T1000

Jan Herrmann

31. Juli 2025

Inhaltsverzeichnis

1 Einleitung

1.1 Hintergrund

In der modernen Softwareentwicklung gewinnt die Wiederverwendbarkeit und Modularität von Programmbestandteilen zunehmend an Bedeutung. Gerade in größeren oder langfristig gepflegten Projekten ist es entscheidend, wiederkehrende Aufgaben wie den Datenbankzugriff klar zu kapseln, um eine saubere Trennung von Logik und Infrastruktur zu gewährleisten. Der Aufbau einer eigenen Verbindungsbibliothek in Form einer DLL ermöglicht es, wiederkehrende Abläufe wie Verbindungsaufbau, Fehlerbehandlung und SQL-Abfragen strukturiert und einheitlich bereitzustellen. Dies erhöht nicht nur die Wartbarkeit, sondern auch die Testbarkeit des Codes, insbesondere im Zusammenhang mit automatisierten Unit- und Integrationstests.

Im Kontext des dualen Studiums wird darüber hinaus ein praktischer Einblick in die professionelle Softwareentwicklung vermittelt. Die Fähigkeit, technische Anforderungen eigenständig zu analysieren, in objektorientierte Konzepte zu überführen und mit aktuellen Tools wie Docker, Logging-Frameworks oder Dependency Injection umzusetzen, ist für die spätere berufliche Praxis im industriellen Umfeld essenziell. Die Relevanz des Themas liegt somit sowohl in der technischen als auch in der didaktischen Bedeutung. Es verbindet theoretisches Wissen mit einer konkreten, anwendungsnahen Lösung. [**broy_Grundlagen_2019**]

1.2 Kontextualisierung der Thematik

In vielen Softwareprojekten spielt der Zugriff auf relationale Datenbanken eine zentrale Rolle. Dabei ist es üblich, dass Anwendungen Daten persistent speichern, abrufen und verändern müssen - etwa zur Benutzerverwaltung, zur Protokollierung von Abläufen oder zur Kommunikation mit anderen Systemen. Gerade in größeren Softwarearchitekturen ist es jedoch problematisch, wenn Datenbankzugriffe direkt und unstrukturiert in die Anwendungslogik eingebettet werden. Dies führt häufig zu unübersichtlichem Code, erschwerten Tests und hoher Abhängigkeit von der konkreten Datenbankimplementierung.

Aus diesem Grund hat sich in der professionellen Softwareentwicklung das Prinzip der Schichtentrennung etabliert: Der Datenbankzugriff wird in einer eigenen Komponente gekapselt, die unabhängig von der übrigen Logik funktioniert und über wohldefinierte Schnittstellen angesprochen wird. Durch die Kapselung der Datenbankinteraktion lässt sich die Wartbarkeit und Wiederverwendbarkeit des Codes erheblich verbessern - insbesondere in komplexen Systemen oder bei langfristiger Weiterentwicklung.

Ein häufig gewählter technischer Ansatz zur Umsetzung dieser Kapselung ist die Entwicklung einer DLL. Sie stellt wiederverwendbare Funktionalität bereit, die in beliebige Anwendungen eingebunden werden kann - etwa für das Öffnen und Schließen von Verbindungen, die Ausführung von SQL-Befehlen oder das zentrale proto-

kollieren von Fehlern. In Verbindung mit Konzepten wie Enums zur Fehlercodierung, Interfaces zur Flexibilisierung oder automatisierten Tests mit Docker kann dadurch eine moderne und praxistaugliche Architektur entstehen, wie sie auch in der Industrie eingesetzt wird.

1.3 Zielsetzung und Projektbeschreibung

Ziel dieser Arbeit ist die Entwicklung einer wiederverwendbaren Verbindungsbibliothek in Form einer DLL für den Zugriff auf eine MySQL-Datenbank mit der Programmiersprache C#. Die Bibliothek soll grundlegende Funktionen wie den Aufbau und die Schließung von Datenbankverbindungen, die Ausführung von SQL-Abfragen (SELECT, INSERT, UPDATE, DELETE) sowie eine strukturierte Fehlerbehandlung kapseln. Besonderer Wert wird dabei auf eine saubere Trennung zwischen Anwendungslogik und Datenbankzugriff gelegt, um die Wartbarkeit, Wiederverwendbarkeit und Testbarkeit des Codes zu verbessern.

