Pfad: **data/containers.json**

Ziel: Erstelle den finalen Inhalt für die JSON-Datei `data/containers.json`. Diese Datei definiert verschiedene Containertypen für das Container-Ladetool.

Rückgabeformat: Gib nur ein valides JSON-Array mit mehreren Container-Objekten zurück. Keine Kommentare, keine Erläuterungen.

Einschränkungen:
- Jeder Eintrag muss die Felder `id`, `name`, `inner_length_mm`, `inner_width_mm`, `inner_height_mm` und `door_height_mm` enthalten.
- Vermeide zusätzliche Felder wie `door_width_mm` oder `max_payload_kg`.
- Verwende gängige Containergrößen (20ft, 40ft, 40ft HC, 40ft OT) und weise jedem eine eindeutige ID zu.
- Alle Maße sind in Millimetern und spiegeln gängige ISO-Normen wider.

Kontext:
- Dieses Tool unterstützt genau einen Container pro Projekt.
- Es muss möglich sein, aus mehreren Containertypen zu wählen.
- Verwende semantische IDs wie „20ft-std“, „40ft-hc“ usw. und passe `name` entsprechend an.

Hier die Maße der einzelnen Container:
20ft Container
inner_length: 6000mm
inner_width: 2300mm
inner_height: 2350mm
door_heigth: 2228mm

40ft Container
inner_length: 12000mm
inner_width: 2300mm
inner_height: 2395mm
door_heigth: 2228mm

40ft HC Container
inner_length: 12000mm
inner_width: 2300mm
inner_height: 2698mm
door_heigth: 2550mm

40ft Open Top Container
inner_length: 12000mm
inner_width: 2300mm
inner_height: 3000mm
door_heigth: 2650mm

—----------

Pfad: src/container_tool/core/models.py

Ziel: Schreibe das vollständige Python-Modul `models.py`. Es implementiert alle Daten- und Geometrieklassen des Projekts, damit andere Module darauf aufbauen können.

Rückgabeformat: Gib ausschließlich den vollständigen Python-Code für dieses Modul aus; ohne Erklärungen oder Codeblöcke.

Einschränkungen:
- Zielversion Python 3.13.5; nur Standardbibliothek, `typing`, `dataclasses`, sowie falls nötig `numpy` verwenden.
- Keine unbenutzten Imports, kein globaler Zustand. Beachte PEP 8.
- Jede Klasse muss serialisierbar sein (Methoden `to_dict`/`from_dict`) für die `.clp`-Datei.
- Implementiere thread-safe Methoden, da die GUI in separaten Threads laufen kann.
- Nutze Rotating-Logging via `logging` nicht in diesem Modul; Fehler sind durch Exceptions zu kommunizieren.

Kontext:
- Definiere `Container` mit Feldern `id`, `name`, `inner_length_mm`, `inner_width_mm`, `inner_height_mm` und `door_height_mm`.
- Implementiere `Box` entsprechend dem Box-Schema: `name`, `length_mm`, `width_mm`, `height_mm`, optional `weight_kg`, `color_hex`, `pos_x_mm`, `pos_y_mm` und `rot_deg` (0 oder 90). Füge Methoden wie `bbox()` (liefert `(x_min, y_min, x_max, y_max)` abhängig von Rotation), `volume_mm3`, `footprint_mm2` und eine Methode zum Rotieren um 90°.
- Implementiere `Stack` als Sammlung identischer `Box`-Objekte. Die Stapelposition wird durch die gemeinsame `pos_x_mm`/`pos_y_mm` bestimmt; die Höhe ergibt sich aus der Summe der Einzelhöhen. Methoden: `bbox()`, `total_height_mm`, `total_weight_kg`, `box_count` sowie ein Iterator über die enthaltenen Boxen. Prüfe, ob eine neue Box zu diesem Stack passt (gleiche Länge/Breite/Rotation; Snap-Toleranz ±10 mm).
- Definiere `Project` mit Attributen: `container: Container`, `boxes: list[Box|Stack]`, `meta` mit `created_at`, `version`, `user`. Füge Methoden zum Hinzufügen/Entfernen von Boxen bzw. Stapeln und zur Berechnung des Gesamtgewichts/Höhe hinzu.
- Achte auf konsistente Einheiten (mm, kg). Wandle keine Werte ohne Notwendigkeit um.

