Eingabepfad befinden in das Zielverzeichnis des Server kopiert und somit dem Wissenschaftler zur Verfügung gestellt. Im letzten Schritt werden alle neuen Visualisierungen im Eingabeverzeichnis gelöscht. Dadurch wird sichergestellt, dass im nächsten Verteilungsdurchgang keine alten Daten an den Server übergeben werden.

6.4.5 Beispielimplementierung eines SimulationController-Plug-In

Ein SimulationController-Plug-In ist eine Erweiterung des JDQVisF, mit dessen Hilfe der Wissenschaftler in die laufende Simulation eingreifen kann. Es wird vom JDQVisController beim Empfang eines Steuerungsbefehls geladen und dessen *processSimulationRequest*-Methode aufgerufen.

Da der Fokus dieser Arbeit auf dem Visualisieren von Datenqualität liegt, wird in diesem Abschnitt lediglich eine prototypische Umsetzung eines SimulationController-Plug-Ins gezeigt. Dieses öffnet ein Fenster, das den Steuerungsbefehl und die UserId des Wissenschaftlers anzeigt. Ein Wissenschaftler kann dann diesen Request, beispielsweise als *Web Services Human Task* (*WS-HumanTask*) [38], in der laufende Simulation ausführen. (Siehe Kapitel 7.4)

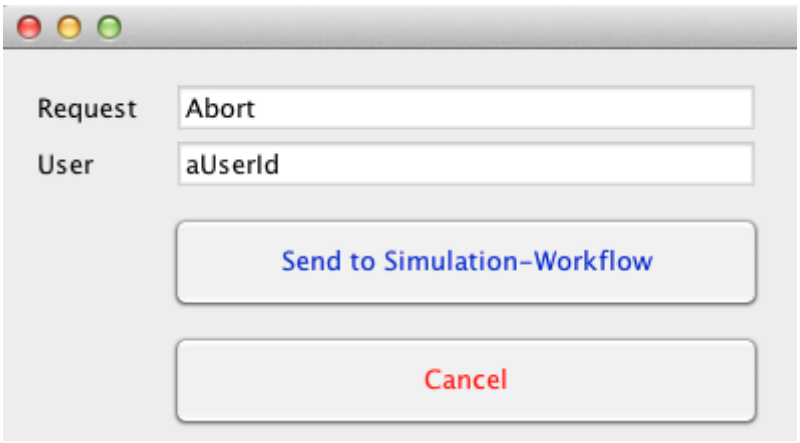


Abbildung 6-14: Beispielimplementierung eines SimulationController-Plug-In

6.5 Der JDQVisClient

Diese Kapitel beschreibt das JDQVisF aus Sicht der *JDQVisClienten*. Es zeigt welche Interaktionsmöglichkeiten diese mit dem JDQVisF haben und wie diese umgesetzt werden können.

Das JDQVisF ist ein WebService für Visualisierung von Datenqualitätswerten. Es bietet über seine WSDL-Datei seinen potentiellen Clients alle nötigen Informationen an.

6.5.1 Registrierung am JDQVisF

Damit ein Wissenschaftler seine Simulation mit Hilfe des JDQVisF überwachen kann benötigt er mindestens ein Client-Programm, das dessen WSDL-Datei interpretieren kann und die *registerUser*-Methode des *JDQVisController* aufruft.

Listing 14 zeigt den Aufbau einer *registerUser*-SOAP-Nachricht des Client für die Registrierung am JDQVisF.

