

Software Design

Julian Huber & Matthias Panny

Ein erster Prototyp unserer Case Study

- Die Entwicklung einer neuen Software stellt uns gleichzeitig vor mehrere Herausforderungen:
 - Welche *Technologien* und *Frameworks* setzen wir ein?
 - Wie funktionieren diese?
 - Welche Architektur lässt sich möglichst *schnell umsetzen*?
 - Welche Architektur ist auch *langfristig wartbar*?
- Meist ist es schwierig im Voraus einen guten Master-Plan zu entwickeln

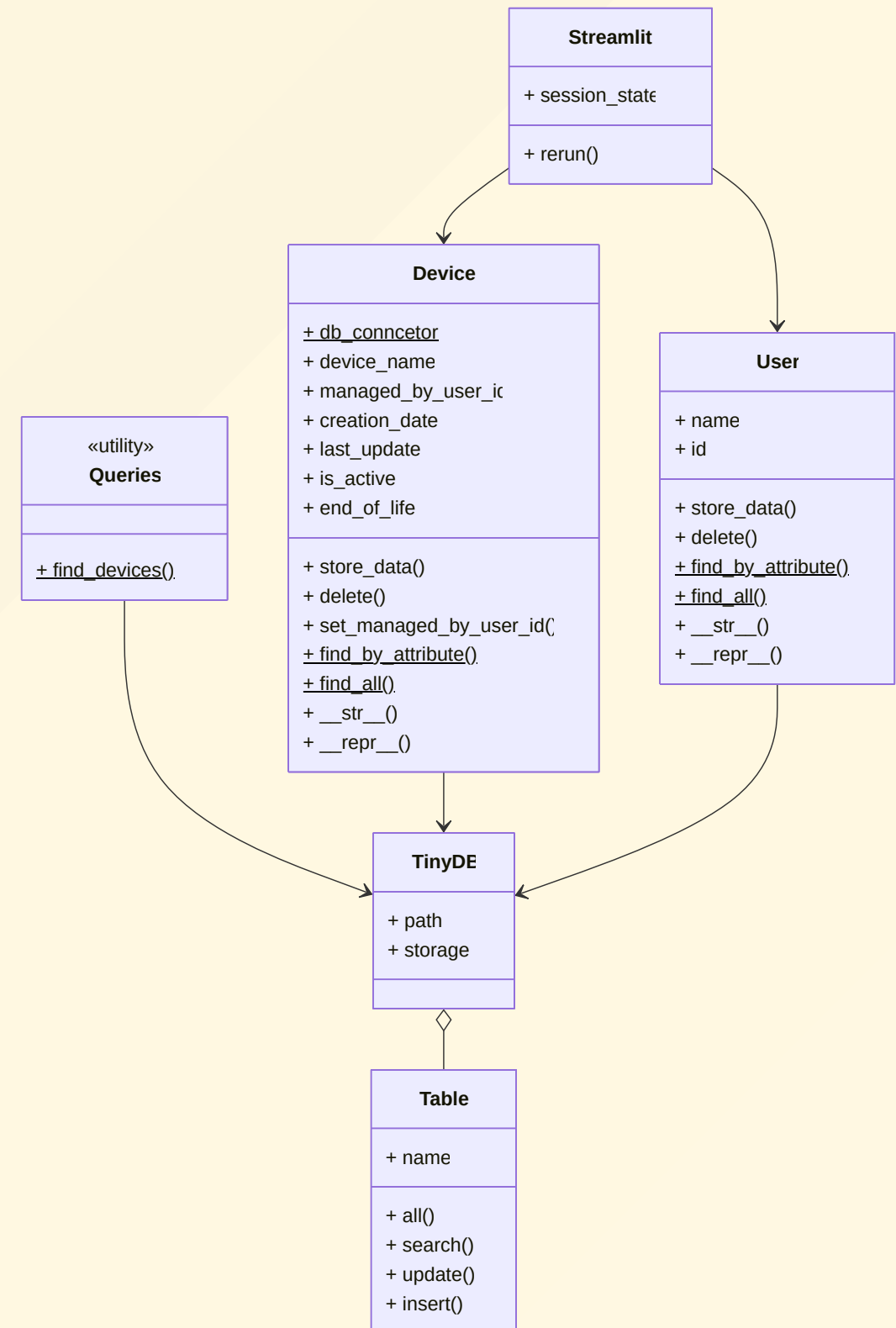
Lösung: Iteratives Vorgehen

- Meist ist die Entwicklung ein **Lernprozess** → unser Wissensstand ist zu Beginn nicht vollumfänglich
- Wir Lösen das Problem in mehreren Entwicklungsiterationen
- Resultat: **Exploration-Exploitation-Dilemma**
 - *Exploration*: Wir entwickeln (suboptimale) Prototypen, um Lösungen zu erarbeiten → Analogie: Greedy-Algorithmus
 - *Exploitation*: Wir verbessern die Lösungen, um die Stabilität und Wartbarkeit des Codes zu sichern → Analogie: Globale Optimierung
 - Dilemma: Wir müssen uns entscheiden, wann wir von Exploration zu Exploitation übergehen

- Wir wollen uns den Lösungsvorschlag aus dem Examples-Ordner genauer ansehen
- Hierzu nutzen wir wieder die Werkzeuge aus den Kapiteln
- 01_06_Objektorientierung_II → Klassen-Diagramme
- 03_01_Activity_Diagramme → Activity-Diagramme

