ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

Bachelor-Thesis

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)
an der Hochschule für Technik und Wirtschaft des Saarlandes
im Studiengang Praktische Informatik
der Fakultät für Ingenieurwissenschaften

Eine LaTeX-Vorlage für Abschlussarbeiten im Bereich Informatik/Mechatronik-Sensortechnik (inkl. DFHI) an der htw saar

vorgelegt von
Max Muster

betreut und begutachtet von
Prof. Dr.-Ing. André Miede
Prof. Dr. Thomas Kretschmer

Saarbrücken, Tag. Monat Jahr

Selbständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit (bei einer Gruppenarbeit: den entsprechend gekennzeichneten Anteil der Arbeit) selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich erkläre hiermit weiterhin, dass die vorgelegte Arbeit zuvor weder von mir noch von einer anderen Person an dieser oder einer anderen Hochschule eingereicht wurde.

Darüber hinaus ist mir bekannt, dass die Unrichtigkeit dieser Erklärung eine Benotung der Arbeit mit der Note „nicht ausreichend“ zur Folge hat und einen Ausschluss von der Erbringung weiterer Prüfungsleistungen zur Folge haben kann.

Saarbrücken, Tag. Monat Jahr

Max Muster

Zusammenfassung

Kurze Zusammenfassung des Inhaltes in deutscher Sprache, der Umfang beträgt zwischen einer halben und einer ganzen DIN A4-Seite.

Orientieren Sie sich bei der Aufteilung bzw. dem Inhalt Ihrer Zusammenfassung an Kent Becks Artikel: `http://plg.uwaterloo.ca/~migod/research/beckOOPSLA.html`.

Inhaltsverzeichnis

1 Einleitung

In diesem Kapitel geben wir eine Einführung in die Ausarbeitung unserer Fallstudie. Wir erläutern unsere Motivation, warum wir uns für das Thema Lambda-Architektur entschieden haben, und stellen unsere Idee für die Analyse eines Parkhaussystems vor, das als Anwendungsfall für die Umsetzung der Lambda-Architektur dient. Anschließend werden wir die konkreten Ziele, die wir mit dieser Fallstudie erreichen wollen, sowie unser weiteres Vorgehen bei der Implementierung und Evaluierung definieren.

1.1 Motivation

Heutzutage werden immer größere Datenmengen verarbeitet. Moderne Anwendungen müssen daher in der Lage sein, große Datenströme effizient zu verarbeiten, zu speichern und bereitzustellen, ohne dabei Verlust an Leistung oder Zuverlässigkeit zu riskieren.

Ein zentrales Problem bei der Verarbeitung großer Datenmengen wird durch das CAP-Theorem beschrieben. Dieses Theorem besagt, dass ein verteiltes Datensystem immer drei grundlegende Eigenschaften berücksichtigen muss:

- **Consistency (Konsistenz)**: Jeder Lesevorgang liefert entweder den aktuellsten verfügbaren Wert oder schlägt fehl.
- **Availability (Verfügbarkeit)**: Jede Anfrage erhält garantiert eine Antwort, auch wenn einige Knoten des Systems ausgefallen sind.
- **Partition Tolerance (Partitionstoleranz)**: Das System funktioniert weiter, auch wenn Teile des Netzes ausfallen oder Kommunikationsprobleme auftreten.

Allerdings stellt das CAP-Theorem eine grundlegende Einschränkung dar: Ein System kann immer nur zwei dieser drei Eigenschaften gleichzeitig garantieren. In datenintensiven Anwendungen, die eine hohe Fehlertoleranz (Partition Tolerance) und Verfügbarkeit (Availability) erfordern, kann daher nicht sichergestellt werden, dass jede Abfrage immer die aktuellsten Daten in einer akzeptablen Zeit liefert. Dies stellt eine große Herausforderung für Systeme dar, die sowohl Echtzeitdatenverarbeitung als auch Langzeitanalysen ermöglichen sollen.