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:registerUser xmlns:ns2="http://user.jdqvis.com/">
            <simulationId>exampleSimulation</simulationId>
            <username>exampleUsername</username>
            <password>123</password>
            <deviceId>default</deviceId>
        </ns2:registerUser>
    </S:Body>
</S:Envelope>
```

Listing 14: Aufbau einer registerUser-Message

/ registerUser
: Dieses Element beschreibt die Methode des *JDQVisController* für die Benutzerregistrierung.

/ registerUser / simulationId
: Dieses Element kennzeichnet die SimulationId für die sich der Wissenschaftler anmelden möchte. Der *JDQVisController* wählt anhand dieser SimulationId das passenden Authorizer-Plug-In aus.

/ registerUser / username
: Dieses Element ist der Benutzername, der durch das Authorizer-Plug-In validiert wird.

/ registerUser / password
: Dieses Element ist das Passwort, das durch das Authorizer-Plug-In validiert wird.

/ registerUser / deviceId
: Dieses Element beschreibt das Anzeigegerät für das die Visualisierungen generiert werden sollen. Dabei ist zu beachten, dass passende Visualizer-Plug-Ins beim JDQVisF registriert sind (siehe Kapitel 6.3.4). Hier muss nicht das aktuelle Gerät, das zur Registrierung am JDQVisF verwendet wird angegeben werden. Beispielsweise könnte ein Client-Programm nur für die Registrierung eingesetzt werden. Der Wissenschaftler könnte die Visualisierungen zu einem späteren Zeitpunkt über einen Internetbrowser betrachten.

Ist der Wissenschaftler erfolgreich am JDQVisController angemeldet, erhält er von diesem seine *UserId* als Antwort.

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:registerUserResponse xmlns:ns2="http://user.jdqvis.com/">
            <return>exampleUserId</return>
        </ns2:registerUserResponse>
    </S:Body>
</S:Envelope>
```

Listing 15: Antwort des JDQVisController bei erfolgreicher Anmeldung

6.5.2 Beschreibung der Benutzerinteraktionen

Ein Wissenschaftler kann mit Hilfe des JDQVisF zum einen die Visualisieren beeinflussen und zum anderen die laufende Simulation steuern. Dazu bietet das JDQVisF zwei Arten von Benutzeranfragen an. Zum einen die *VisualizationControlRequests* und *FilterControlRequests* zur Beeinflussung der Visualisierungs-Plug-Ins und zu anderen die *SimulationControlRequests* zur Steuerung der SimulationController-Plug-Ins. Die Trennung dieser Methoden wurde in den vorangegangenen Kapiteln argumentiert.

6.5.3 Beeinflussung der Generierungen einer Visualisierung

Der Wissenschaftler kann die Visualisierungs-Plug-Ins mit Hilfe von *VisualizationControlRequests* oder *FilterControlRequest* beeinflussen. Dazu bietet der JDQVisController in der WSDL-Datei die *modifyVisualizerSpecification()* und *modifyFilterSpecification()* Methoden an. Diese erhalten als Parameter die *UserId*, *SimulationId* und die gewünschte Modifikationen.

Der *JDQVisController* passt über diese Methoden die entsprechende *Filter-* oder *VisualizerSpecification* des Wissenschaftlers für die angelmeldete Simulation an. Somit wird

beim nächsten Visualisierungsaufruf die veränderte Visualisierungsspezifikation in den Plug-Ins verarbeitet. Damit ein *Filter*- oder *VisualizationRequest* von einem Plug-In umgesetzt werden kann, muss dieser auch implementiert sein. Das bedeutet insbesondere, dass es eine Kopplung zwischen Clientprogramm und den Visualisierungs-Plug-Ins gibt. Für jede im Clientprogramm mögliche Interaktion muss die entsprechenden Realisierung in den Plug-Ins existieren.

Abbildung 6-15 zeigt beispielhaft die Steuerung des Filter-Plug-Ins durch ein Client-Programm. Der rote Pfeil kennzeichnet die einzelnen Verarbeitungsschritte.