—--------

Pfad: src/container_tool/core/stack.py

Ziel: Implementiere die Stapel-Logik in `stack.py`. Dieses Modul stellt Funktionen bereit, um aus einzelnen Boxen einen Stapel zu bilden und Snap-Toleranzen zu prüfen.

Rückgabeformat: Liefere nur den vollständigen Python-Code für dieses Modul.

Einschränkungen:
- Verwende nur die Klassen aus `models.py`; keine GUI-Elemente.
- Berücksichtige die Snap-Toleranz: Ein Stapel entsteht, wenn die Mittelpunkte zweier Boxen in X- und Y-Richtung maximal ±10 mm voneinander abweichen.
- Stapeln ist nur erlaubt, wenn Länge, Breite und Rotation der Boxen identisch sind und die resultierende Stapelhöhe die Türhöhe des Containers nicht überschreitet.
- Gib aussagekräftige Exceptions zurück, wenn Stapeln nicht möglich ist.

Kontext:
- Implementiere eine Funktion `can_stack(box_a: Box, box_b: Box, container: Container) -> bool`, die prüft, ob zwei Boxen stapelbar sind (Snap-Toleranz, identische Abmessungen, Höhe unterhalb `door_height_mm`).
- Implementiere `create_stack(boxes: list[Box], container: Container) -> Stack`, welche aus einer Liste kompatibler Boxen einen neuen `Stack` erzeugt. Setze die `pos_x_mm`/`pos_y_mm` des Stapels auf die des ersten Elements und summiere die Höhen.
- Implementiere `add_to_stack(stack: Stack, box: Box, container: Container) -> Stack`, um einen weiteren Karton hinzuzufügen, sofern zulässig.
- Nutze Helferfunktionen zur Berechnung der vertikalen Resthöhe; diese sollten Exceptions werfen, wenn die Türhöhe überschritten wird.

—----------

Pfad: src/container_tool/core/collision.py

Ziel: Entwickle das Modul `collision.py`, das sämtliche 2D-Kollisionen und die Prüfung der Türhöhe übernimmt. Es soll vom GUI-Canvas genutzt werden.

Rückgabeformat: Gib ausschließlich den vollständigen Python-Code zurück.

Einschränkungen:
- Keine GUI-Bibliotheken importieren; das Modul darf nur logische Funktionen bereitstellen.
- Kollisionserkennung muss performant sein: ≤ 50 ms bei 200 Boxen und 40 Typen. Nutze effiziente Algorithmen (z. B. Achsenorientierte Bounding-Box).
- Nicht blockieren – Funktionen müssen schnell sein; keine Sleep-Aufrufe.

Kontext:
- Implementiere `check_collisions(candidate: Box|Stack, placed: list[Box|Stack], container: Container) -> tuple[bool, list[Box|Stack]]`. Rückgabewert ist ein Tupel `(ok,

kollidierende)`, wobei `ok=True` bedeutet, dass der Kandidat ohne Kollision platziert werden kann. Prüfe:
- Überschreitet der Kandidat die Containergrenzen in X-/Y-Richtung? Falls ja, gilt dies als Kollision.
- Überlappt das Bounding-Box des Kandidaten mit dem eines vorhandenen Objekts? Wenn ja, füge dieses Objekt zur Liste der Kollisionen hinzu.
- Übersteigt die Stapelhöhe die Türhöhe des Containers? Gib eine spezielle Kollision zurück.
- Implementiere `is_within_container(obj: Box|Stack, container: Container) -> bool`.
- Stelle Hilfsfunktionen wie `overlaps(a_bbox, b_bbox)` bereit.
- Gib keine visuelle Markierung zurück; das Rendering-Modul setzt bei Kollision Rot ein.

—----------
Pfad: src/container_tool/core/io_clp.py

Ziel: Implementiere die Persistenzfunktionen in `io_clp.py`. Dieses Modul liest und speichert Projektdateien im `.clp`-Format und nutzt `containers.json` für Container-Referenzen.

Rückgabeformat: Nur der vollständige Python-Code.