Im Rahmen der Vorlesung „Software-Architektur“ haben wir uns entschieden, eine Fallstudie zur Lambda-Architektur zu erstellen. Die Lambda-Architektur ist ein Architekturmuster, das speziell für Big-Data-Anwendungen entwickelt wurde, also für Systeme, die große Datenmengen effizient verarbeiten müssen. Ihr Ziel ist es, eine fehlertolerante und hochverfügbare Architektur bereitzustellen, die trotz der Einschränkungen des CAP-Theorems eine möglichst hohe Datenkonsistenz gewährleistet. Sie begegnet dem Problem des CAP-Theorems, indem sie zwei Verarbeitungswege kombiniert: den Batch Layer für konsistente Langzeitanalysen und den Speed Layer für schnelle Echtzeitabfragen.

In dieser Fallstudie wollen wir diese Eigenschaften der Lambda-Architektur analysieren und bewerten. Dazu simulieren wir einen realen Anwendungsfall, in dem sowohl die Langzeitanalyse großer Datenmengen als auch die schnelle Bereitstellung neuer Daten eine zentrale Rolle spielen.

Wir orientieren uns dabei an bestehenden Anwendungsgebieten der Lambda-Architektur. Diese Architektur ist insbesondere in Web- und IoT-Anwendungen weit verbreitet, da sie eine skalierbare Verarbeitung großer Datenmengen bei gleichzeitig schneller Verfügbarkeit aktueller Daten ermöglicht.

Für unsere konkrete Fallstudie haben wir uns für eine IoT-Anwendung im Bereich der Parkhausanalyse entschieden. Ziel ist es, ein simuliertes Parkhaussystem zu entwickeln, das mithilfe von Kameras die Ein- und Ausfahrten in mehreren Parkhäusern erfasst und analysiert. Dabei sollen fahrzeugspezifische Informationen gespeichert und verarbeitet werden. Dieses Szenario eignet sich aus unserer Sicht besonders gut für die Lambda-Architektur, da es sowohl Langzeitanalysen (z.B. Belegungsmuster über mehrere Wochen) als auch Echtzeitabfragen (z.B. aktuelle Parkplatzverfügbarkeit) kombiniert.

1.2 Forschungsfragen

Mit dem Ziel unserer Fallstudie vor Augen können wir die folgenden Forschungsfragen definieren, die wir durch unsere Implementierung der Lambda-Architektur beantworten möchten:

- **RQ1:** Inwiefern lässt sich eine einfache, aber funktionsfähige Lambda-Architektur in einer kleinen, simulierten Umgebung implementieren?
- **RQ2:** Welche architektonischen Eigenschaften der Lambda-Architektur lassen sich in unserer Fallstudie identifizieren?
- **RQ3:** Welche Herausforderungen treten bei der Implementierung der Lambda-Architektur in einer kleinen, nicht skalierten Umgebung auf?

1.3 Forschungsansatz

Unser Ziel in dieser Fallstudie ist es, eine einfache, aber funktionsfähige Implementierung der Lambda-Architektur in einer simulierten Umgebung zu entwickeln. Der Fokus liegt dabei nicht auf der Entwicklung einer produktionsreifen Lösung für ein reales Parkhaussystem, sondern vielmehr auf der Schaffung eines praxisnahen Szenarios, das hilft, die Architektur besser zu verstehen und ihre Vor- und Nachteile zu evaluieren.

Um dieses Ziel zu erreichen, entwickeln wir eine simulierte Datenquelle, die künstliche Fahrzeugdaten generiert. Diese Daten durchlaufen die verschiedenen Schichten der Lambda-Architektur, so dass sowohl Echtzeit- als auch Batch-Verarbeitung getestet werden können.

Um die erste Forschungsfrage (RQ1) zu beantworten, haben wir uns für eine Implementierung auf Basis bewährter Open-Source-Technologien entschieden. Anstatt eine eigene Architektur von Grund auf neu zu entwickeln, verwenden wir existierende Werkzeuge, die bereits in der Praxis für die Lambda-Architektur eingesetzt werden. Die Auswahl basiert auf einer Literaturrecherche zu typischen Technologien für die drei Schichten der Lambda-Architektur. Eine detaillierte Erläuterung der Komponenten und Funktionsweisen der Lambda-Architektur erfolgt in Kapitel 2.