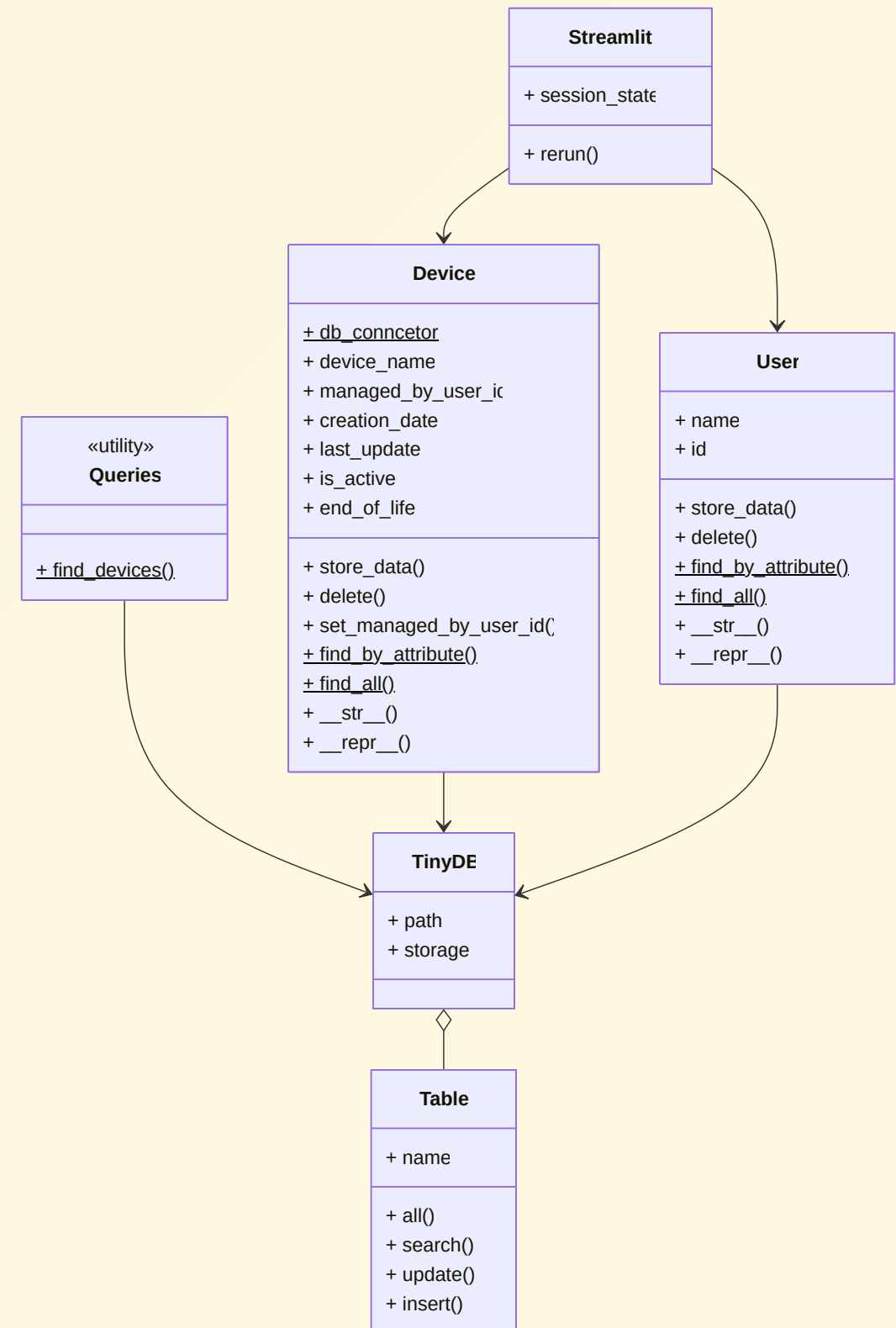
Grundarchitektur

- Eine Aufteilung in Schichten ist sinnvoll
 - **Streamlit** als User Interface
 - verschiedene Objekte in der Business Logik
 - **TinyDB** in der Datenbank
- Offensichtlich sind **User** und **Device** sinnvolle Klassen
- Es gibt verschiedene Anfragen an die Datenbank (**Query**)



