T1000

Jan Herrmann

14. August 2025

Inhaltsverzeichnis

Abkürzungsverzeichnis

Architekturpattern Wiederverwendbares Lösungsmuster zur Strukturierung von Softwaresystemen auf höherer Abstraktionsebene. 5

Array Datenstruktur zur Speicherung von Elementen gleicher Typen in fester Reihenfolge. 8

Assoziation Beziehungsart in der objektorientierten Modellierung, die eine strukturelle Verbindung zwischen zwei Klassen definiert; in UML durch eine Verbindungslinie dargestellt. 7

Ausnahme Ausnahmeobjekt zur Fehlerbehandlung bei unerwarteten Programmzuständen. 7

Boilerplate-Code Wiederverwendbarer Standardcode, der oft unverändert in vielen Programmen vorkommt, z. B. zur Initialisierung oder Strukturierung. 5

CI-Pipeline Continuous-Integration-Pipeline – automatisierter Prozess zum Testen, Bauen und Verteilen von Software. 5, 16

connection-Pooling Technik zur Wiederverwendung offener Datenbankverbindungen, um Verbindungsaufbauzeiten zu minimieren und Systemressourcen effizienter zu nutzen. 5

Connector Schnittstelle, die die Kommunikation zwischen Anwendung und Datenbank ermöglicht. 5

Container-Umgebung Laufzeitumgebung, in der Anwendungen isoliert und reproduzierbar in Containern betrieben werden, z. B. mit Docker. 5

DLL Dynamic Linked Library – eine zur Laufzeit ladbare Programmbibliothek. 4–7, 11–13

Eingebettet spezialisiertes, fest integriertes Rechnersystem zur Steuerung oder Überwachung technischer Geräte. 8

Enum Aufzählungstyp zur Definition fester Wertebereiche, z. B. für Statuscodes. 7, 13

Integrationstest Test, bei dem mehrere Komponenten gemeinsam geprüft werden. 5, 11, 14

Interface Schnittstelle, die Methoden vorgibt, aber keine konkrete Implementierung enthält. 6, 11–13

Klasse Bauplan für Objekte, der Attribute und Methoden definiert. 6

Komposition Stärkste Form einer Aggregation in der objektorientierten Modellierung, bei der das Teil vollständig vom Ganzen abhängig ist und ohne dieses nicht existieren kann. 7

Logger-Interface Abstraktes Interface für Logging, das per DI eingespeist wird. 13

Methode Funktion innerhalb einer Klasse, die das Verhalten eines Objekts beschreibt. 6, 7

Mock Platzhalter-Objekt, das in Tests reale Abhängigkeiten simuliert, um das Verhalten einzelner Komponenten isoliert zu prüfen. 5, 14

MySQL Open-Source-Datenbankmanagementsystem. 13

Objekt Instanz einer Klasse, die Daten (Attribute) und Verhalten (Methoden) kapselt. 6

OOP Objektorientierte Programmierung – ein Programmierparadigma, das auf den Konzepten von Klassen, Objekten, Vererbung, Polymorphie und Kapselung basiert. Es kombiniert Zustände und Funktionen innerhalb sogenannter Objekte, die über definierte Schnittstellen miteinander kommunizieren. Ziel ist die Strukturierung, Wiederverwendbarkeit und Wartbarkeit komplexer Softwaresysteme.. 6, 7

retry-Mechanism Automatischer Wiederholungsversuch bei fehlgeschlagenen Operationen, z. B. bei Netzwerkproblemen oder Datenbankzugriffen. 4, 11

SQL Structured Query Language – Sprache zur Verwaltung und Abfrage relationaler Datenbanken. 5, 13

SQL-Exception Fehlermeldung, die beim Ausführen eines SQL-Befehls auftritt, z. B. bei Syntaxfehlern, Verbindungsproblemen oder Zugriffsverletzungen. 4

Struktur Datenstruktur in C zur Gruppierung mehrerer Werte (Felder) unterschiedlichen Typs. 8

UML Unified Modeling Language – standardisierte grafische Sprache zur Modellierung und Dokumentation von Softwaresystemen, insbesondere in der objektorientierten Entwicklung. 7

Unit-Test Test einer einzelnen Codeeinheit isoliert von anderen Komponenten. 5, 11

Vererbung Mechanismus, bei dem eine Klasse Eigenschaften und Verhalten einer anderen übernimmt. 6, 7