Einschränkungen:
- Nutze ausschließlich Standardbibliothek (`json`, `datetime`, `pathlib`, `logging`) sowie die Klassen aus `models.py`.
- Bei Lese- oder Schreibfehlern sollen Exceptions ausgelöst werden; die GUI fängt diese ab und zeigt sie an.
- Speichere Dateien atomar und prüfe vor dem Schreiben, ob die Datei geöffnet oder schreibgeschützt ist.
- Versionierung nach SemVer muss in `meta.version` gespeichert werden. Die Funktion zum Speichern erhält die neue Versionsnummer.

Kontext:
- Implementiere `load_clp(path: str) -> Project`, die eine `.clp`-Datei (JSON) einliest, Container- und Box-Objekte via `from_dict` erzeugt und die Meta-Daten (`created_at`, `version`, `user`) validiert. Bei unbekanntem Containertyp soll eine Exception ausgelöst werden.
- Implementiere `save_clp(project: Project, path: str, user: str, version: str) -> None`: setzt `meta.created_at` auf die aktuelle UTC-Zeit, `meta.user` auf den übergebenen Benutzer und `meta.version` auf die übergebene Version. Serialisiert Container und Boxen mit `to_dict` und schreibt das JSON formatiert (indent=2).
- Implementiere `load_containers_definitions(path: str = None) -> dict[str, Container]`, das `data/containers.json` lädt. Akzeptiere optional einen alternativen Pfad für Tests.
- Verwende relative Pfade, um Portabilität zu sichern.

—----------

Pfad: src/container_tool/gui/table_widget.py

Ziel: Entwickle `table_widget.py`. Dieses Modul definiert einen QTableWidget-basierten Eingabedialog, mit dem Benutzer Boxen konfigurieren können. Es sendet beim Erstellen der Boxen ein Signal an andere Komponenten.

Rückgabeformat: Vollständiger Python-Code für dieses Modul.

Einschränkungen:
- Zielbibliothek: PySide6 6.5.2; importiere nur benötigte Qt-Klassen (`QTableWidget`, `QTableWidgetItem`, `QColor`, `QBrush`, `QHeaderView`, `QWidget`, `QPushButton`, `QHBoxLayout`, `QObject`, `Signal`).
- Der Widget darf die Applikation nicht blockieren; Eingaben müssen sofort validiert werden.
- Ungültige Eingaben (Nicht-Integer bei L/B/H/Menge, negative Werte, Gewicht mit mehr als zwei Dezimalstellen) sind durch rote Zellrahmen zu kennzeichnen und werden beim Erstellen ignoriert.
- Jedem Box-Typ muss automatisch eine farbenblind-freundliche Zufallsfarbe zugewiesen werden. Der Benutzer kann diese Farbe über eine Inline-Palette ändern.
- Keine Undo/Redo-Funktion implementieren.

Kontext:
- Der Konstruktor richtet eine Tabelle mit den Spalten „Name“, „Menge“, „L“, „B“, „H“, „Gewicht“ und „Farbe“ ein. Die Farbspalte zeigt einen Farbbutton.
- Definiere ein Qt-Signal `boxes_created` vom Typ `Signal(list)`. Dieses wird ausgelöst, wenn der Benutzer Boxen generieren möchte (z. B. durch Klick auf einen externen Button). Der Slot liest alle gültigen Zeilen aus, erzeugt entsprechend viele `Box`-Objekte mit Startposition im Wartebereich (0/0) und Rotation 0° sowie der gewählten Farbe und gibt sie als Liste zurück.
- Implementiere eine Methode `read_boxes(stacked: bool) -> list[Box]`, die die Tabelle ausliest und Boxen erstellt. Wenn `stacked=True`, soll pro Typ nur ein Exemplar erzeugt werden; die Stückzahl bestimmt die Stapelhöhe.
- Stelle sicher, dass Farben im GUI und im PDF identisch sind.
- Nutze Signale/Slots für Interaktionen; blockiere die GUI nicht bei Validierung.

—----------

Pfad: src/container_tool/gui/canvas_2d.py

Ziel: Implementiere `canvas_2d.py`, das für die maßstabsgetreue Darstellung des Containers und des Wartebereichs verantwortlich ist. Dieses Modul soll Drag-&-Drop, Live-Kollisionen, manuelles Stapeln und Zoom unterstützen.

Rückgabeformat: Nur der vollständige Python-Code.