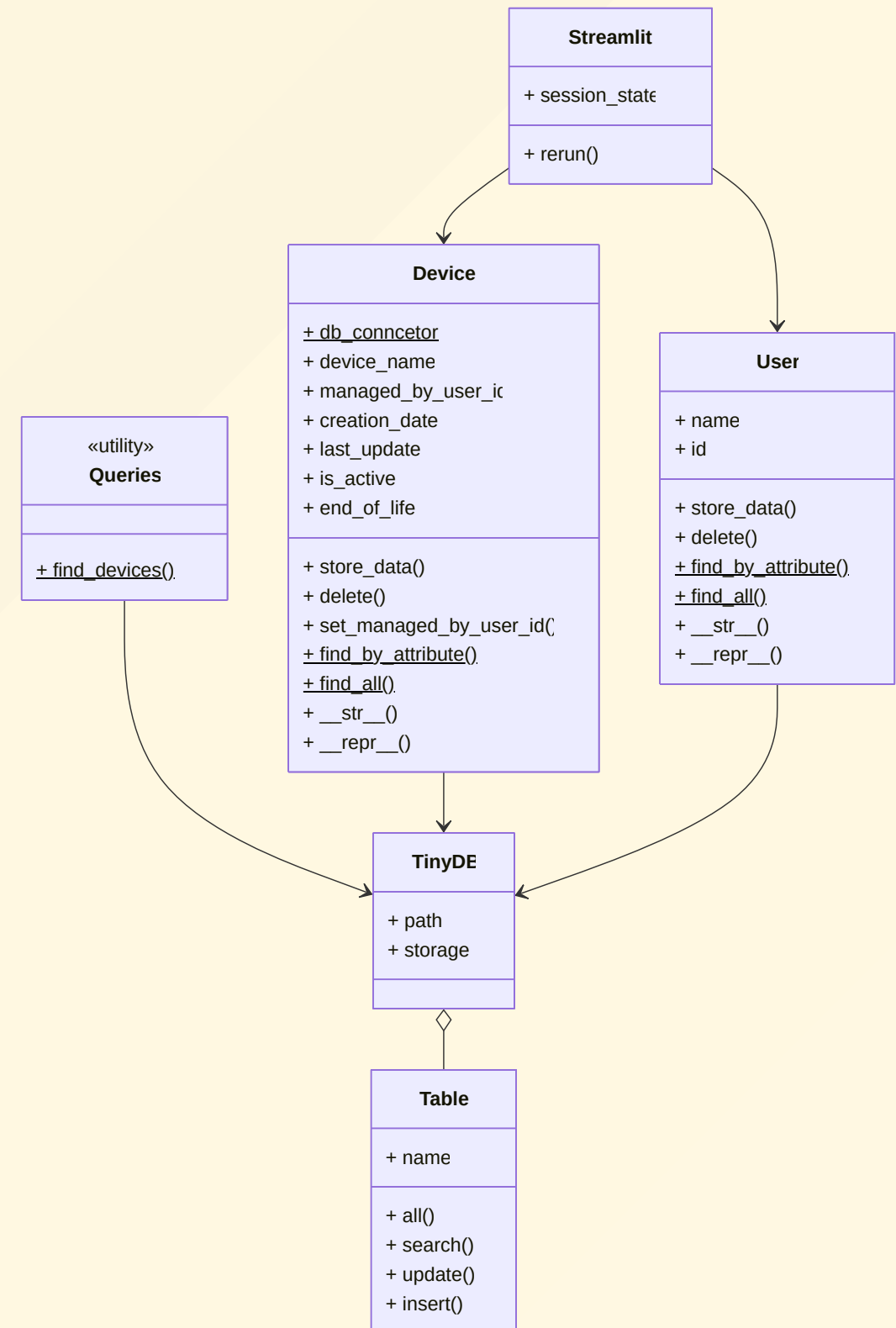
Streamlit

- Das UI speichert die Variablen, welche die aktuelle Anzeige bestimmen (mit oder ohne **session_state**), diese könne einzelne Attribute von **User** oder **Device** oder auch die Objekte selbst sein
- Nach jeder Interaktion mit Streamlit wird ein **rerun()** durchgeführt, was bedeutet, dass alle Methoden und Funktionsaufrufe im Skript ausgeführt werden