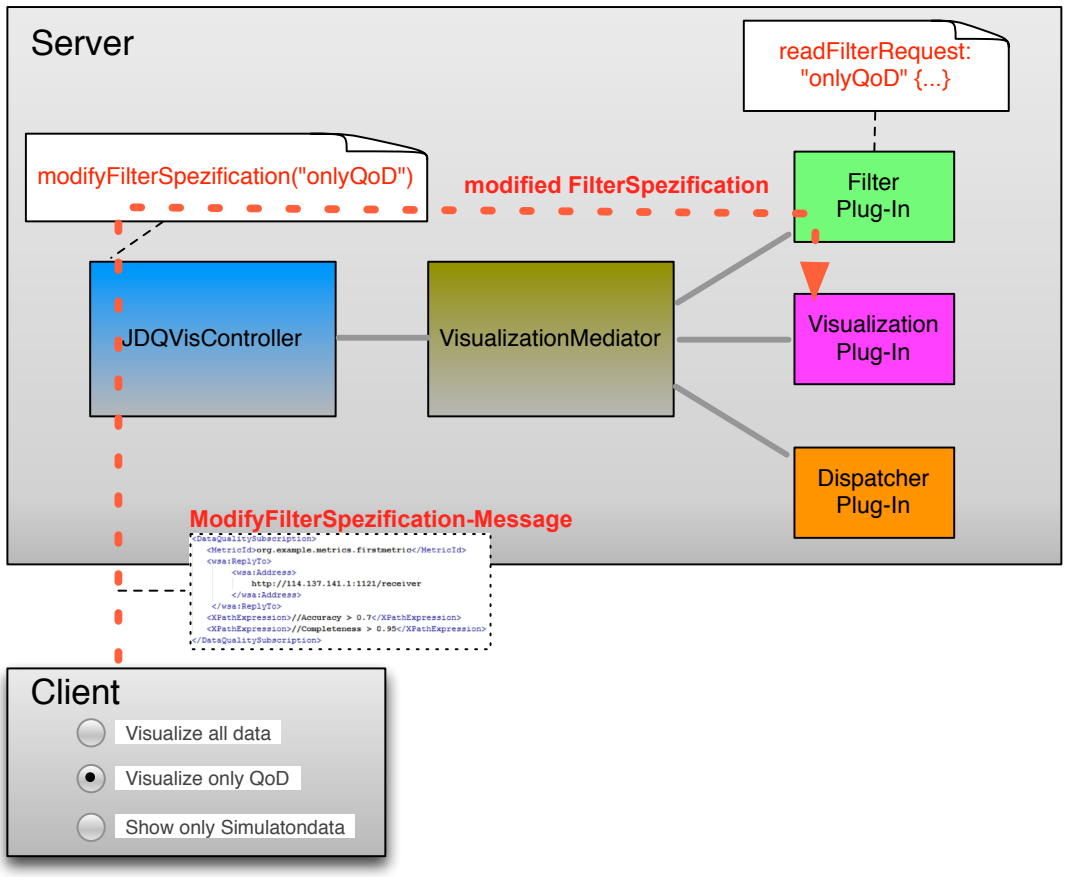


Abbildung 6-15: Steuerung des Filter-Plug-Ins durch ein Client-Programm. Der rote Pfeil markiert die Bearbeitung des FilterRequest

Durch die Registrierung der Plug-Ins für eine bestimmte Rolle wird zusätzlich sichergestellt, dass nur *VisualizationRequests* verarbeitet werden, für die der *JDQVisClient* auch die Rechte besitzt. Beispielsweise kann ein Wissenschaftler festlegen, dass er nur bei Unterschreitungen eines Schwellenwertes informiert wird. Ein Hilfswissenschaftler kann dagegen keine Auswahl der Daten vornehmen und muss dadurch die ihm zugeteilte Datenmenge auswerten.

