Management Summary – Praxisarbeit Programmiertechnik (C)

Projekt: Labyrinth (Konsolenspiel)
Autor: Salih San · **Klasse:** TE.X.BA.3_4.46-ZH-S2504 · **Datum:** 30.09.2025

Ziel & Kontext

Ziel dieser Arbeit war es, ein kleines, aber sauberes Konsolenspiel in **C** zu entwickeln und dabei zentrale Inhalte aus der Programmiertechnik praktisch anzuwenden: Datenstrukturen, Algorithmik, Steuerungslogik, dynamische Speicherverwaltung und Build-Automatisierung.
Das Spiel erzeugt ein zufälliges Labyrinth; der Spieler **P** bewegt sich per **W/A/S/D** zum Schatz **T**. **Q** beendet das Spiel. Der Fokus lag auf **klarer Modularisierung**, **nachvollziehbarer Vorgehensweise** und **praxistauglichem Ergebnis** (einfacher Build/Run, kurze Doku, kleiner Test).

Lösung in Kürze

- **Technologie & Struktur:** C11, Module `main.c` (Programmfluss & I/O), `labyrinth.c` (Spieltlogik), `labyrinth.h` (Schnittstellen). Build via **Makefile** (`make`, `make run`, `make test`, `make clean`).
- **Datenmodell:** `struct Maze` mit dynamischem 2D-Array (`char**`), Felder: **`.`** (frei), **`0`** (Hindernis), sowie Positionen von **P** und **T**.
- **Algorithmen:** Zufällige Hindernis-Generierung nach Verhältnis **ratio**; Kollisionsprüfung gegen Rand/Wände; Siegprüfung bei Positionsgleichheit von **P** und **T**.
- **Parametrisierung:** Optionaler Start mit `./labyrinth [rows cols ratio]` (z. B. `./labyrinth 12 12 0.2`) zur Anpassung von Größe und Hindernisdichte.
- **Steuerung: W/A/S/D** (Bewegung), **Q** (Beenden).

Erfüllte Anforderungen (Mapping)

- **Anzeige des Labyrinths** → `maze_draw`
- **Bewegung des Spielers** inkl. Kollision → `maze_move`
- **Schatz finden & Siegmeldung** → `maze_is_on_treasure` + Ausgabe in `main`
- **Beenden des Spiels** → Abfrage **Q** in `main`
- **Dokumentation** → README, diese Management Summary, Dossier (Design, Implementierung, Test)

Vorgehen & Qualitätssicherung

- **Vorgehen:** Zuerst Datenmodell & Schnittstellen im Header definiert, danach die Funktionen iterativ implementiert und über die Konsole getestet. Randfälle (Wand/Randkollision, Beenden) wurden früh überprüft.
- **Speicher & Robustheit:** Alle dynamischen Speicherbereiche werden freigegeben (`maze_free`); Eingaben werden validiert; Out-of-Bounds-Bewegungen verhindert.
- **Build & Tests:** Ein-Befehl-Build mit **`make`**. Ein kleiner **Autotest** (`make test`) verifiziert elementare Bewegungen/Kollision – Ausgabe

„OK“. Zusätzlich manuelle Tests (tabellarisch im Dossier) und Screenshots der Programmausgabe (`docs/diagramme`).

Praxistauglichkeit & Nutzen

Schnell reproduzierbar (nur **`make`**), klar dokumentiert (README), modulare Struktur (Header/Implementierung) → dadurch gut wartbar und erweiterbar (z. B. zusätzliche Regeln, Punkte oder Level). Das Projekt eignet sich als Vorlage/Übung für grundlegende C-Konzepte.

Grenzen & Risiken

Die Darstellung ist bewusst **textbasiert** (keine GUI, keine Farben). Bei sehr hoher Hindernisdichte kann ein Pfad **unwahrscheinlicher** werden; daher sind Standardwerte **moderat** gewählt. Direkte Tastenabfrage ohne Enter wurde zugunsten der Portabilität nicht umgesetzt.

Nächste Schritte (Roadmap)

- Mehrere Schätze/Level und **Punkte-Zähler**
- **Respawn** eines neuen Schatzes statt sofortigem Spielende
- Direkte Tastenabfrage ohne Enter; farbliche Hervorhebung
- Optional: Seed-Parameter für reproduzierbare Layouts

Kurzanleitung

```bash make ./labyrinth # Standard: 10x10, ~15% Hindernisse # Beispiel mit Parametern: ./labyrinth 12 12 0.2 make test # Autotest -> “OK”
```