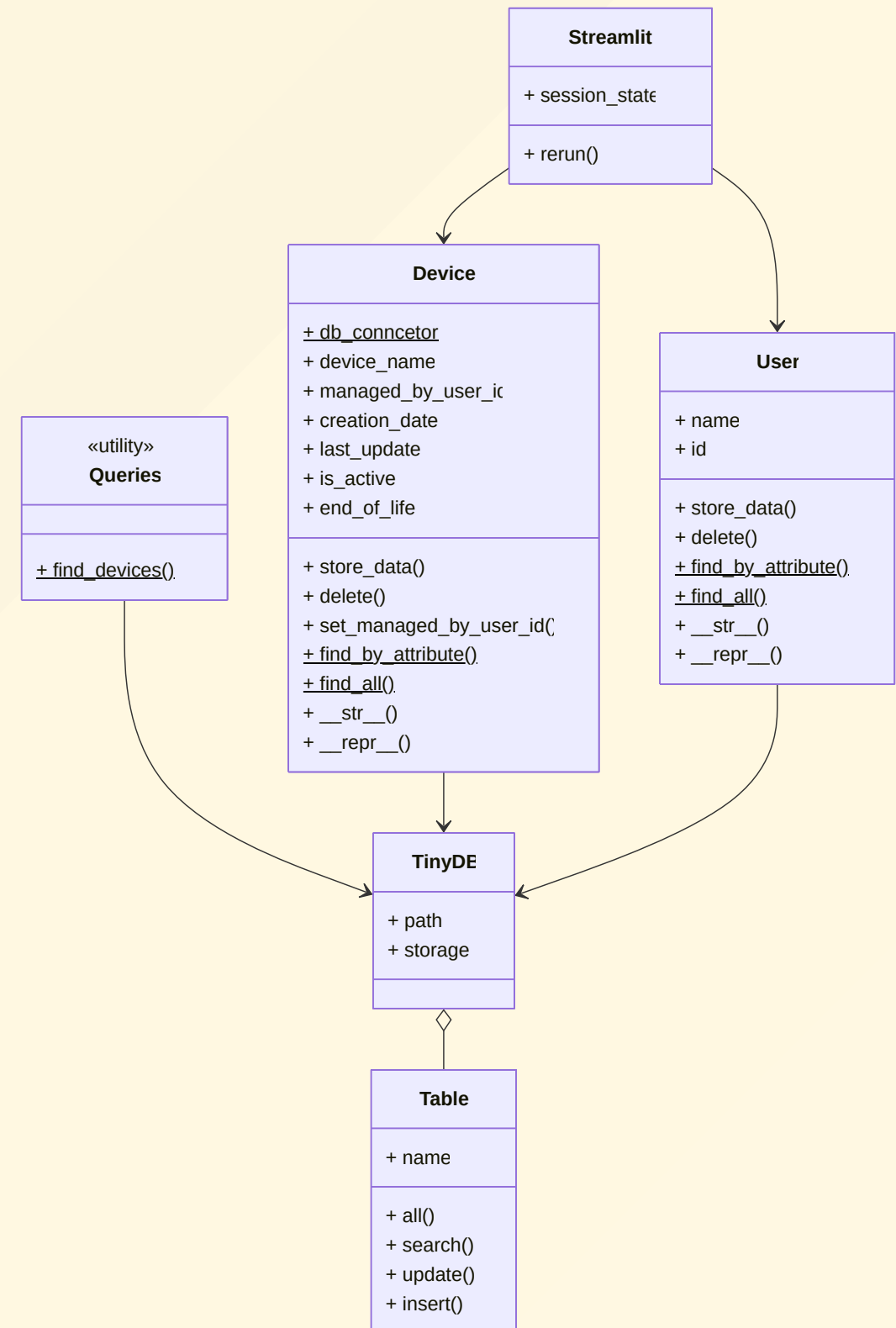
Anbindung der Datenbank über TinyDB

- Vereinfachte Darstellung TinyDB-Dokumentation
- Das Objekt zur Datenbankverbindung beschreibt einen **path** zur JSON-Datei, die die Daten speichert
- Eine Datenbank kann mehrere **Tables**, bei uns