Das JDQCF könnte über die Angabe von XPath-Ausdrücken ebenfalls so manipuliert werden, dass es nur bestimmt Daten als Eingabedaten an das JDQVisF sendet. Diese Form der Manipulation wird aber vom JDQVisF nicht unterstützt, da es zu Folge hätte,

dass alle *JDQVisClients* von der Änderung betroffen wären. Ist eine solche globale Manipulation den Wissenschaftlern gewünscht, so gibt es zwei Möglichkeiten dies Umzusetzen. Zum einen können entsprechende Filter-Plug-Ins für die betroffene Rolle und Simulation in das JDQVisF eingebunden werden. Zum anderen kann das JDQCF durch das Einbinden eines *ExternalTasks* dahingehend modifiziert werden [1].

Listing 16 zeigt den Aufbau einer *modifyFilterSpecification*-Nachricht in der ein *JDQVisClient* festlegt, dass er nur visualisierte Datenqualitätswerte empfangen möchte. Das Laden der Simulationsdaten innerhalb des Filter-Plug-Ins ist somit nicht nötig.

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:modifyFilterSpezification xmlns:ns2="http://user.jdqvis.com/">
            <userId>exampleUserId</userId>
            <simulationId>exampleSimulation</simulationId>
            <filterReqeust>OnlyQoD</filterReqeust>
        </ns2:modifyFilterSpezification>
    </S:Body>
</S:Envelope>
```

Listing 16: Aufbau einer modifyFilterSpecification-Nachricht

6.5.4 Steuerung der Simulation durch einen SimulationControlRequests

Das JDQVisF bietet den Wissenschaftlern über die *SimulationControlRequests* eine Möglichkeit in die laufende Simulation einzugreifen. Dazu bietet der *JDQVisController* in der WSDL-Datei die *sendSimulationControlRequest()*-Methode an. Diese erhält als Parameter die *UserId* und den Steuerbefehl.
Empfängt der *JDQVisController* über diese Methode einen Steuerungsbefehl, leitet er diesen an das entsprechende SimulationController-Plug-In weiter welches ihn verarbeiten kann.

7 Ausblick

Durch die vorgestellte flexible Architektur sind verschiedenste und breit gestreute Einsatzgebiete für das JDQVisF denkbar. In diesem Kapitel werden mögliche zukünftige Arbeiten im Zusammenhang mit dem JDQVisF vorgestellt.

7.1 Integration in die Simulation-Workflowumgebung

Durch die Integration in eine existierende Simulation-Workflow-Umgebung, könnte das JDQVisF den Wissenschaftlern helfen, eine laufende Simulation anhand visualisierter Datenqualitätswerte zu überwachen. Denkbar wäre, dass das JDQVisF zusammen mit dem JDQCF als gemeinsamer Datenqualitätsservice in den Workflow integriert wird. Dadurch könnten sie als eine Einheit für die Überwachung und Steuerung der Simulation eingesetzt werden.