Das Projekt wird im Rahmen des dualen Studiums an der DHBW Stuttgart umgesetzt und verbindet theoretische Inhalte aus der Hochschulausbildung mit praxisorientierter Softwareentwicklung. Die entwickelte DLL soll so konzipiert sein, dass sie einfach in andere Anwendungen eingebunden werden kann und über eine einheitliche Schnittstelle angesprochen wird. Ergänzend werden Unit- und Integrationstests durchgeführt, unter anderem mit Docker-Containern, um die Funktionalität in realitätsnahen Szenarien sicherzustellen.

2 Grundlagen der Softwareentwicklung

2.1 Objektorientierte Programmierung

Die objektorientierte Programmierung (OOP) ist ein zentrales Konzept in der modernen Softwareentwicklung. Sie beruht auf der Idee, Programme aus modularen Einheiten, sogenannten Objekten, aufzubauen. Diese Objekte basieren auf Klassen, die als Baupläne dienen.

Klassen definieren die Attribute und die Methoden, welche ihre Objekte besitzen. Eine Methode ist dabei eine Funktion innerhalb der Klasse, die bestimmte Aktionen ausführt und den Zustand des Objekts verändern kann. Ein wesentliches Merkmal der OOP ist die Vererbung. Sie erlaubt es, dass Klassen die Attribute und Methoden anderer Klassen übernehmen, wodurch Code wiederverwendet und strukturiert erweitert werden kann. Interfaces spielen ebenfalls eine wichtige Rolle, indem sie die Struktur und das Verhalten festlegen, das eine Klasse bereitstellen muss, ohne konkrete Implementierungen vorzugeben.

Fehlerbehandlung erfolgt in der OOP häufig über sogenannte Exceptions. Statt Fehlercodes zurückzugeben, werfen Methoden bei Problemen Ausnahmen, die dann ge-

zielt behandelt werden können. Eine weitere hilfreiche Spracheigenschaft sind Enumerationen (Enums), die eine fest definierte Menge von Werten beschreiben, etwa Statuscodes und so die Lesbarkeit und Wartbarkeit des Codes erhöhen.

[**broy_grundlagen_2019**]

2.2 Dynamic Linked Library

Eine Dynamic Linked Library (DLL) ist eine Sammlung von Funktionen, die nicht direkt in das Hauptprogramm eingebunden werden, sondern bei Bedarf zur Laufzeit geladen werden. Das hat den Vorteil, dass Programmteile modular ausgelagert und mehrfach verwendet werden können, ohne dass der Code dupliziert werden muss.

DLLs werden häufig verwendet, um gemeinsame Funktionalität wie Datenbankzugriffe, mathematische Berechnungen oder Hardware-Kommunikation bereitzustellen. In C#werden DLLs durch separate Projekte erstellt, die öffentliche Klassen und Methoden enthalten. Diese Bibliotheken können dann in andere Projekte eingebunden und wie gewöhnliche Komponenten verwendet werden.

Die Vorteile einer DLL liegen in ihrer Wiederverwendbarkeit, Modularität und Wartbarkeit. Mehrere Programme können auf dieselbe DLL zugreifen, was Speicher spart und die Pflege erleichtert, da Änderungen nur an einer zentralen Stelle erfolgen müssen.

2.3 Modellierung

Die Modellierung ist ein zentrales Element in der Softwareentwicklung, da sie hilft, komplexe Systeme strukturiert darzustellen, zu analysieren und zu planen. Besonders in der objektorientierten Programmierung ist die visuelle Darstellung von Klassen, Beziehungen und Abläufen essenziell für das Verständnis und die Kommunikation innerhalb eines Teams.

Ein häufig verwendetes Werkzeug ist das Klassendiagramm. Es gehört zur Unified Modeling Language (UML) und stellt die Struktur eines Softwaresystems grafisch dar. In einem Klassendiagramm werden Klassen mit ihren Attributen und Methoden dargestellt, ebenso wie die Beziehungen zwischen ihnen. Dazu zählen Assoziationen (z.B. eine Klasse verwendet eine andere), Vererbungen (eine Klasse erbt von einer anderen) oder Aggregationen und Kompositionen (eine Klasse besteht aus anderen). Klassendiagramme ermöglichen es, die logische Architektur eines Systems übersichtlich darzustellen, bevor mit der eigentlichen Implementierung begonnen wird. Sie dienen sowohl der Dokumentation als auch der Kommunikation im Projektteam.