für unterschiedliche Daten aus der gleichen Klasse enthalten
- das **storage**-Objekt beschreibt, wie die Objekte in der JSON-Datei repräsentiert (serialisiert und de-serialisiert) werden



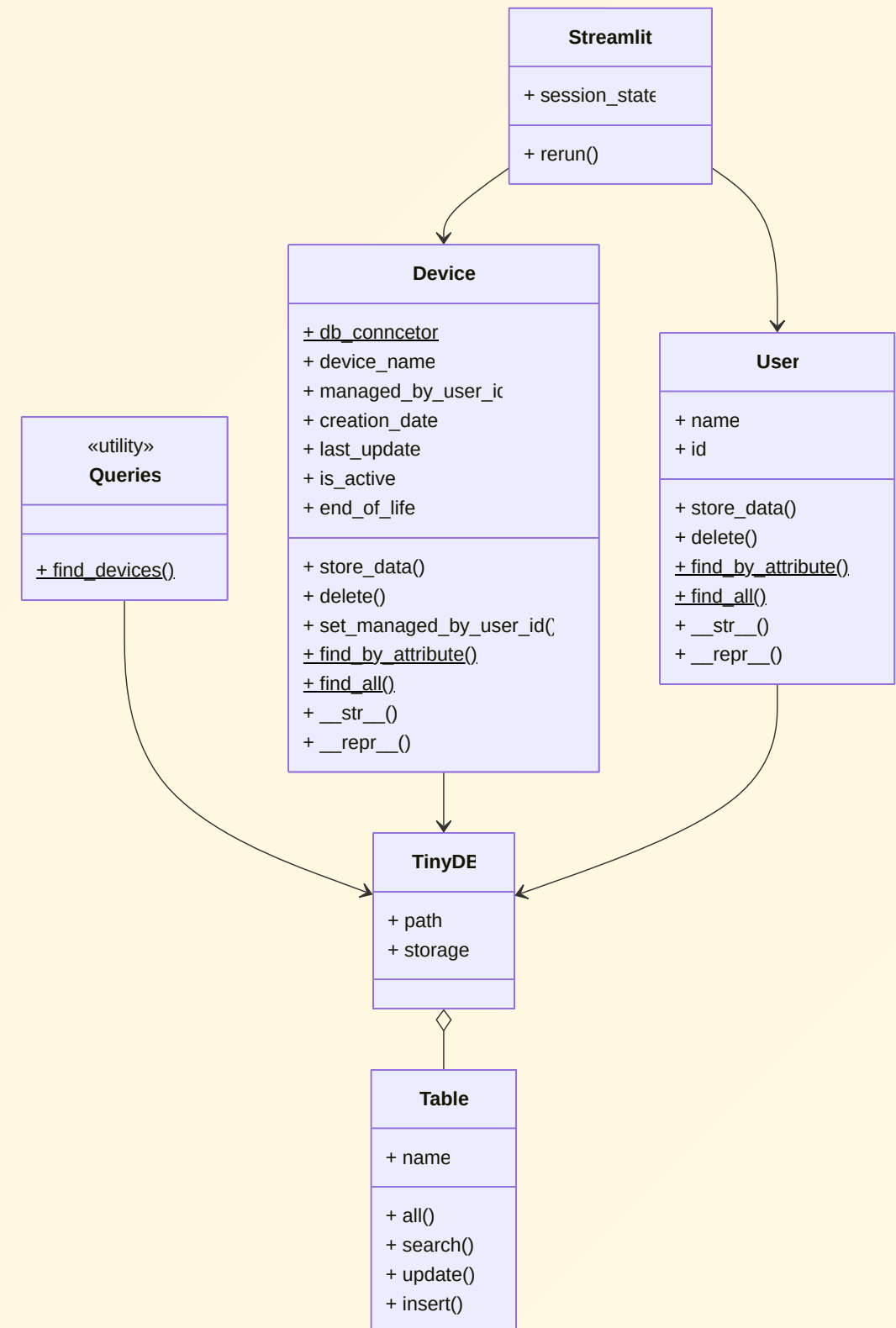
Interaktionen mit der Datenbank über **Tables**