Ein weitere Möglichkeit wäre das JDQVisF in die Service Discovery eines Enterprise Service Bus (ESB) zu integrieren. In einer parallel zu dieser Arbeit entstehenden Diplomarbeit wird ein Enterprise Service Bus entwickelt, welcher eine datenqualitätsgesteuerte Service Discovery auf Basis der WS-Policy Spezifikation ermöglicht. Durch spezielle Visualisierungen könnten die zum Teil komplexen Abhängigkeiten zwischen Datenqualitätsanforderungen und Datenqualitätszusicherungen bei der Service Discovery verständlich wiedergegeben werden. Insbesondere könnten durch die Integration des JDQVisF in den ESB auch dessen Entscheidungen bei der Serviceauswahl den beteiligten Wissenschaftler visuell präsentiert werden.
Eine weitere Anwendung könnte die Auswahl geeigneter Services auf Grundlage der visualisierten Datenqualitätswerte durch die Wissenschaftler sein.

7.2 Visualisierung weiterer Datenqualitäten

In dieser Arbeit wurde das Gebiet der Datenqualität in einem engen Rahmen durch die vorgestellten sechs Datenqualitätsdimensionen betrachtet. Bei komplexen FEM-basierten Simulationen, haben die Wissenschaftler jedoch zusätzlich Ansprüche an die Daten, wie beispielsweise an die *Matrixpopulation* oder die *Hauptdiagonale einer Matrix*. Um auch solche Datenqualitätsdimensionen durch das JDQVisF visualisieren zu könne, müssen entsprechende Plug-Ins entwickelt und in dieses eingebunden werden.

Für reine Datenqualitäten könnte zudem eine einheitliche Visualisierung durch Standarddiagramme in das JDQVisF eingebunden werden. Diese haben den Vorteil, dass sie der Gefahr einer möglichen Fehlinterpretation durch eine einheitliche Darstellung

entgegen wirken und dadurch eine gemeinsame Kommunikationsgrundlage für die unterschiedlichen Domänenspezialisten bilden können.

7.3 Entwicklung von domänen-spezifischen Plug-Ins

Die flexible Architektur des JDQVisF ermöglicht es, domänen-spezifische Visualisierungen zu generieren oder simulationsabhängige Autorisierungen und Steuerungen einzubinden. Speziell bei der Generierung der Visualisierungen können leicht neue Algorithmen eingebunden werden. Eine mögliche Erweiterung wäre, neben den reinen Datenqualitätswerten, eine Kombination