Einschränkungen:

- Verwende PySide6 (`QGraphicsView`, `QGraphicsScene`, `QGraphicsRectItem`, `QWheelEvent`, `QMouseEvent`, `QTransform`, `QPen`, `QBrush`).
- Halte die Render-Performance von ≥ 25 fps ein. Nutze Double-Buffering und vermeide unnötige Neuzeichnungen.
- Zoomstufen: 25 %, 50 %, 100 % und 200 %, schaltbar über Ctrl + Mausrad; Wartebereich und Container müssen synchron zoomen.
- Zeige Kisten/Stapel als farbige, beschriftete Rechtecke im exakten Maßstab. Die Beschriftung enthält Name, Grundmaße, Menge und Gesamthöhe.
- Kollisionen werden mittels `check_collisions` aus `core/collision.py` geprüft; bei Verstoß wird das Objekt rot dargestellt und verbleibt an seinem Platz.
- Manuelles Stapeln: Wird ein Objekt auf ein anderes bewegt und die Snap-Toleranz ±10 mm erfüllt, rufe `create_stack` aus `core/stack.py` auf, ersetze beide Objekte durch den neuen Stack und setze die oberste Box an die markierte Position.
- Bei Drag-&-Drop zwischen Wartebereich und Container müssen die Objekte entsprechend der Zone umgeschichtet werden; außerhalb liegende Objekte sind rot zu markieren und dürfen nicht abgelegt werden.

Kontext:
- Erzeuge zwei `QGraphicsScene`-Instanzen: eine für den Wartebereich (Ablagezone) und eine für den Container. Jede Szene kennt die jeweilige Skalierung (mm→Pixel).
- Implementiere `on_drag_move(event: QMouseEvent)` zur Live-Kollisionserkennung; `mousePressEvent` zum Start des Dragging und `mouseReleaseEvent` zum finalen Platzieren.
- Implementiere einen Zoom-Handler, der auf `wheelEvent` reagiert und beide Szenen proportional skaliert.
- Verwende Signale, um Änderungen an andere Komponenten zu melden (z. B. Aktualisierung der PDF-Ansichten).

—----------

Pfad: src/container_tool/gui/window.py

Ziel: Baue in `window.py` das Hauptfenster des Container-Ladetools. Dieses verbindet Tabelle, Wartebereich, Container-Canvas und Buttons zu einer kohärenten GUI.

Rückgabeformat: Nur der vollständige Python-Code.

Einschränkungen:
- Nutze PySide6 (`QMainWindow`, `QWidget`, `QHBoxLayout`, `QVBoxLayout`, `QSplitter`, `QPushButton`, `QComboBox`, `QFileDialog`, `QMessageBox`, `QStatusBar`, `QToolBar`).
- Die GUI darf nicht einfrieren; lange Operationen (z. B. PDF-Export, Lade/Speicher-Vorgänge) sollen in separaten Threads mittels `QThread` oder `concurrent.futures` ausgeführt werden, und Fortschrittsanzeigen im UI-Thread aktualisieren.
- Es gibt genau sechs Buttons: Containerauswahl (Dropdown oder Button-Gruppe), Kisten gestapelt erstellen, Kisten einzeln erstellen, Projekt laden, Projekt speichern, Projekt abgeschlossen/Daten exportieren.

Kontext:
- Im Konstruktor werden `TableWidget`, `Canvas2D` (mit zwei Szenen) und die Buttons instanziiert. Verwende einen `QSplitter`, um Tabelle und Grafikbereiche flexibel in der Größe anzupassen.
- Containerauswahl: Biete eine Auswahl der vier definierten Containertypen; nach Auswahl wird die Container-Szene mit den entsprechenden Maßen neu skaliert.
- Die Buttons „gestapelt erstellen“ und „einzeln erstellen“ rufen die Methode `read_boxes()` des TableWidgets auf und übergeben das Ergebnis an die Canvas-Wartebereichs-Szene; bei erneutem Klick werden vorhandene Boxen gelöscht und neu generiert.
- „Projekt laden“ ruft `io_clp.load_clp()` auf und lädt Container und Boxen/Stapel in GUI und Tabelle. Neue Box-Typen können ergänzt und erneut generiert werden.
- „Projekt speichern“ ruft `io_clp.save_clp()` auf und speichert den aktuellen Zustand samt Meta-Informationen.
- „Projekt abgeschlossen / Daten exportieren“ erzeugt ein PDF über `export_pdf()` und zeigt bei Erfolg einen Dialog.
- Richte ein Logging ein: Verwende `logging.getLogger(__name__)` und RotatingFileHandler (`logs/error.log`, 7-Tage-Retention).
- Definiere eine Statusleiste, die aktuelle Aktionen (z. B. Anzahl der platzierten Boxen, Warnungen) anzeigt.