- **all()** gibt alle Einträge in einer **table** der Datenbank zurück
- **search()** durchsucht eine Tabelle nach Einträgen mit einem bestimmten Suchmuster (**Query**)
- **insert()** fügt einen neuen Eintrag in eine Tabelle ein
- **update()** verändert einen bestehenden Eintrag



User & Device

- damit die Objekte gespeichert werden können (**store_data()**) wird eine Verbindung zur Datenbank benötigt (**db_connector**) als Attribut
- Die Klasse **TinyDB** müssen wir nicht selbst implementieren, sondern können auf die Funktionalität von **TinyDB** zurückgreifen



Aufgabe

- Finden Sie das Attribut `db_connector` in der Klasse `Device`
- Was fällt uns bei der Positionierung und Syntax des Attributs im Bezug auf Klasse und Methoden auf?
- Diese Art von Attribut haben wir bereits in einer anderen Einheit diskutiert. erinnert sich noch jemand?

Class Attributes

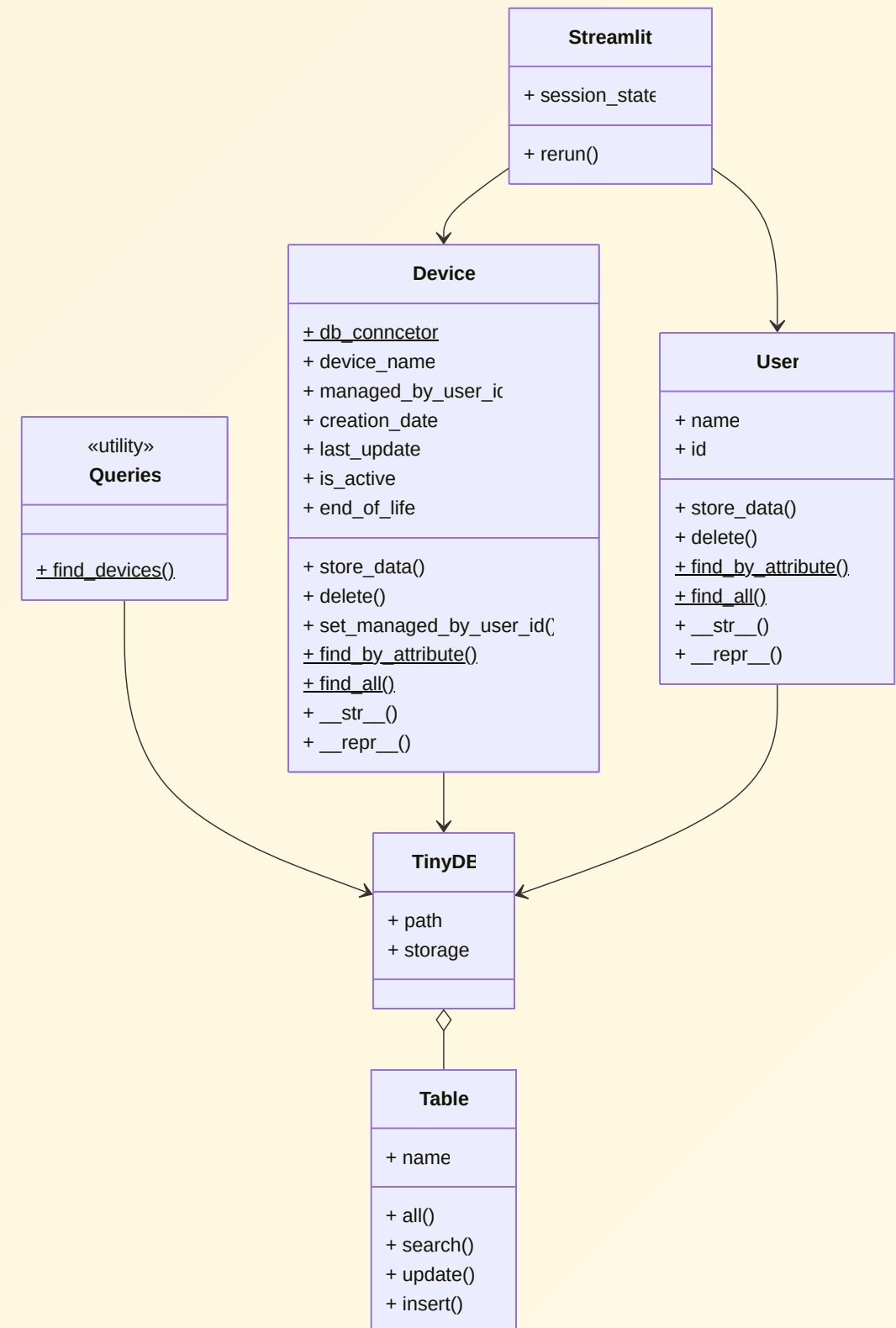
- bei `db_connector` (ohne `self`) handelt es sich um ein sogenanntes **Class Attribute**. Diese hängt nicht an der einzelnen Instanz (Objekt) sondern an der Klasse selbst
- Diese bieten sich in vielen Fällen an:
 - Default-Werte: Jedes Gerät soll die gleiche Standard-Datenbank nutzen
 - Informationsaustausch zwischen Instanzen: Alle Geräte sollen immer in die gleiche Datenbank geschrieben werden
 - **Singletons** (dazu später mehr): Klassen, von denen nur eine Instanz geschaffen werden soll

User & Device