aus diesen mit den originalen Simulationsdaten darzustellen. Dabei könnte eine kombinierte Visualisierung generiert werden, die dem Wissenschaftler direkt anzeigt in welchem Bereich seiner Simulationsdaten welche Datenqualität herrscht. Abbildung 7-1 zeigt ein mögliches Beispiel in der die Stellen rot markiert werden, bei denen die berechneten Simulationsdaten nicht mit den tatsächlichen Daten, die aus Lehrbüchern bekannt sind, übereinstimmt.

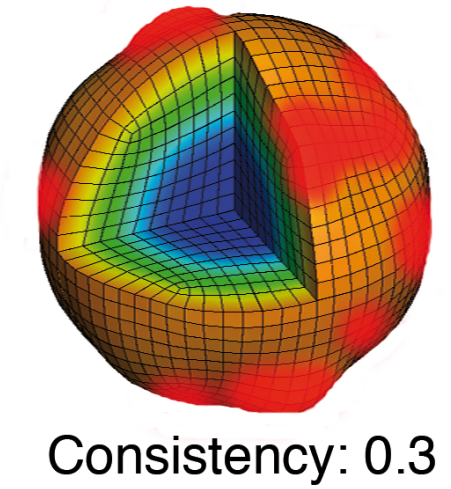


Abbildung 7-1: Beispiel: Datenqualitätsvisualisierungen innerhalb der Simulationsdaten

7.4 Integration eines WS-HumanTask Systems

Das JDQVisF visualisiert die zuvor berechneten und interpretierten Datenqualitätswerte. Soll die Datenqualität jedoch von einem Wissenschaftler bewertet werden, könnte das JDQVisF diesem nur zuvor berechneten Metrik-Ergebnisse visuell präsentieren. Diese können anschließend von dem Wissenschaftler interpretiert werden.
Für die Umsetzung einer solchen subjektiven Bewertung oder zur Steuerung der laufenden Simulation, könnte die Integration eines WS-HumanTask-Systems [38] in das JDQVisF erfolgen. Es erlaubt Menschen in einen Simulation-Workflow einzubinden und Aufgaben auszuführen. Mit Hilfe eines geeigneten Client-Programms, könnten Wissen-

schaftler so die visualisierten Simulationsdaten bewerten und mit Hilfe eines SimulationController-Plug-Ins und des JDQVisF an den Simulation-Workflow weiterleiten (siehe Abbildung 7-2).

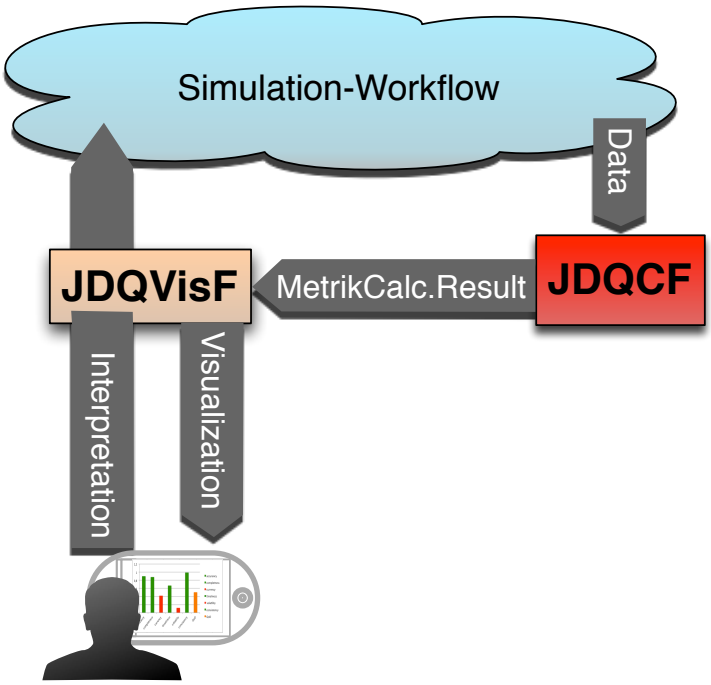


Abbildung 7-2: WS-HumanTask zur Bewertung der Datenqualität

Literaturverzeichnis

[1] U. Breitenbücher: Datenqualität in Simulations-Workflows, 2011

[2] C. Batini, M. Scannapieco: *Data Quality – Concepts, Methologies and Technics:* Spinger, 2006

[3] S. Schick, B. Theobald: Informationsvisualisierung im WM: Projekt Wissensmanagement Universität des Saarlands, 2005

[4] ITWissen – Home: http://www.itwissen.info/definition/lexikon/Daten-data.html, Abgerufen am 14. Mai 2012

[5] T. Hey, S. Tansley, K. Tolle, Herausgeber: *The Fourth Paradigm: Data-Intensive Scientific Discovery*: Microsoft Reserch, 2009

[6] International Association for Information and Data Quality – Home: http://iaidq.org/main/glossary.shtml#I, Abgerufen am 14. Mai 2012

[7] M. Reiter, H. Truong, S. Dustdar, D. Karastoyanova, R. Krause, F. Leymann, D. Pahret: On Analyzing Quality of Data Influences on Perfomance of Finite Elements driven Computional Simulations, 2012