Zeiger Zeiger auf Speicheradressen. 8

1 Einleitung

1.1 Motivation

Die Entwicklung einer eigenständigen DLL für den Datenbankzugriff bietet in modernen Softwaresystemen mehrere entscheidende Vorteile. Erstens schafft sie eine klare Trennung von Infrastruktur- und Geschäftslogik, indem sämtliche Verbindungsdetails, Fehlerbehandlung und Transaktionssteuerung an einem zentralen Ort gekapselt werden. Anstatt in jedem Modul oder Service redundanten Code für Connection-Strings, retry-Mechanism oder SQL-Exceptions zu pflegen, greift die Anwendungslogik nur noch über eine wohldefinierte Schnittstelle auf die DLL zu.

Wie Robert C. Martin in *Clean Architecture* beschreibt, ist die Trennung von Verantwortlichkeiten ein zentrales Prinzip robuster Softwarearchitekturen. Dies zeigt sich etwa im *Single Responsibility Principle* und im *Dependency Inversion Principle*, die beide dem Grundgedanken des *Separation of Concerns* folgen. Diese Prinzipien fördern nicht nur die Lesbarkeit und Wartbarkeit des Codes, sondern beschleunigen auch die Einarbeitung neuer Teammitglieder (vgl. [1], Kap. 7.1, 7.2, 11).

Zweitens ermöglicht eine modulare DB-Connector-Bibliothek die Wiederverwendbarkeit über unterschiedliche Projekte hinweg. Ob Web-API, Desktop-Anwendung oder Microservice-Architektur, alle Konsumenten beziehen ihre Datenbankfunktionen aus derselben DLL. Bei Aktualisierungen, etwa wenn auf eine neue SQL-Bibliothek umgestellt oder connection-Pooling optimiert werden soll, genügt ein Austausch der DLL ohne Änderung an den Verbraucherprojekten. So lässt sich Validierung, Logging oder Performance-Tuning einheitlich implementieren und regressionsfrei einführen.

Schließlich erhöht die Auslagerung in eine DLL die Testbarkeit enorm: In Unit-Tests und Integrationstests kann die Datenbankzugriffsschicht gezielt gemockt oder gegen eine In-Memory-Datenbank ausgetauscht werden, ohne tief in die Business-Logik eingreifen zu müssen. Auch in CI-Pipelines und Container-Umgebungen lässt sich die DB-Connector-DLL isoliert versionieren, verteilen und überwachen. Gerade im Rahmen eines dualen Studiums, in dem neben technischer Exzellenz auch Best Practices der Team- und Prozessarbeit vermittelt werden, leistet dieses *Architekturpattern* einen wertvollen Beitrag zur praxisnahen und skalierbaren Softwareentwicklung (vgl. [2], Kap. "Component-based assembly (CBD) architecture").

1.2 Problemstellung und Ziel

In vielen unserer Anwendungen wiederholt sich der gleiche, plattformabhängige Boilerplate-Code-Code für den Datenbankzugriff: Verbindungsaufbau, Parameterzuweisung, Fehlerbehandlung, Transaktionsmanagement und das Schließen von Verbindungen werden immer wieder neu implementiert. Das führt nicht nur zu erheblicher Redundanz, sondern erschwert auch die zentrale Pflege von Änderungen, etwa beim Umstieg auf neue Treiber, geänderte Sicherheitsrichtlinien oder optimierte

Konfigurationsparameter. Außerdem ist das Testen einzelner Komponenten durch fehlende Schnittstellen zur Mock-Erzeugung kompliziert und fehleranfällig.

Vor diesem Hintergrund soll eine DLL entstehen, die sämtliche wiederkehrenden Aufgaben rund um den Datenbankzugriff in einer klaren, wohlabgrenzten Schnittstelle bündelt.

Die DLL übernimmt dabei:

- Einlesen von Verbindungsparametern
- Öffnen und Schließen von Verbindungen
- Erzeugen und Ausführen von CRUD-Befehlen
- Einheitliche Fehlerbehandlung
- Unterstützung von Logging-Frameworks
- Austauschbarkeit unterschiedlicher Datenbankanbieter

Ziel ist es, eine zentrale, plattformunabhängige Komponente zu schaffen, die sich einfach und schnell in Projekte integrieren lässt und dank eines Interfaces die sich leicht mocken und testen lässt. Durch die Kapselung der Infrastruktur in einer DLL reduzieren wir den Wartungsaufwand auf genau eine Stelle: Neue Anforderungen, Security-Patches oder Performance-Optimierungen werden künftig nur noch in der Bibliothek umgesetzt und stehen damit sofort in allen Anwendungen zur Verfügung. Zudem erleichtert die klare Trennung zwischen Geschäftslogik und Datenbankzugriff das Refactoring und steigert nachhaltig die Qualität und Zuverlässigkeit unserer Software.