Der Speed Layer der Lambda-Architektur ist für die Echtzeitverarbeitung zuständig. Im Speed Layer müssen die Daten schnell verarbeitet werden. Für die Implementierung der Speed Layer wurde daher Apache Kafka gewählt. Kafka ermöglicht eine schnelle und zuverlässige Verarbeitung und Weiterleitung von Datenströmen.

Der Serving Layer übernimmt die Aggregation und Bereitstellung der verarbeiteten Daten. Er verbindet den Speed- und den Batch-Layer und bietet eine zusammenführende

Sicht auf die gespeicherten Informationen. Wir haben uns entschieden, den Serving Layer mit Apache Cassandra zu realisieren. Cassandra ist eine NoSQL-Datenbank, die speziell für schnelle Abfragen optimiert ist und in vielen realen Implementierungen der Lambda-Architektur für den Serving Layer verwendet wird.

Im Batch Layer werden alle historischen Rohdaten gespeichert und verarbeitet. In der klassischen Lambda-Architektur werden hier häufig verteilte Systeme wie Apache Hadoop oder Apache Spark eingesetzt, da diese für den Umgang mit großen Datenmengen optimiert sind. Da in unserer Fallstudie jedoch nicht mit so großen Datenmengen gearbeitet wird, verzichten wir auf eine skalierbare, verteilte Speicherung. Stattdessen verwenden wir auch hier Apache Cassandra, da es sowohl für schnelle Lesezugriffe als auch für die Langzeitspeicherung großer Datenmengen optimiert ist.

Die Anwendung wird in C# entwickelt, da diese Sprache eine gute Integration mit Apache Kafka und Apache Cassandra bietet und sich gut für die Implementierung unserer simulierten Umgebung eignet. Für die Verwaltung und Bereitstellung der verschiedenen Komponenten verwenden wir Docker. Diese Container-Technologie ermöglicht eine einfache Installation, Verwaltung und Reproduzierbarkeit unserer Umgebung, da alle benötigten Dienste in isolierten Containern laufen und unabhängig von der Konfiguration des Host-Systems sind.

2 Verwandte Arbeiten

In diesem Kapitel geben wir einen Überblick über bestehende Arbeiten zur Lambda-Architektur und deren Einfluss auf die Entscheidungen, die wir bei der Entwicklung unserer Fallstudie getroffen haben. Zunächst erläutern wir die grundlegende Struktur der Lambda-Architektur, ihre Kernkomponenten und ihre Ziele. Anschließend betrachten wir existierende Implementierungen, die als Inspiration für unser eigenes Fallstudienszenario dienten.

Darauf aufbauend analysieren wir die architektonischen Eigenschaften der Lambda-Architektur sowie ihre bekannten Vor- und Nachteile, um eine Grundlage für die spätere Bewertung unserer Fallstudie zu schaffen. Abschließend vergleichen wir die Lambda-Architektur mit der Kappa-Architektur, einer alternativen Architektur, die in bestimmten Szenarien die Schwächen der Lambda-Architektur umgehen soll.

2.1 Überblick über die Lambda-Architektur

Die Lambda-Architektur wurde von Nathan Marz als Antwort auf die wachsenden Herausforderungen bei der Verarbeitung großer Datenmengen im Zeitalter von Big Data entwickelt. Sie entstand aus der Erkenntnis, dass herkömmliche Architekturen oft zu komplex, fehleranfällig und schwer skalierbar sind. In vielen Systemen kann bereits ein einziger Datenfehler oder der Ausfall einer Komponente dazu führen, dass das gesamte System inkonsistent wird, Daten verloren gehen oder verfälscht werden und das System nicht mehr zuverlässig arbeitet.