[8] R. Y. Wang, D.M. Strong: Beyound Accurancy: What Data Quality Means to Data Consumer, 1996

[9] L. L. Pipino, Y. W. Lee, R. Y. Wang: Data Quality Assessment, 2002

[10] SKA – Home: www.ska.goc.au, Abgerufen am 15. Mai 2012

[11] CERN – Home: http://public.web.cern.ch/public/en/LHC/LHC-en.html, Abgerufen am 15. Mai 2012

[12] Pan-STARS – Home: http://pan-starrs.ifa.hawaii.edu/public , Abgerufen am 15. Mai 2012

[13] S. Hartmann: The World as a Process: Simulations in the Natural and Social Sciences, 2005.

[14] T. Schlegel: Graphical-Interactive Systems, Institut für Visualisierung, Universität Stuttgart, WS 09/10

[15] R. Däßler: Informationsvisualisierung – Stand, Kritik und Perspektiven, 1999

[16] J. Rang: Visualisierung wissenschaftlicher Daten, TU Braunschweig, 2006

[17] H. Schuhmann, W. Müller: Visualisierung – Grundlagen und allgemeine Methoden, 2000

[18] S, Weerawarana, F. Curbera, F. Leymann, T. Storey, D. F. Ferguson: *Web Services Platform Architecture*, 2005

[19] M. Burch: Informationsvisualisierung, Institut für Visualisierung und Interaktive Systeme, Universität Stuttgart, WS 11/12

[20] Prof. D. Jackèl, Dr.-Ing. B. Karstens, C. Becker: Vortragsseminar: Visuelle Wahrnehmung und 3D-Displays, Objektwahrnehmung und Gestaltgesetzte nach Wertheimer, Universität Rostock, WS00/01

[21] J. Bertin: Grafische Darstellung und die graphische Weiterverarbeitung der Information, 1982

[22] Leibniz-Institut für Länderkunde – Home: http://www.nationalatlas.de/deutscher-nationalatlas%20/kartographie/grundelemente-der-karte/

[23] L. Sijvesma: Colloquim Map Design, 2009

[24] Prof. A. Schmidt, Institut für Visualisierung und Interaktive Systeme, Universität Stuttgart, Persönliches Gespräch, 3. Juli 2012

[25] J. W. Seifert: Visualisieren, Präsentieren, Moderieren, 2001

[26] M. Reiter, Institut für Architektur von Anwendungssystemen, Universität Stuttgart, Persönliches Gespräch, 18. September 2012

[27] R. Sulo, S. Eick, R. Grossman: Davis: A Tool for Visualizing Data Quality, 2005

[28] M. Reiter, U. Breitenbücher, D. Karastoyanova, O. Kopp: Quality Driven Simulation-Workflows, 2012

[29] W3C- Home: http://www.w3.org/TR/ws-arch/#whatis, Abgerufen am 29. September 2012

[30] K. Görlach, M. Sonntag, D. Karastoyanova, F. Leymann, M. Reiter: Conventional Workflow Technology for Scientific Simulations, 2011