1.3 Zielsetzung und Projektbeschreibung

Ziel dieser Arbeit ist die Entwicklung einer wiederverwendbaren DLL für den Datenbankzugriff mit C#, die im Rahmen des dualen Studiums an der DHBW Stuttgart entsteht. Die Bibliothek kapselt zentrale Datenbankfunktionen (Verbindungsaufbau, SQL-Befehle, Fehlerbehandlung) und wird über ein klar definiertes Interface angesprochen, um saubere Entkopplung, hohe Wiederverwendbarkeit und einfache Testbarkeit sicherzustellen.

Die praktische Umsetzung wird durch strukturierte Unit- und Integrationstests ergänzt, um die Zuverlässigkeit und Einsatzfähigkeit der DLL in realitätsnahen Anwendungsszenarien zu belegen.

2 Grundlagen der Softwareentwicklung

2.1 Objektorientierte Programmierung

Die 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 (vgl. [3], Kap. 8.1).

Fehlerbehandlung erfolgt in der OOP häufig über sogenannte Ausnahmen. Statt Fehlercodes zurückzugeben, werfen Methoden bei Problemen *Ausnahmen*, die dann gezielt behandelt werden können. Eine weitere hilfreiche Spracheigenschaft sind *Enums*, die eine fest definierte Menge von Werten beschreiben, etwa Statuscodes, und so die Lesbarkeit und Wartbarkeit des Codes erhöhen (vgl. [4], S. 241-243, 147-149).

2.2 Dynamic Linked Library

Eine *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 *acrshortdll* 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 (vgl. [5], S. 509, 560).

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 OOP 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 Ünified

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*, Vererbung und *Kompositionen*. *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 (vgl. [6], Seite 16-17, 19-22, 25, 27-30).

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 eignen 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 (vgl. [7], Seite 373).

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(vgl. [6], Seite 16).

2.4 Speicherverwaltung

Ein grundlegendes Thema in der Softwareentwicklung, insbesondere in der System- und Eingebettete-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 Zeiger eine zentrale Rolle. *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(vgl. [8], Seite 2-10, 31-39).

2.5 Grundlagen eingebetteter Systeme

Eingebettete Systeme sind spezialisierte Computer, die fest in ein größeres technisches System integriert sind, um dort spezifische Steuerungs- oder Überwachungsaufgaben zu übernehmen. Sie verfügen in der Regel über begrenzte Ressourcen, arbeiten meist ohne Benutzerinteraktion und sind häufig auf Echtzeitanforderungen ausgelegt. Typische Einsatzbereiche reichen von Automobilsteuergeräten über Industrieanlagen bis hin zu Haushalts- und Medizingeräten. Für die Entwicklung solcher Systeme kommt oft die Sprache C zum Einsatz, da sie hardwarenahe Programmierung mit feinkörniger Kontrolle über Speicher, Register und Peripherie ermöglicht (vgl. [9], Kap. 1.1–1.2). Zentral für die Interaktion mit der Hardware ist der Zugriff auf sogenannte Memory-Mapped Register .Das sind festgelegte Speicheradressen, über die beispielsweise Timer, Sensoren oder LED-Ausgänge angesteuert werden. Solche Register werden üblicherweise über vordefinierte Strukturen und Bitmasken abstrahiert (vgl. [10], Kap. 4.3.2).

Zur Analyse und Fehlersuche dienen Debugger, die eine schrittweise Ausführung des Programmcodes, das Setzen von Breakpoints und das Auslesen von Speicherinhalten in Echtzeit ermöglichen. Gerade bei hardwarenahen Projekten wie der gezielten LED-Steuerung über Speicheradressen helfen Debugger dabei, Registerwerte direkt zu überwachen und das Timing des Programms präzise zu verifizieren.

2.6 Grundlagen von Datenbanken

Datenbanken stellen eine zentrale Infrastrukturkomponente moderner Softwaresysteme dar. Sie ermöglichen die dauerhafte, strukturierte und konsistente Speicherung großer Datenmengen und bilden damit das Rückgrat vieler Anwendungen wie beispielsweise in der Benutzerverwaltung, der Protokollierung von Prozessen oder im Umgang mit geschäftskritischen Informationen (vgl. [11], Seite 2).