Ein weiteres Problem klassischer Architekturen ist, dass sie Echtzeit- und Batch-Verarbeitung nicht effizient kombinieren können. Bei Big-Data-Anwendungen muss das System nicht nur jederzeit auf die Daten zugreifen und auf Anfragen reagieren können, sondern auch sicherstellen, dass die gelieferten Informationen stets aktuell sind. Dies würde bedeuten, dass bei jeder einzelnen Anfrage alle gespeicherten Daten vollständig durchsucht und verarbeitet werden müssten, um die aktuellsten Werte zu gewährleisten. Bei wachsenden Datenmengen wird dies jedoch schnell ineffizient und führt zu hohen Latenzzeiten und Systembelastungen.

Um diese Probleme zu lösen, verfolgt die Lambda-Architektur einen Ansatz, der große Datenmengen effizient verarbeitet und gleichzeitig Fehlertoleranz, Skalierbarkeit und Konsistenz gewährleistet. Ein zentrales Konzept ist dabei die unveränderliche Speicherung von Daten (Immutable Data). Dadurch können historische Daten jederzeit erneut verarbeitet werden, ohne dass das System inkonsistent wird.

Das Hauptziel der Lambda-Architektur ist es, Echtzeit- und Batch-Verarbeitung strukturell zu trennen, um die Vorteile beider Methoden zu kombinieren. Zu diesem Zweck ist die Architektur in drei Schichten unterteilt (siehe Abbildung 1):

- **Batch Layer:** Speichert alle Rohdaten unverändert und dient als Langzeitarchiv. Ein wesentlicher Aspekt dieser Schicht ist, dass die gespeicherten Daten nicht verändert werden, sondern ausschließlich für regelmäßige Langzeitanalysen verwendet werden.

- **Speed Layer:** Verarbeitet Datenströme in Echtzeit, um sofortige Ergebnisse zu liefern. Da der Batch-Layer aufgrund seiner Struktur nicht für schnelle Abfragen geeignet ist, übernimmt der Speed-Layer die Aufgabe, aktuelle Daten schnell zur Verfügung zu stellen.
- **Serving Layer:** Verbindet die Batch- und die Speed-Layer und stellt die Daten für Benutzeranfragen bereit. Kombiniert die Ergebnisse der beiden Schichten und ermöglicht so eine konsistente und effiziente Datenbereitstellung.

Durch diese klare Trennung der Verarbeitungsebenen ermöglicht die Lambda-Architektur ein Gleichgewicht zwischen geringer Latenz und hoher Genauigkeit. Während die Batch Layer eine robuste und fehlertolerante Verarbeitung großer Datenmengen gewährleistet, wäre es zu zeitaufwändig, diese Schicht bei jeder Anfrage vollständig zu durchlaufen, um aktuelle Daten bereitzustellen. Hier kommt der Speed Layer ins Spiel: Er verarbeitet neue Daten in Echtzeit und stellt sie sofort zur Verfügung, um eine schnelle Reaktionszeit zu gewährleisten.

2.2 Anwendungsfälle der Lambda-Architektur

Da traditionelle Systeme wie relationale Datenbanken und klassische Batch-Processing-Systeme zunehmend an ihre Grenzen stoßen, wenn es darum geht, die stetig wachsenden Datenmengen effizient zu verarbeiten, gewinnt die Lambda-Architektur für moderne Anwendungen immer mehr an Bedeutung. Insbesondere im Zeitalter des Internets, in dem kontinuierlich riesige Datenmengen generiert, übertragen und analysiert werden, hat sich die Lambda-Architektur als leistungsfähige Lösung etabliert.

Yuvraj Kumar (2020) beschreibt in seinem Paper eine Vielzahl von modernen Anwendungsfällen der Lambda-Architektur sowie die Technologien, die häufig für deren Implementierung verwendet werden. Große Technologieunternehmen wie Twitter, LinkedIn, Netflix und Amazon nutzen die Lambda-Architektur, um ihre enormen Datenmengen effizient zu analysieren. Vor allem im Bereich der personalisierten Werbung spielt sie eine zentrale Rolle: Historische Kundendaten werden mit aktuellen Nutzerinteraktionen kombiniert, um in Echtzeit maßgeschneiderte Werbung auszuspielen.