Ein weiteres wichtiges Modellierungswerkzeug ist das Aktivitätsdiagramm. Es beschreibt den Ablauf eines Prozesses oder eines bestimmten Anwendungsfalls in Form eines Flussdiagramms. Dabei werden verschiedene Aktivitäten, Entscheidungswege, Verzweigungen und parallele Abläufe grafisch dargestellt. Aktivitätsdiagramme eig-

nen sich besonders gut, um Geschäftsprozesse, Programmabläufe oder Algorithmen verständlich abzubilden. In der Softwareentwicklung wird das Aktivitätsdiagramm häufig eingesetzt, um komplexe Verhaltensweisen, wie etwa den Verbindungsaufbau zu einer Datenbank oder eine Benutzerinteraktion mit dem System, detailliert zu analysieren.

Durch die Verwendung solcher Modellierungswerkzeuge lassen sich Anforderungen klarer formulieren, Entwurfsentscheidungen fundierter treffen und potenzielle Fehlerquellen bereits in der Planungsphase erkennen. Die Modellierung bildet somit eine wichtige Brücke zwischen der konzeptionellen Planung und der technischen Umsetzung von Softwaresystemen.

2.4 Speicherverwaltung

Ein grundlegendes Thema in der Softwareentwicklung, insbesondere in der System- und Embedded-Programmierung, ist der Umgang mit Speicher. Die effiziente Verwaltung von Speicherressourcen ist entscheidend für die Leistungsfähigkeit und Stabilität eines Programms. In vielen Programmiersprachen wie C spielt dabei der direkte Zugriff auf Speicher durch sogenannte Pointer eine zentrale Rolle.

Pointer (Zeiger) sind Variablen, die Speicheradressen enthalten und dadurch auf bestimmte Speicherbereiche verweisen können. Sie ermöglichen unter anderem die dynamische Speicherreservierung sowie den Zugriff auf Arrays oder Strukturen. Durch diese Flexibilität sind sie in der Low-Level-Programmierung unverzichtbar, erfordern jedoch ein genaues Verständnis des zugrundeliegenden Speichermodells.

In Sprachen wie C erfolgt die Speicherverwaltung manuell. Das bedeutet, dass der benötigte Speicher explizit durch den Entwickler angefordert und wieder freigegeben werden muss. Eine fehlerhafte Handhabung kann dabei zu Speicherlecks, Zugriffsfehlern oder Instabilitäten führen. Daher erfordert die manuelle Speicherverwaltung ein hohes Maß an Sorgfalt und technischem Verständnis.

Die Fähigkeit, Speicher effizient und sicher zu verwalten, bildet eine wichtige Grundlage für das Entwickeln performanter und zuverlässiger Software, insbesondere im Kontext ressourcenbeschränkter Systeme wie Mikrocontroller oder eingebetteten Geräten.

2.5 Grundlagen von Datenbanken

Datenbanken spielen eine zentrale Rolle in der heutigen Softwareentwicklung, insbesondere wenn es darum geht, Informationen dauerhaft, strukturiert und zuverlässig zu speichern. Sie bilden das Rückgrat vieler Anwendungen, sei es in der Benutzerverwaltung, der Protokollierung von Prozessen oder der Speicherung geschäftsrelevanter Daten. In dieser Arbeit steht der Zugriff auf relationale Datenbanken im Fokus, insbesondere in Kombination mit einer modularen und wiederverwendbaren Anbindung über eine DLL.

Relationale Datenbanken

Relationale Datenbanken gehören zu den am weitesten verbreiteten Datenbankmodellen. Ihre Grundlage bildet das relationale Datenbankmodell, das auf der mathematischen Mengenlehre basiert. In relationalen Datenbanken werden Daten in Tabellen gespeichert, die aus Spalten (Attributen) und Zeilen (Datensätzen) bestehen. Jede Zeile entspricht dabei einem Datensatz mit genau einem Wert pro Spalte.