- Wir müssen an mehreren Stellen mit der Datenbank interagieren, um neue Geräte anzuzeigen

1 Wir müssen die Namen aller Geräte finden, um diese im User Interface anzuzeigen

2 Sobald der User ein Gerät auswählt, müssen wir dieses in den Speicher laden, um alle Information anzuzeigen



Aufgabe

- Durch welche Klassen und Methoden werden **1** und **2** ermöglicht
 - im UML-Diagramm und Code gibt es zwei Stellen die **1** ermöglichen
 - Was sind die Unterschiede zwischen den beiden Methoden?
 - Was sind die Vor- und Nachteile der beiden Methoden?

Direkter Abruf der Liste aller Gerätenamen aus der Datenbank

- wird in `queries.find_devices()` implementiert und gibt eine Liste aller Gerätenamen in der Datenbank zurück. Hierbei wurde auf eine objektorientierte Implementierung verzichtet.
- Wir sparen uns Objekte vom Typ `Device`, sondern greifen direkt auf die Datenbank zu.
- Vorteile:
 - Einfachheit
 - Performance
- Nachteile:
 - Wir können keine Methoden eines nicht vorhandenen Objekts aufrufen
 - Verletzung des Prinzips der Schichtentrennung

Objektorientierte Lösung über Class Methods

- zudem werden die Gerätenamen auch in `Device.find_all()` geladen
- dies wirkt zunächst nicht intuitiv, da wir vor der Suche noch kein Objekt vom Typ `Device` im Speicher haben
- Um dies zu Umgehen wird eine `Class Method` verwendet, die es uns erlaubt, auf die Methode `Device.find_all()` zuzugreifen, ohne dass wird zuvor ein Objekt vom Typ `Device` zu erstellen