Im Fokus dieser Arbeit steht der Zugriff auf relationale Datenbanken. Diese basieren auf dem relationalen Modell nach Codd und organisieren Daten in Tabellen, die über definierte Schlüsselbeziehungen miteinander verknüpft werden. Die Kommunikation mit der Datenbank erfolgt in der Regel über die deklarative Abfragesprache sql) (vgl. [11], Seite 55).

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 bieten darüber hinaus zahlreiche Funktionen zur Datenhaltung, Sicherung, Wiederherstellung und Abfrageoptimierung an. Transaktionen ermöglichen es, mehrere Datenbankoperationen logisch zusammenzufassen und im Fehlerfall rückgängig zu machen, was insbesondere in sicherheitskritischen Anwendungen von großer Bedeutung ist (vgl. [11], Seite 84 ff.).

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 (vgl. [11], S. 84–85). Ergänzend dazu können Einschränkungen wie „nicht leer“, „eindeutig“ oder „Standardwert“ definiert werden, um die Datenqualität zu sichern (vgl. [11], S. 85–86).

Die Definition und Pflege von Primär- und Fremdschlüsseln ist essenziell für die referenzielle Integrität innerhalb des Datenbankschemas (vgl. [11], S. 86–87). Durch Mechanismen wie das kaskadierende Löschen oder Aktualisieren lassen sich Inkonsistenzen vermeiden und ein konsistenter Datenbestand sicherstellen (vgl. [11], S. 86–87).

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 (vgl. [11], S. 229–244 und 294–310).

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 (vgl. [11], S. 84–89).

Der Einsatz von SQL-Befehlen erlaubt es, gezielt auf einzelne Daten zuzugreifen, komplexe Zusammenhänge abzubilden und verschiedenste Operationen auf effiziente Weise durchzuführen (vgl. [11], S. 84–89).

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 (vgl. [11], S. 35–39 und 84–87).

3 Vorgehensweise bei der Softwareentwicklung

3.1 Zielsetzung

Die Zielsetzung eines Softwareentwicklungsprojekts beschreibt den angestrebten Zweck des Systems und dient als verbindliche Grundlage für alle nachfolgenden Aktivitäten im Entwicklungsprozess. Sommerville definiert im Rahmen des Requirements Engineering, dass diese Zieldefinition sowohl aus den strategischen Geschäftszielen als auch aus den Bedürfnissen der Stakeholder abgeleitet werden muss. Sie umfasst dabei sowohl funktionale Ziele, wie die zu erbringenden Systemdienste. Als auch nicht-funktionale Ziele, welche Qualitätsmerkmale wie Leistung, Zuverlässigkeit, Sicherheit oder Wartbarkeit festlegen (vgl. [12], Seite 116–122).

Für das Projekt T1000 bedeutet dies konkret, eine modulare, wiederverwendbare DLL für den Datenbankzugriff zu entwickeln, die als einheitliche Schnittstelle zwischen Anwendungslogik und MySQL-Datenbank fungiert.

Die spezifischen Ziele umfassen:

- Entwicklung einer klar definierten, entkoppelten Schnittstelle (Interface) für CRUD-Operationen,
- Sicherstellung der Wiederverwendbarkeit in unterschiedlichen Projekten und Architekturen,
- Implementierung standardisierter Mechanismen zur Fehlerbehandlung, zum Logging und für den retry-Mechanism,
- Gewährleistung einer hohen Testbarkeit durch gezielte Unit-Tests und Integrationstests.

3.2 Analyse

Die Analysephase dient dazu, die in der Zielsetzung definierten Projektziele in präzise, überprüfbare Anforderungen zu überführen. Nach Sommerville ist dieser Schritt Teil des *Requirements Engineering* und umfasst insbesondere die systematische Erhebung, Strukturierung und Validierung der Anforderungen, um eine belastbare Grundlage für Entwurf und Implementierung zu schaffen (vgl. [12], Seite 114).