Ein wesentliches Merkmal relationaler Datenbanken ist die Verwendung von Schlüsseln zur eindeutigen Identifizierung und Verknüpfung von Daten. Der Primärschlüssel identifiziert jeden Datensatz innerhalb einer Tabelle eindeutig. Fremdschlüssel hingegen stellen Beziehungen zwischen verschiedenen Tabellen her und ermöglichen es, Daten logisch miteinander zu verbinden, ohne sie zu duplizieren. So kann beispielsweise eine Benutzer-ID als Fremdschlüssel in einer Tabelle für Logeinträge verwendet werden, um den Zusammenhang zwischen Nutzer und Aktion herzustellen. Diese Struktur sorgt für Datenkonsistenz und fördert ein normalisiertes, redundanzfreies Datenmodell.

Relationale Datenbanksysteme wie MySQL, PostgreSQL oder Microsoft SQL Server bieten darüber hinaus zahlreiche Funktionen zur Datenhaltung, Sicherung, Wiederherstellung und Abfrageoptimierung. Transaktionen ermöglichen es, mehrere Datenbankoperationen logisch zusammenzufassen und im Fehlerfall rückgängig zu machen, was insbesondere in sicherheitskritischen Anwendungen von großer Bedeutung ist.

Aufbau von Tabellen und Schlüsselkonzepte

Beim Aufbau relationaler Datenbanktabellen müssen neben der inhaltlichen Struktur auch technische Aspekte berücksichtigt werden. Jede Spalte einer Tabelle besitzt einen spezifischen Datentyp (z.B. Ganzzahlen, Zeichenketten oder Datumswerte), der definiert, welche Art von Informationen dort gespeichert werden können. Ergänzend dazu können Einschränkungen wie „nicht leer“, „eindeutig“ oder „Standardwert“ definiert werden, um die Datenqualität zu sichern.

Die Definition und Pflege von Primär- und Fremdschlüsseln ist essenziell für die referenzielle Integrität innerhalb des Datenbankschemas. Durch Mechanismen wie das kaskadierende Löschen oder Aktualisieren lassen sich Inkonsistenzen vermeiden und ein konsistenter Datenbestand sicherstellen.

Ein durchdachtes Datenbankdesign ist besonders in komplexen Softwaresystemen von zentraler Bedeutung. Es hilft dabei, die Wartbarkeit zu erhöhen, die Erweiterbarkeit zu erleichtern und die Fehleranfälligkeit zu reduzieren. Visuelle Hilfsmittel wie Entity-Relationship-Diagramme (ER-Diagramme) oder UML-Klassendiagramme unterstützen diesen Entwurfsprozess. SQL-Befehle

Die Interaktion mit relationalen Datenbanken erfolgt in der Regel über die strukturierte Abfragesprache SQL (Structured Query Language). SQL ist ein weltweit etablierter Standard und erlaubt sowohl die Definition der Datenstruktur als auch das Bearbeiten und Abfragen von Daten.

Der Einsatz von SQL-Befehlen erlaubt es, gezielt auf einzelne Daten zuzugreifen,

komplexe Zusammenhänge abzubilden und verschiedenste Operationen auf effiziente Weise durchzuführen. Eine saubere Abgrenzung von Anwendungs- und Datenbanklogik, wie sie in dieser Arbeit durch die Nutzung einer DLL erfolgt, sorgt dabei für ein klares und wartbares Systemdesign. Bedeutung im Kontext der Anwendung

Im Rahmen dieser Arbeit wird eine MySQL-Datenbank eingesetzt, auf die über eine eigens entwickelte C#-Bibliothek zugegriffen wird. Diese Bibliothek kapselt alle Datenbankzugriffe in Form einer wiederverwendbaren DLL und stellt Methoden für das Öffnen und Schließen von Verbindungen, die Ausführung von SQL-Abfragen sowie die Fehlerbehandlung bereit.

Ein zentraler Aspekt dabei ist die Trennung von Datenzugriff und Anwendungslogik. Durch die Auslagerung des Datenbankzugriffs in eine separate Komponente wird die Wartbarkeit erhöht, die Testbarkeit verbessert und die Wiederverwendbarkeit gewährleistet. Die klare Struktur der Datenbanktabellen, die Verwendung von Schlüsseln zur Sicherung der Integrität sowie der gezielte Einsatz von SQL ermöglichen eine performante und robuste Datenhaltung, wie sie auch in produktiven Anwendungen in der Industrie üblich ist.