—----------

Pfad: src/container_tool/export/pdf_export.py
Ziel: Entwickle in `pdf_export.py` eine Funktion zum Erstellen eines umfassenden PDF-Exports des aktuellen Projekts. Das PDF enthält 2D-Ansichten, eine 3D-Ansicht und Tabellen.

Rückgabeformat: Gib nur den finalen Python-Code zurück.

Einschränkungen:
- Nutze `reportlab` (z. B. `canvas.Canvas`, `ImageReader`, `Table`, `TableStyle`, `colors`) sowie optional `Pillow` zur Bildverarbeitung; keine weiteren Bibliotheken.
- Die PDF-Größe ist A4 hoch mit 20 mm Rand; Bilder werden in 200 dpi gerendert (800 × 600 px).
- Die Erstellung muss bei maximaler Projektgröße (200 Boxen / 40 Typen) in ≤ 5 s erfolgen. Verwende daher vorberechnete Bilder.
- Beschriftungen und Farben müssen den GUI-Ansichten entsprechen.

Kontext:
- Implementiere `export_pdf(project: Project, path: str) -> None`. Diese Funktion erzeugt eine neue PDF-Datei an `path`.
- Erzeuge zunächst zwei 2D-Bilder: Draufsicht (Top-View) und Seitenansicht (Side-View). Nutze Methoden des Canvas oder exportiere die `QGraphicsScene` als Bild (z. B. über `QImage` → `ImageReader`).
- Rufe `render_3d.render_scene(project) -> PIL.Image` auf, um eine 3D-Visualisierung zu erhalten. Füge alle drei Bilder mit Beschriftungen in das PDF ein.

- Generiere zwei Tabellen: eine mit den verladenen Kisten (Name, Anzahl, Länge, Breite, Höhe, Gewicht) und eine mit nicht verladenen Kisten (im Wartebereich). Verwende `reportlab.platypus.Table` und wende z. B. eine abwechselnde Hintergrundfarbe an.
- Füge Kopf- und Fußzeile hinzu (Projektname, Datum/Zeit, Version). Halte den Stil schlicht und professionell; wähle eine gut lesbare Schrift (z. B. Helvetica).

—----------

Pfad: src/container_tool/export/render_3d.py

Ziel: Implementiere `render_3d.py`. Dieses Modul erzeugt eine 3D-Darstellung des beladenen Containers.

Rückgabeformat: Nur der vollständige Python-Code.

Einschränkungen:
- Nutze `PyOpenGL` (GL, GLU) und `numpy`. Optional kannst du `Pillow` verwenden, um das gerenderte Bild als `PIL.Image` zurückzugeben.
- Rendering muss offline funktionieren (keine externe GPU-Abhängigkeit) und darf die GUI nicht blockieren; führe das Rendern gegebenenfalls in einem separaten Thread aus.
- Verwende perspektivische Projektion und einfache Lichtquellen; achte darauf, dass Farben der Boxen korrekt dargestellt werden und farbenblind-freundlich bleiben.
- Die Bildauflösung beträgt 800 × 600 Pixel.

Kontext:
- Implementiere eine Funktion `render_scene(project: Project) -> PIL.Image`, die die Containergeometrie und alle Boxen/Stapel in 3D zeichnet. Lege das Koordinatensystem so an, dass die X-Achse der Länge, Y-Achse der Breite und Z-Achse der Höhe des Containers entspricht.
- Zeichne den Container als transparenten Quader mit Kanten und stelle die Boxen als opake Quader dar. Stapel werden mit korrekter Höhe gezeichnet.
- Platziere die Kamera so, dass sowohl Vorder- als auch Seitenansicht sichtbar werden (isometrische Perspektive).
- Implementiere Helferfunktionen zur Erstellung einfacher 3D-Primitive (Quader) und zur Farbumrechnung (Hex→RGB).
- Gib das Ergebnis als `PIL.Image` zurück; das PDF-Export-Modul konvertiert es für ReportLab.