[31] H. Motahari-Nezad, B. Stephanson, S. Singhai: Outsourcing Business to Cloud Computing, 2009

[32] A. S. Vedamuthu, D. Orchard, F. Hirsch, M. Hondo, P. Yendluri, T. Boubez: Web services policy 1.5-framework: W3C Recommendation, 2007

[33] F. Leymann: Web Services, Institut für Architektur von Anwendungssystemen, Universität Stuttgart, WS 11/12

[34] W3C – Home: http://www.w3.org/XML, Abgerufen am 4. Oktober 2012

[35] Portable Network Graphic – Home: http://www.libpng.org/pub/png/, Abgerufen am 3. Oktober 2012

[36] OnlyOpenSource – Home: http://www.only-open-source.com/dokus/3d-visualisierungsformate.html#vrml, Abgerufen am 24. Oktober 2012

[37] Wörterbuchsubstanz aus: Duden – Wissensnetz deutsche Sprache, 2011

[38] A. Agrawal, M. Amend, M. Kloppmann, D. König, F. Leymann, et al.: Web Services Human Task (WS-HumanTask), Version 1.0, 2007.

[39] W. Bils: *Warum das Auge sehen kann*, Quelle & Meyer, 2010

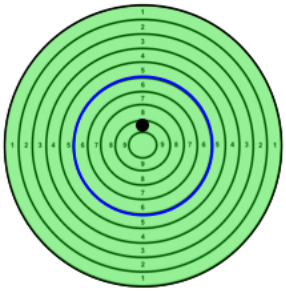
[40] H. L. Truong, S. Dustar: On Evaluating an Publishing Data Concerns for Data as a Service, Distributed System Group, Vienna University of Technology

[41] E. Gamma, R. Helm, R. E. Johnson, J. Vlissides: *Design Patterns. Elements of Reusable Object-Oriented Software*, 1994

[42] M. Kate Beard: NCGIA Research Initiative 7 Visualization of Spatial Data Quality, University of Maine, 1991

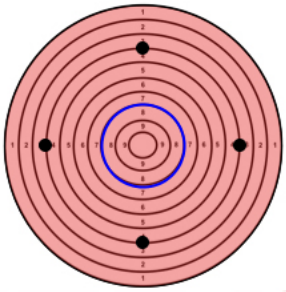
8 Appendix A – Beispielvisualisierung Datenqualität

Accuracy



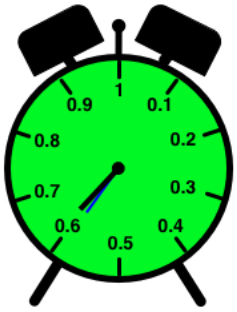
0.85 | 0.5

Consistency



0.3 | 0.7

Timeliness

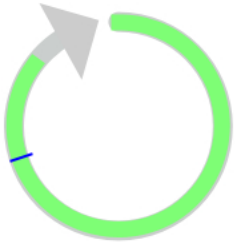


0.62 | 0.6

Completeness

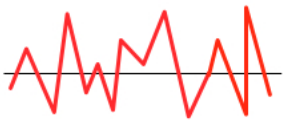
0.77 | 0.6

Currency



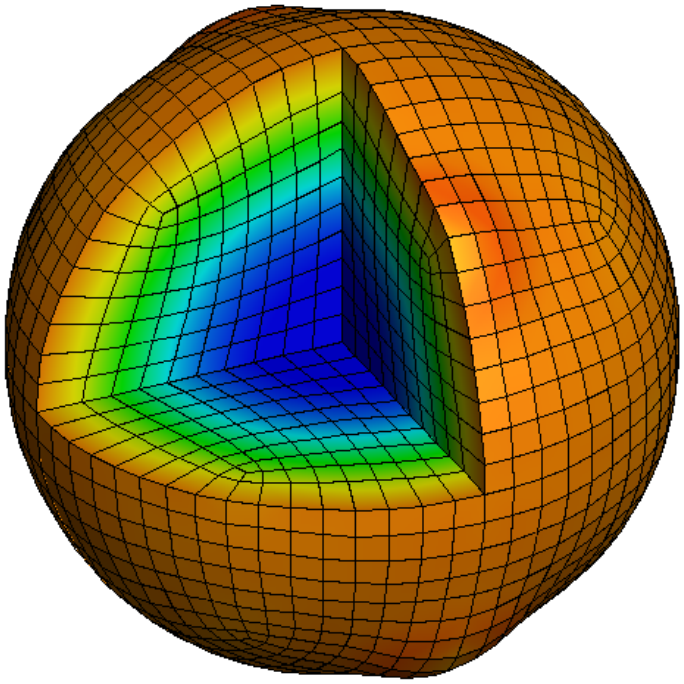
0.85 | 0.7

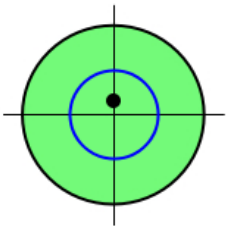
Volatility



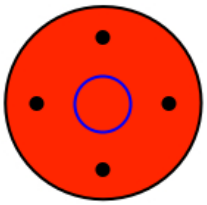
0.2 | 0.6

9 Appendix B – Beispielvisualisierung Datenqualität mit originalen Simulationsdaten





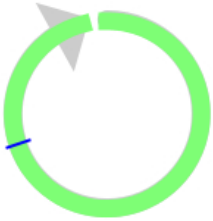
0.85 | 0.5



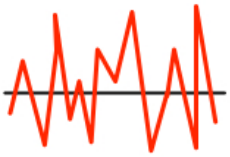
0.3 | 0.7

0.62 | 0.6

0.77 | 0.6



0.96 | 0.7



0.2 | 0.6

Erklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie alle wörtlich oder sinngemäß übernommenen Stellen in der Arbeit gekennzeichnet habe. Die Arbeit ist in gleicher oder ähnlicher Form noch nicht als Prüfungsarbeit eingereicht worden.

_____

(Marcel Russ)