Projekt Supermarkt

Ziele

- **Implementierung eines Supermarktes:** Systemverstaendnis foerden
- **Integration von Komponenten aus unterschiedlichen Programmiersprachen:** Modulare und mehrsprachige Architektur: Durch die Aufteilung in klar getrennte Module und die gezielte Nutzung von C++ (Backend) sowie Python (Frontend) wird Skalierbarkeit ermöglicht
- **Umgang von Zugriffsrechten auf Dateisysteme und Ressourcen:** Sicherstellung des synchronisierten Zugriffs (z. B. per std::mutex), um Race Conditions und Deadlocks bei gleichzeitiger Dateiverarbeitung zu verhindern.
- **Schnelle Testbarkeit trotz hoher Komplexität:** Trotz hohem Overhead durch C++ sollen die Tests in Python weniger Millisekunden brauchen
- **Erfassung der Coverage:** Mittels Befehlen im Terminal wird die Coverage in C++ und Python ermittelt

Ausführung

- **Executable generieren:** Ctrl + Shift + B
- **C++-Modul in Python integrieren:** pip install .
- **Ermitteln der Coverage in Python:** ./run_python_tests.sh im Terminal als Befehl eingeben
- **Ermitteln der Coverage in C++:** ./run_cpp_coverage.sh im Terminal als Befehl eingeben

Design

- **Basisklassen:** Datum, Kunde, Haendler, Produkt, Konto, Warenkorb, Supermarkt, Kassenzettel
- **Datensätze:** Export von Kassenzetteln, Inventur- und Kunden-, Haendler-, sowie Warenkorbdateien
- **Fokus:** Trennung von Datenzugriff (`ReadData`) und Auswertung (`Statistik`) sowie Monitoring (`Logging`)
- **Vermeidung von Race Conditions:** Multithreading-fähiger Dateizugriff mittels Mutexen

Hauptfunktionen

CMakeLists.txt

- Zentrales Dokument zur Steuerung vom Compile-Vorgang
- **src:** Ablageort für die Source-Dateien
- **inc:** Ablageort für die Header-Dateien, Unterteilung in base und utils
- **Verwendeter Standard:** CXX 17
- **ccache:** Zwischenspeichern der Build-Objekte im Cache zur Beschleunigung vom Build-Vorgang
- **Optimierung O2:** Beschleunigung vom Build-Vorgang und Debugging sind möglich
- **Debugging:** Aktivierung vom Debuggung, um die Fehlersuche zu vereinfachen

.vscode

- **tasks.json:** JSON-Datei fuers Steuern vom Kompilier-Vorgang inkl. Shortcuts fuer die Tastatur, vor allem CTRL+SHIFT+B zum Kompilieren
- **settings.json:** JSON-Datei zur Konfiguration für C++ und meinem Modul py_bindings

build

- Speicherort fuer die Executables, Binärdateien und Befehle fürs Komplieren

inc

- Ablageort für die Header-Dateien
- **base:** Unterordner für die einfachen Klassen
- **utils:** Unterordner für die Template-Klassen, die in anderen Projekten nutzbar sind

src

- Speicherort für die Source-Dateien
- **tests.cpp:** Test-File fuer alle Executables

Ordner data

- Aufbewahrungsort fuer die generierten Kassenzettel, Warenbestaende, Kundenliste, Haendlerlisten
- Referenzpunkt fuer die Datenverarbeitung

Integration von C++ in Python

bindings

- Ablageort fuer die Executables aus C++, um mit pybind11 zu interagieren
- Deklaration der Funktionen aus C++ fuer Python, **Achtung:** Funktionen müssen *Snake-Case* sein, damit Sie in Python nahtlos funktionieren

py_bindings.pyi

- **Zweck:** Minimierung der Fehlermeldungen, die durch Pylint erkennt werden
- **Umsetzung:** Reine Deklaration der Funktionen aus der Datei ./bindings/py_bindings.cpp, damit der Compiler erkennt, dass diese Funktionen bereits in C++ definiert wurden.

pybind11

- **Installation:** Klonen vom Git-Repository Pybind11
- **Warum:** Noetig, um eine Schnittstelle zwischen C++ und Python zu ermoeglichen

Dateien zur Python-Konfiguration

- **pytest.ini:** Konfiguration der Coverage fuer den Code in Python
- **.env:** Konfiguration, um dem System mitzuteilen, wo der Ordner fuer die Executable sich befindet
- **setup-py:** Notwendig, damit der Python-Interpreter das Modul in Python verwenden kann

Integration in GitHub

- **ci.yml im Ordner .github:** Konfiguration von CI/CD
- **.gitignore:** Ordner und Dateien, die ignoriert werden koennen
- **gitmodules:** Mitteilung gegenüber dem System, dass pybind11 als Submodul zu verwenden ist

Ergebnisse

- **CI/CD-Tests:** Tests laufen stabil, inklusive C++- und Python-Komponenten auf GitHub Actions.
- **Pybind11:** konsistente Schnittstelle zwischen C++ und Python.
- **Python-Tests über C++-Module:** Worst Case benötigt unter 200 ms, im Best-Case ca. 140 ms
- **Formatierung für C++:** clang-format, cpplint
- **C++-Style von Google:** Skalierbar und wartbar dank durchdachter Architektur