Ein zentraler Aspekt ist die Anforderungsvalidierung, bei der überprüft wird, ob die erfassten Anforderungen die tatsächlichen Bedürfnisse der Stakeholder widerspiegeln und gleichzeitig umsetzbar sind. Sommerville nennt hierfür folgende zentrale Prüfkriterien:

- **Eindeutigkeit** - jede Anforderung muss klar und ohne Interpretationsspielraum formuliert sein.
- **Konsistenz** - es dürfen keine Widersprüche zwischen einzelnen Anforderungen bestehen.
- **Vollständigkeit** – alle relevanten Funktionen und Einschränkungen müssen erfasst sein.
- **Realisierbarkeit** – die Umsetzung muss mit den vorhandenen Ressourcen, Technologien und im gegebenen Zeitrahmen möglich sein.
- **Verifizierbarkeit** – Anforderungen müssen so formuliert sein, dass ihre Erfüllung später messbar und testbar ist.

(vgl. [12], Seite 144)

Für das Projekt *T1000* bedeutet dies, dass die funktionalen Anforderungen also die Implementierung von CRUD-Operationen über ein Interface als auch die nichtfunktionalen Anforderungen wie Wiederverwendbarkeit, Performance und Testbarkeit, anhand dieser Kriterien überprüft werden. Die Analyse stellt somit sicher, dass sowohl technische als auch organisatorische Rahmenbedingungen berücksichtigt werden und dass die geplante DLL in späteren Projektphasen ohne grundlegende Anpassungen umgesetzt werden kann.

3.3 Entwurf

Der Entwurf beschreibt die strukturelle und architektonische Gestaltung des Systems, um die in der Analyse definierten Anforderungen effizient und wartbar umzusetzen. Nach Sommerville stellt die *Schichtenarchitektur* ein etabliertes Architekturmuster dar, bei dem das System in klar abgegrenzte Ebenen unterteilt wird, die jeweils definierte Verantwortlichkeiten haben (vgl. [12], Kap. 6.3.1, S. 194 ff.). Dieses Muster erleichtert die Modularisierung, unterstützt die Wiederverwendbarkeit einzelner Komponenten und reduziert die Kopplung zwischen den Systemteilen.

Für das Projekt *T1000* wurde eine dreischichtige Struktur umgesetzt:

1. **Anwendungsschicht** – übernimmt die Steuerung des Programmablaufs und ruft über die definierten Schnittstellen die benötigten Datenbankoperationen auf (`Program.cs`).
2. **Abstraktionsschicht** – definiert mit dem Interface `IConnector` die vertraglichen Methoden für den Datenbankzugriff und kapselt die Implementierungsdetails.
3. **Datenzugriffsschicht** – implementiert mit der Klasse `MySqlAccess` die konkreten CRUD-Operationen auf Basis des MySQL-Connectors.

Die klare Trennung zwischen diesen Schichten ermöglicht es, Änderungen an der Datenbankimplementierung vorzunehmen, ohne die Anwendungsschicht anzupassen, und erleichtert die Wiederverwendbarkeit der DLL in unterschiedlichen Projekten. Zudem wird die Testbarkeit erhöht, da sich durch die Verwendung des Interfaces `IConnector` sowohl Unit-Tests mit Mocks als auch Integrationstests gegen eine reale Datenbankumgebung durchführen lassen.

3.4 Umsetzung

3.5 Validierung

4 Verbindungsbibliothek

4.1 Anforderungen

Fachliche Anforderungen:

Die entwickelte Verbindungsbibliothek soll eine einfache, sichere und wiederverwendbare Schnittstelle für den Zugriff auf eine relationale MySQL-Datenbank bereitstellen.

Dazu gehören insbesondere:

- Aufbau und Abbau einer Datenbankverbindung
- Durchführung von SQL-Operationen: `select`, `INSERT`, `UPDATE`, `DELETE`
- Rückgabe konsistenter Fehlercodes über den Enum `errorValues`
- Verwendung eines Logger-Interface zur Protokollierung aller relevanten Vorgänge

Technische Anforderungen:

- Integration über ein generisches Interface (`IConnector.cs`)
- Implementierung in C mit Unterstützung für Dependency Injection
- Unterstützung typischer Fehlerquellen (z.,B. Authentifizierungsprobleme, Timeout, Syntaxfehler)
- Kompatibilität mit allen modernen .NET-Versionen

Nichtfunktionale Anforderungen:

- *Testbarkeit*: Unit- und Integrationstests müssen möglich sein
- *Zuverlässigkeit*: Robust gegenüber ungültigen Eingaben oder fehlerhaften Abfragen
- *Wartbarkeit*: Durch saubere Trennung zwischen Schnittstelle und Implementierung
- *Erweiterbarkeit*: Leicht auf andere Datenbankanbieter übertragbar

4.2 Konzept

Architekturübersicht:

Die Bibliothek besteht aus einem klaren Interface `IConnector`, das von der konkreten Klasse `MySqlAccess` implementiert wird. Diese Trennung ermöglicht es, die konkrete Implementierung in Tests durch Mock-Objekte zu ersetzen und verschiedene Datenbank-Backends zu unterstützen.