Auch im Finanzsektor nutzen Unternehmen die Lambda-Architektur, um historische Transaktionsdaten mit Echtzeitanalysen zu verknüpfen. So können verdächtige Muster identifiziert und Betrugsversuche frühzeitig erkannt werden. Auch im Internet of Things (IoT) findet die Lambda-Architektur breite Anwendung. So wird sie beispielsweise in Smart Cities eingesetzt, um Logistikprozesse zu optimieren oder die Abfallentsorgung effizienter zu gestalten.

Das Paper von Kiran et al. zeigt, dass insbesondere Smart City Anwendungen stark von der Lambda-Architektur profitieren, da hier große Mengen an Sensordaten kontinuierlich erfasst und analysiert werden müssen. Die Autoren beschreiben eine konkrete Implementierung der Lambda-Architektur zur Verarbeitung und Analyse von Sensordaten in einer Netzwerkumgebung.

Als praktischen Anwendungsfall untersuchen Kiran et al. die Verarbeitung von Netzwerk- und Sensordaten aus dem Energy Sciences Network (ESnet). Ziel der Implementierung ist die Erkennung von Anomalien und die Optimierung der Netzwerkauslastung. Die Lamda-Architektur ist für diese Anwendung besonders geeignet, da sie die gleichzeitige Verarbeitung historischer und aktueller Netzdaten ermöglicht. Während im Batch-Layer Langzeitanalysen durchgeführt werden, um wiederkehrende Muster zu identifizieren, kann der Speed-Layer Anomalien in Echtzeit erkennen und darauf reagieren.

Die in der Literatur beschriebenen Anwendungsfälle, insbesondere die Arbeiten von Kiran et al. und Kumar, haben uns geholfen, ein geeignetes Szenario für unsere Fallstudie zu identifizieren. Die Forschung zeigt, dass die Lambda-Architektur besonders häufig in Smart City-Anwendungen eingesetzt wird, da sie die Echtzeitverarbeitung großer Sensordatenströme mit Langzeitanalysen kombiniert.

Darauf aufbauend wollten wir für unsere Fallstudie eine Anwendung im Smart City Kontext wählen, die ebenfalls Echtzeitanalysen mit Langzeitprognosen kombiniert. Dies führte uns schließlich zur Parkhausanalyse. Ähnlich wie in der Arbeit von Kiran et al. ist es in unserem Szenario sinnvoll, historische Daten zur Vorhersage der Parkhausauslastung mit Echtzeitinformationen über die aktuelle Belegung zu verknüpfen.

2.3 Technologien der Lambda-Architektur

Der vorherige Abschnitt hat gezeigt, dass die Lambda-Architektur eine weit verbreitete Big-Data-Architektur ist, die bereits in einer Vielzahl von Anwendungen implementiert wurde. In der Literatur werden verschiedene Technologien beschrieben, die zur Implementierung der drei Schichten der Lambda-Architektur verwendet werden können.

Ein konkretes Beispiel für die technologische Umsetzung liefert das Paper von Kiran et al. (2015), das eine Cloud-basierte Implementierung der Lambda-Architektur für die Sensordatenverarbeitung beschreibt. Die Autoren verwenden unterschiedliche Technologien für den Batch Layer, den Speed Layer und den Serving Layer.

Für den Batch Layer verwenden Kiran et al. das Apache Hadoop Distributed File System (HDFS). HDFS ist eine der am häufigsten verwendeten Technologien für die Lambda-Architektur, da es ein skalierbares, fehlertolerantes und verteiltes Dateisystem darstellt, das speziell für die Speicherung großer Datenmengen entwickelt wurde.