—----------

Pfad: src/container_tool/main.py

Ziel: Erstelle die Einstiegspunkt-Datei `main.py`, die das gesamte Programm startet.

Rückgabeformat: Liefere nur den vollständigen Python-Code.

Einschränkungen:
- Nutze keine globalen Variablen außer dem Logger; kapsle Logik in Funktionen.
- Das Programm darf nur offline laufen; es gibt keinen Update-Mechanismus außer dem Austausch der EXE.
- Initialisiere die Rotating-Logdatei `logs/error.log` (Retentionszeit 7 Tage).
- Die Datei muss PyInstaller-kompatibel sein (keine unbedingten relativen Imports, keine if __name__ == '__main__' Abfragen in anderen Modulen).

Kontext:
- Importiere `QApplication` von PySide6 und setze `HighDpiScaleFactorRoundingPolicy` für bessere Skalierung.
- Lade Containerdefinitionen via `io_clp.load_containers_definitions()`. Wähle standardmäßig den ersten Container (z. B. 40ft-Standard) oder öffne einen Dialog in `window.py`.
- Instanziiere das Hauptfenster `MainWindow` und übergib die Containerdefinitionen.
- Registriere einen globalen Exception-Hook: Bei unbehandelten Exceptions schreibe den Stacktrace in das Log und zeige eine Fehlermeldung.
- Starte das Qt-Event-Loop mit `sys.exit(app.exec())`.

—----------
Pfad: tests/test_models.py

Ziel: Schreibe Unit-Tests in `test_models.py`, die die Datenmodelle aus `models.py` validieren.

Rückgabeformat: Gib ausschließlich den ausführbaren Python-Testcode zurück (mit pytest).

Einschränkungen:
- Verwende `pytest` und die in `models.py` definierten Klassen. Keine externen Bibliotheken.
- Definiere mehrere Testfunktionen; wähle aussagekräftige Namen.
- Tests dürfen nicht länger als einige Millisekunden laufen.

Kontext:
- Teste, ob `Box.to_dict()` und `Box.from_dict()` einen Round-Trip erlauben (Serialisierung und Deserialisierung ergeben gleichwertige Objekte).
- Prüfe `bbox()` sowohl für Rotation 0° als auch 90°.
- Prüfe, dass `Stack.total_height_mm` der Summe der enthaltenen Boxhöhen entspricht.
- Prüfe, dass `Project` die Gesamtzahl und das Gesamtgewicht der Boxen korrekt berechnet.
- Verwende Fixtures, um Beispiel-Boxen und Container anzulegen; teste Edge-Cases wie fehlendes `weight_kg`.

—----------

Pfad: tests/test_collision.py

Ziel: Implementiere in `test_collision.py` Unit-Tests für die Kollisionslogik.

Rückgabeformat: Nur ausführbarer pytest-Code.

Einschränkungen:
- Verwende die Funktionen aus `collision.py` und die Modelle aus `models.py`.
- Führe keine aufwendigen Renderings durch.

Kontext:
- Schreibe Tests für `is_within_container()`: ein Objekt, dessen Bounding-Box die Containergrenzen überschreitet, wird korrekt als außerhalb erkannt.
- Teste `check_collisions()` mit überlappenden Boxen und erwarte, dass die Liste kollidierender Objekte zurückgegeben wird. Überprüfe, dass sich nicht kollidierende Boxen gleichzeitig platzieren lassen.
- Teste die Türhöhenprüfung, indem ein Stapel erzeugt wird, der die `door_height_mm` überschreitet; `check_collisions()` soll den entsprechenden Fehler signalisieren.
- Verwende parametrisierte Tests und Pytest-Fixtures.

—----------

Pfad: tests/test_stack.py

Ziel: Verfasse Tests in `test_stack.py`, die die Stapel-Logik validieren.

Rückgabeformat: Vollständiger pytest-Code ohne Erklärungen.

Einschränkungen:
- Importiere `Stack`, `Box`, `Container` und die Funktionen aus `stack.py`.
- Kein GUI-Code.

Kontext:
- Teste `can_stack()` mit kompatiblen Boxen (identische Maße, Snap-Toleranz eingehalten) und nicht kompatiblen (abweichende Maße, Rotation oder zu großer Abstand).
- Prüfe, dass `create_stack()` die Höhe korrekt summiert und die Position unverändert lässt; bei Überschreiten der Türhöhe muss eine Exception auftreten.
- Teste, dass `add_to_stack()` einen bestehenden Stapel erweitert und die Boxanzahl erhöht.
- Verwende Fixtures mit Container-Definitionen.