- `IConnector.cs`: Definiert alle Methoden für die Interaktion mit einer Datenbank
- `MySqlAccess.cs`: Implementiert das Interface für das spezifische DBMS `MySQL`
- `ErrorCodes.cs`: Enthält das zentrale Enum `errorValues` zur Rückgabe standardisierter Statuswerte
- `ErrorMessages.xml`: Ermöglicht sprachspezifische Rückmeldungen für Clients

Klassenstruktur:

Die Klasse `MySqlAccess` verwaltet die Lebensdauer der Verbindung und implementiert alle CRUD-Operationen (Create, Read, Update, Delete). Dabei wird bei jedem Methodenaufruf geprüft, ob die Verbindung geöffnet ist, und gegebenenfalls

ein Reconnect durchgeführt. Fehler werden über den Logger dokumentiert und als strukturierte `errorValues` zurückgegeben.

Die Verwendung eines Logger-Interfaces (`ILogger<MySqlAccess>`) gewährleistet lose Kopplung und ermöglicht es, Log-Ausgaben gezielt zu steuern oder während Tests vollständig zu deaktivieren.

Designentscheidungen:

- **Dependency Injection**: Die `ILogger`-Instanz wird von außen eingespeist, um Testbarkeit und Flexibilität zu erhöhen
- **Fehlertoleranz**: Durch umfassendes `try-catch` mit Fehlerdifferenzierung nach SQL-Fehlernummern (z.,B. `1045` für falsches Passwort)
- **Kapselung**: Alle Details zur Verbindung, Fehlerbehandlung und SQL-Erstellung sind in einer Klasse gekapselt
- **Transparenz**: Der Verbindungsstatus wird über das Attribut `flagStatus` jederzeit sichtbar gemacht

4.3 Implementierung

Zentrale Klassen und Methoden:

- `openConnection()`: Öffnet eine bestehende Verbindung oder gibt einen Fehler zurück (z.,B. `ServerConnectionFailed`)
- `closeConnection()`: Schließt eine aktive Verbindung, behandelt doppelte Aufrufe mit `ConnectionAlreadyClosed`
- `select(...)`: Führt eine Abfrage durch und gibt die Ergebnisse in der Konsole aus
- `insert(...)`: Fügt Datensätze ein, prüft auf Duplikate (`DuplicateEntry`) und Constraints
- `update(...)`: Führt Aktualisierungen durch, optional mit JOIN- und WHERE-Klauseln
- `delete(...)`: Löscht gezielt Datensätze mit optionalem LIMIT

Wichtige Codebeispiele:

```
public errorValues insert(string tableName, string values)
{
if (string.IsNullOrEmpty(tableName) || string.
   IsNullOrEmpty(values))
return errorValues.EmptyInputParameters;
```

```
csharp
Code kopieren
string query = $"INSERT␣INTO␣{tableName}␣VALUES({values});
    ";

using (var cmd = new MySqlCommand(query, connection))
{
     int affectedRows = cmd.ExecuteNonQuery();
     return affectedRows > 0 ? errorValues.Success :
        errorValues.NoData;
}
}
```

Teststrategie:

Die Teststrategie folgt dem AAA-Muster (Arrange, Act, Assert) und deckt sowohl positive als auch negative Fälle ab:

- **Unit Tests:** Getrennte Tests für Konstruktor, Verbindungsaufbau, CRUD-Methoden
- **Integration Tests:** Realer Zugriff auf eine Testdatenbank in einem Docker-Container mit echtem MySQL-Server
- **Mocking:** Einsatz von `Moq` zur Simulation der Logger-Komponente
- **CI-Integration:** Die Tests laufen automatisiert in einer CI-Pipeline und melden regressionssicher alle Statuswerte zurück

Beispiel für einen Unit-Test:

```
[Fact]
public void
    Constructor_SetsFlagStatusToError_WhenParametersAreInvalid
    ()
{
var mockLogger = new Mock<ILogger<MySqlAccess>>();
var mySqlAccess = new MySqlAccess(null, mockLogger.Object)
    ;
Assert.Equal(errorValues.EmptyInputParameters, mySqlAccess
    .flagStatus);
}
```

Durch diesen Testaufbau wird gewährleistet, dass sowohl ungültige Parameter als auch Verbindungsfehler, SQL-Probleme und Systemausfälle zuverlässig erkannt und differenziert behandelt werden können.