2.6 Grundlagen Eingebetteter Systeme

Was ist ein Embedded System Was ist ein Debugger Programmiersprache C in Embedded Systemen Grundlagen zur Speicherverwaltung Zugriff auf Hardware Entwicklung mit dem Debugger

3 Vorgehensweise bei der Softwareentwicklung

3.1 Zielsetzung

Zu Beginn eines Softwareentwicklungsprozesses steht die präzise Ermittlung und Beschreibung der Anforderungen an das System. Diese umfassen sowohl funktionale als auch nicht-funktionale Anforderungen. Die Zielsetzung in dieser Phase besteht darin, alle relevanten Anforderungsziele, Erwartungen der Nutzer, Erfolgskriterien und Rahmenbedingungen festzuhalten und im Idealfall zu formalisieren.

Ein zentrales Ziel ist die Erstellung einer Anforderungsspezifikation, die als vertragliche Grundlage zwischen Auftraggeber und Auftragnehmer dient. Die Spezifikation definiert, was ein System leisten soll, und bildet die Basis für Entwurfsentscheidungen, Verifikation, Validierung und letztlich für die Qualitätssicherung.

Dabei wird besonderes Augenmerk auf die Vollständigkeit, Eindeutigkeit und Prüfbarkeit der Anforderungen gelegt. Die so dokumentierten Ziele ermöglichen es, die spätere Implementierung zu bewerten und den Grad der Zielerreichung objektiv zu überprüfen

3.2 Analyse

Im Analyseeschritt steht die Frage im Vordergrund, wie sich die in der Anforderungsspezifikation beschriebenen fachlichen und technischen Anforderungen zu einem tragfähigen Architekturentwurf verdichten lassen. Dabei skizziert man zunächst eine grobe Schichtenarchitektur, in der typische Funktionsbereiche, wie etwa Präsentation, Geschäftslogik und Datenzugriff, klar voneinander getrennt werden. Diese erste Zerlegung erlaubt es, Verantwortlichkeiten eindeutig zuzuordnen und spätere Änderungsauswirkungen bereits auf konzeptioneller Ebene abzuschätzen.

Anschließend erfolgt die Auswahl bewährter Entwurfsmuster, um wiederkehrende Probleme systematisch zu lösen und Konsistenz im gesamten System zu gewährleisten. So kommt zum Beispiel das „Data Access Object"-Muster (DAO) zum Einsatz, um die Datenbankzugriffe von der Geschäftslogik zu entkoppeln, während das Schichtenmuster die Kommunikation zwischen Ober- und Unterschichten regelt. Auch Aspektorientierung oder Dependency Injection können frühzeitig integriert werden, um Querschnittsbelange wie Logging, Transaktionsmanagement oder Sicherheit lose und doch zentral gesteuert abzubilden.

Ein wesentlicher Teil der Analyse ist zudem die Definition klarer Schnittstellen zwischen den Komponenten. Hier wird festgelegt, welche Methoden, Datentypen und Protokolle erforderlich sind, um die einzelnen Subsysteme miteinander zu verbinden, ohne deren Implementierungsdetails preiszugeben. Dieses Interface-Design legt den Grundstein für Parallelentwicklungen in agilen Teams und ermöglicht schon sehr früh frühe Tests von Mock-Komponenten oder Prototypen.

Schließlich fließen in der Analyse auch nicht-funktionale Anforderungen wie Performanz, Skalierbarkeit oder Betriebssicherheit in die Architekturentscheidungen ein. Schon in dieser frühen Phase wird geprüft, ob beispielsweise eine Microservices- oder eine monolithische Deploymentstrategie besser geeignet ist und welche Infrastrukturkomponenten (Container, Messagingsysteme, Caching-Layer) erforderlich sein werden. Durch diese methodisch saubere Vorarbeit in der Analysephase entsteht eine flexible, wartbare und erweiterbare Systemarchitektur, auf die alle nachfolgenden Entwurfs- und Implementierungsschritte aufbauen können

3.3 Entwurf

Für die Implementierung wird zwischen zwei Stufen unterschieden: Zunäc hst wählt man auf einer hohen Abstraktionsebene passende Algorithmen und Datenstrukturen aus und modelliert diese formal, um ihre Korrektheit und Effizienz verifizieren zu können. Darauf aufbauend erfolgt die konkrete Codierung in einer Programmiersprache und gegebenenfalls eine Optimierung, um sie auf einer bestimmten Ausführungsplattform performant zum Laufen zu bringen.

.