Aufgrund seiner Architektur ist HDFS jedoch relativ langsam bei Lese- und Schreibzugriffen, weshalb es hauptsächlich für Batch-Processing-Aufgaben verwendet wird. In der Implementierung von Kiran et al. wird HDFS verwendet, um historische Rohsensordaten von Netzwerkroutern unverändert zu archivieren. Da Lesezugriffe auf HDFS relativ langsam sind, können periodische Batch-Analysen sehr zeitaufwändig sein.

Für die Speed Layer verwenden Kiran et al. Apache Spark Streaming. Apache Spark ist eine vielseitige Technologie, die sowohl für Batch Processing als auch für Stream Processing eingesetzt werden kann. Spark Streaming ist eine Erweiterung von Apache Spark und ermöglicht die Verarbeitung von Datenströmen in Echtzeit, indem eingehende Daten kontinuierlich analysiert und aggregiert werden. Der Vorteil von Spark Streaming gegenüber klassischen Batch-Technologien ist, dass es Echtzeitdatenverarbeitung mit geringer Latenz ermöglicht, ohne dass die Daten vorher vollständig gespeichert werden müssen.

Für den Serving Layer verwenden Kiran et al. Amazon S3, um die Sensordaten aus dem Batch Layer zu aggregieren, zu berechnen und für historische und Echtzeit-Abfragen bereitzustellen. Die Wahl von Amazon S3 basiert darauf, dass die Autoren eine Cloud-basierte Implementierung der Lambda-Architektur entwickelt haben, die auf skalierbare Speicherdienste angewiesen ist.

Im Gegensatz dazu planen wir in unserer Fallstudie eine lokale Implementierung der Lambda-Architektur, weshalb alternative Technologien für die Speicherung und Aggregation im Serving Layer in Betracht gezogen werden müssen.

In der Literatur werden zahlreiche weitere Technologien beschrieben, die für die verschiedenen Schichten der Lambda-Architektur eingesetzt werden können. Besonders hervorzuheben ist die Arbeit von Kumar (2020), der mehrere alternative Technologien diskutiert, die auch für unsere Fallstudie vielversprechend sind.

Laut Kumar (2020) ist eine der am häufigsten verwendeten Technologien für die Lambda-Architektur Apache Kafka. Kafka ist ein verteiltes Publish-Subscribe-Messaging-System, das für Echtzeit-Datenströme optimiert ist. Es ermöglicht das Publizieren neuer Daten in Echtzeit in eine Queue. Von dort aus können die Daten sowohl an die Batch-Layer zur langfristigen Speicherung als auch an die Speed-Layer zur sofortigen Verarbeitung weitergeleitet werden. Damit ist sichergestellt, dass Echtzeit- und Langzeitverarbeitung gleichzeitig erfolgen können.

Eine weitere Technologie, die laut Kumar (2020) häufig in der Lambda-Architektur eingesetzt wird, ist Apache Cassandra. Cassandra ist eine hochskalierbare NoSQL-Datenbank, die sich besonders für den Serving Layer eignet. Im Serving Layer werden Echtzeitabfragen aus dem Speed Layer und aggregierte Batch Views aus dem Batch Layer zusammengeführt und gespeichert. Cassandra zeichnet sich durch extrem schnelle Lese- und Schreibzugriffe aus, was für die kontinuierliche Aktualisierung der Daten im Serving Layer wichtig ist.

3 Konzept

4 Implementierung

5 Evaluierung

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

Anhang

A Erster Abschnitt des Anhangs

In den Anhang gehören „Hintergrundinformationen“, also weiterführende Information, ausführliche Listings, Graphen, Diagramme oder Tabellen, die den Haupttext mit detaillierten Informationen ergänzen.

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln. Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln. Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

Kolophon

Dieses Dokument wurde mit der LaTeX-Vorlage für Abschlussarbeiten an der htw saar im Bereich Informatik/Mechatronik-Sensortechnik erstellt (Version 2.25, August 2024). Die Vorlage wurde von Yves Hary und André Miede entwickelt (mit freundlicher Unterstützung von Thomas Kretschmer, Helmut G. Folz und Martina Lehser). Daten: (F)10.95 – (B)426.79135pt – (H)688.5567pt