Literatur

[1] R. C. Martin, *Clean architecture : das Praxis-Handbuch für professionelles Softwaredesign : Regeln und Paradigmen für effiziente Softwarestrukturen*, Deutsche Ausgabe, 1. Auflage. Frechen: mitp, 2018, ISBN: 978-3-95845-724-9.

[2] P. R. Chelliah und A. Amandeep, *Architectural Patterns: Uncover Essential Patterns in the Most Indispensable Realm of Enterprise Architecture*. Birmingham: Packt Publishing Limited, 2017.

[3] M. Broy, „Grundlagen der Objektorientierung," in *Logische und Methodische Grundlagen der Programm- und Systementwicklung: Datenstrukturen, funktionale, sequenzielle und objektorientierte Programmierung - Unter Mitarbeit von Alexander Malkis*, M. Broy, Hrsg., Wiesbaden: Springer Fachmedien, 2019, S. 381–433, ISBN: 978-3-658-26302-7. DOI: 10.1007/978-3-658-26302-7_8. besucht am 22. Apr. 2025. Adresse: https://doi.org/10.1007/978-3-658-26302-7_8.

[4] J. Bloch, *Effective Java*. Addison-Wesley Professional, 8. Mai 2008, 375 S., Google-Books-ID: ka2VUBqHiWkC, ISBN: 978-0-13-277804-6.

[5] A. Troelsen, *Pro C# 7 : With .NET and .NET Core* (SpringerLink Bücher), 8th ed. Berkeley, CA: Apress, 2017, ISBN: 978-1-4842-3018-3.

[6] B. Rumpe, „Klassendiagramme," in *Modellierung mit UML: Sprache, Konzepte und Methodik*, B. Rumpe, Hrsg., Berlin, Heidelberg: Springer, 2011, S. 15–40, ISBN: 978-3-642-22413-3. DOI: 10.1007/978-3-642-22413-3_2. besucht am 1. Aug. 2025. Adresse: https://doi.org/10.1007/978-3-642-22413-3_2.

[7] Object Management Group, „OMG Unified Modeling Language (OMG UML) Version 2.5.1," Object Management Group, Techn. Ber. formal/2017-12-05, Dez. 2017, Online verfügbar unter https://www.omg.org/spec/UML/, zuletzt abgerufen am 01.08.2025. Adresse: https://www.omg.org/spec/UML/.

[8] R. M. Reese, *Understanding and using C pointers: core techniques for memory management*. Beijing Sebastopol, California: O'Reilly Media, 2013, 1 S., ISBN: 978-1-4493-4418-4 978-1-4493-4456-6.

[9] T. Noergaard, *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. Newnes, 31. Dez. 2012, 670 S., Google-Books-ID: 96jSXetmlzYC, ISBN: 978-0-12-382197-3.

[10] C. Ünsalan, H. D. Gürhan und M. E. Yücel, *Embedded System Design with ARM Cortex-M Microcontrollers, Applications with C, C++ and MicroPython*, Englisch, 1. Aufl. Springer Cham, 2022, S. XIV + 569, Access provided by DHBW Stuttgart über SpringerLink, ISBN: 978-3-030-88439-0. DOI: 10.1007/978-3-030-88439-0. Adresse: https://doi.org/10.1007/978-3-030-88439-0.

[11] R. Elmasri und S. B. Navathe, *Fundamentals of Database Systems: Pearson New International Edition*. Harlow: Pearson Education UK, 2013, Zugriff am 4.8.2025, ISBN: 978-1-292-03803-2. Adresse: http://ebookcentral.proquest.com/lib/dhbw-stuttgart/detail.action?docID=5248269.

[12] I. Sommerville, *Software Engineering*. Pearson Deutschland GmbH, 2012, ISBN: 978-3-86326-512-0. besucht am 11. Aug. 2025. Adresse: http://ebookcentral.proquest.com/lib/dhbw-stuttgart/detail.action?docID=5133710.