—----------

Pfad: tests/test_io.py

Ziel: Implementiere Tests in `test_io.py` für das Ein- und Auslesen von `.clp`-Dateien.

Rückgabeformat: Ausschließlich pytest-Code.

Einschränkungen:
- Tests müssen offline lauffähig sein; keine Abhängigkeit von vorhandenen Dateien außerhalb des Testverzeichnisses.
- Nutze temporäre Dateien (`tmp_path`-Fixture von pytest) und Fixtures für Boxen und Container.

Kontext:
- Schreibe einen Test `test_round_trip_clp()`, der ein Projekt mit mehreren Boxen/Stapel erstellt, mittels `save_clp()` speichert und anschließend mit `load_clp()` wieder lädt. Vergleiche, ob alle Attribute übereinstimmen.
- Teste das Laden einer `.clp`-Datei mit unbekanntem Containertyp und erwarte eine Exception.
- Teste Fehlerbedingungen wie unlesbare Dateien (PermissionError) und invalide JSON-Strukturen.
- Stelle sicher, dass relative Pfade korrekt verarbeitet werden.

—----------

Pfad: tests/test_performance.py

Ziel: Schreibe `test_performance.py`, um die Performance-Anforderungen automatisiert zu überprüfen.

Rückgabeformat: Nur pytest-Code unter Verwendung von `pytest-benchmark`.

Einschränkungen:
- Verwende `pytest-benchmark` als Fixture (Benchmark-Framework muss im Projekt installiert sein).
- Keine aufwendigen GUI-Operationen; konzentriere dich auf Kernfunktionen.

Kontext:
- Erzeuge in einem Fixture 200 Boxen mit 40 unterschiedlichen Typen sowie einen Container der Größe 40 ft.
- Messe die Laufzeit von `check_collisions()` beim Platzieren eines neuen Boxenobjekts und verifiziere, dass der Median unter 50 ms liegt.
- Optional: Messe die Zoom-Performance, indem du die Skalierung des Canvas 100-mal änderst und sicherstellst, dass die FPS ≥ 25 und die Latenz < 100 ms liegt. Diese Tests dürfen einen gewissen Toleranzbereich haben.
- Markiere Tests mit `@pytest.mark.benchmark` und liefere klare Assertion-Grenzen.

—----------

Pfad: tests/smoke_gui.py

Ziel: Implementiere `smoke_gui.py` für einen einfachen GUI-Smoke-Test mithilfe von `pytest-qt`.

Rückgabeformat: Nur pytest-Testcode.

Einschränkungen:
- Verwende das Plugin `pytest-qt` (`qtbot`-Fixture) und importiere `MainWindow` aus `gui/window.py`.
- Tests sollen nicht mehr als ein paar Sekunden dauern.

Kontext:
- Teste, ob das Hauptfenster ohne Fehler gestartet werden kann: rufe `qtbot.show()` auf dem MainWindow auf und prüfe, dass es sichtbar ist.
- Simuliere das Hinzufügen von zwei Boxen über das TableWidget: Fülle Felder (Name, Menge, Maße, Gewicht), triggere den „einzeln erstellen“-Button und verifiziere, dass `boxes_created`-Signal ausgelöst wird.
- Prüfe, ob per Containerauswahl ein anderer Container geladen wird und die Szenengröße angepasst wird.
- Stelle sicher, dass das PDF-Export-Popup (z. B. QFileDialog) geöffnet wird, wenn der entsprechende Button angeklickt wird; breche vor dem Schreiben ab, um Zeit zu sparen.

—----------

Pfad: logs/error.log
Ziel: Lege die Datei `logs/error.log` als leere Datei an. Während der Laufzeit werden Exceptions durch RotatingFileHandler in diese Datei geschrieben. Der Inhalt wird durch das Programm generiert; es muss hier kein Platzhaltercode enthalten sein.

Rückgabeformat: Gib einfach eine leere Datei ohne Inhalt zurück.

Einschränkungen:
- Es dürfen keine Zeilen in dieser Datei stehen; die Log-Rotation erfolgt zur Laufzeit.