1. Abstrakte Implementierung In diesem Schritt wird das Programm „auf hoher Abstraktionsebene“ beschrieben. Man legt fest, welche Datenstrukturen (z. B. Listen, Bäume, Maps) und Algorithmen (z. B. Sortier- oder Suchverfahren) zum Einsatz kommen. Diese Spezifikation ist eng mit den Entwurfsspezifikationen verknüpft und erlaubt eine formale Verifikation, etwa durch Ableiten von Korrektheitsbeweisen oder Bestimmen der Laufzeitkomplexität.

2. Verifikation (auf der abstrakten Ebene)

Die abstrakten Implementierungen werden gegen ihre Spezifikationen geprüft. Hierzu nutzt man formale Methoden wie:

Beweis von Invarianten und Korrektheit mittels Induktion oder Fixpunktargumenten

Analyse der Terminierung und Ressourcenabschätzung

Ziel ist, schon vor der eigentlichen Programmierung Fehler im Design aufzudecken und zu beheben.

3. Konkrete Implementierung

Nach erfolgreicher Verifikation vervollständigt man den Code in einer spezifischen Programmiersprache. Dabei werden Details wie Speicherverwaltung, Ein-/Ausgabe-Schnittstellen oder Sprachkonstrukte (z. B. Rekursion vs. Schleifen) berücksichtigt und oft optimiert, um auf der Zielplattform bestmögliche Performance zu erzielen.

4. Verifikation (auf der konkreten Ebene)

Abschließend wird überprüft, dass die konkrete Implementierung die abstrakte Spezifikation tatsächlich erfüllt. Dies geschieht typischerweise durch Tests, statische Analysen oder erneut formale Beweise (z. B. Hoare-Logik oder Model Checking).

Dieser mehrstufige Ansatz, abstrakte Modellierung und Verifikation, gefolgt von konkreter Umsetzung und abschließender Prüfung, stellt sicher, dass bereits frühzeitig die konzeptionelle Korrektheit gewährleistet ist und spätere Optimierungen die funktionale Korrektheit nicht gefährden

3.4 Umsetzung

Unit Tests und Integrationstests Dockerbasierter Testaufbau für MySQL

4 Verbindungsbibliothek

4.1 Anfordungen

Fachliche Anforderungen

- Die Bibliothek muss es einem Anwender ermöglichen, sich mit einer Datenbank zu verbinden und diese Verbindung auch wieder zu trennen.
- Sie soll es erlauben, Datenbank-Abfragen auszuführen und die Ergebnisse als Tabellen darzustellen.
- Sie muss Befehle wie „Daten einfügen“, „aktualisieren“ oder „löschen“ unterstützen.
- Bei jeder Aktion soll klar zurückgemeldet werden, ob sie geklappt hat oder welcher Fehler aufgetreten ist.

Technische Anforderungen

- Konfigurationsparameter müssen wahlweise über Connection-String oder ein stark typisiertes Konfigurationsobjekt übergeben werden können.
- Die Bibliothek soll sich per Dependency Injection einbinden lassen.
- Öffentliche APIs müssen XML-Dokumentationskommentare besitzen, damit automatisch Intellisense-Hilfe und Referenz-Dokumentation erzeugt werden kann.
- Transaktionen (BeginTransaction / Commit / Rollback) müssen unterstützt werden.

Nichtfunktionale Anforderungen

- *Wartbarkeit*: Saubere Trennung von Schnittstelle und Implementierung, um Erweiterungen einfach zu realisieren.
- *Portierbarkeit*: Die DLL soll auf allen .NET-Implementierungen (z. B. .NET Core, .NET 5+) lauffähig sein.
- *Zuverlässigkeit*: Einheitliches Fehler- und Ausnahmehandling mit klar definierten Rückgabecodes.
- *Testbarkeit*: Alle Kernkomponenten müssen so entkoppelt sein, dass sie sich mit Unit- und Integrationstests vollständig abdecken lassen.

4.2 Konzept

Architekturübersicht

Klassenstruktur

Designentscheidungen

4.3 Implementierung

Zentrale Klassen und Methoden

Wichtige Code Beispiele

Teststrategie

Literatur

[1] Arne Heimeshoff. “Certification Exercise”. In: (2025).

[2] TaeHun Kim u. a. *Planck isocurvature constraint on primordial black holes lighter than a kiloton*. arXiv:2503.14581. März 2025. DOI: 10.48550/arXiv.2503.14581. URL: http://arxiv.org/abs/2503.14581 (besucht am 20. 03. 2025).