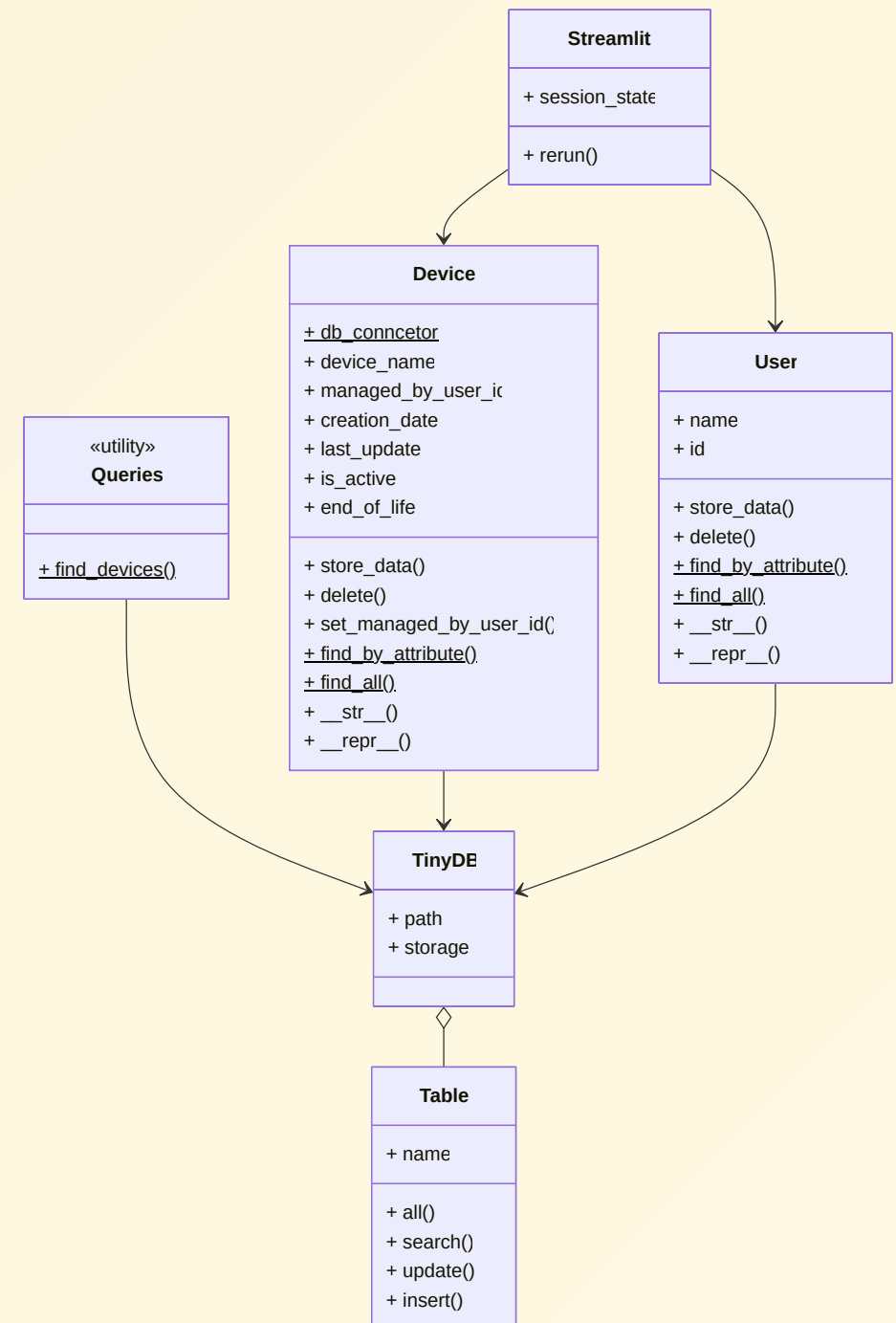
```
@classmethod  
def find_all(cls):  
    pass
```

- Analog zum `Class Attribute` können wir eine `Class Method` implementieren, die es uns erlaubt das Objekt aus der Datenbank zu instanziiieren.

- Vorteile:
 - Wir finden den Code in dem Modul, in dem wir ihn erwarten
 - Wir können Methoden eines Objekts aufrufen, z.B. um mehr als nur den Namen des Geräts zu laden
- Nachteile:
 - Komplexität
 - Performance, da wird erst alle möglichen Geräte laden, obwohl wir nur die Namen benötigen

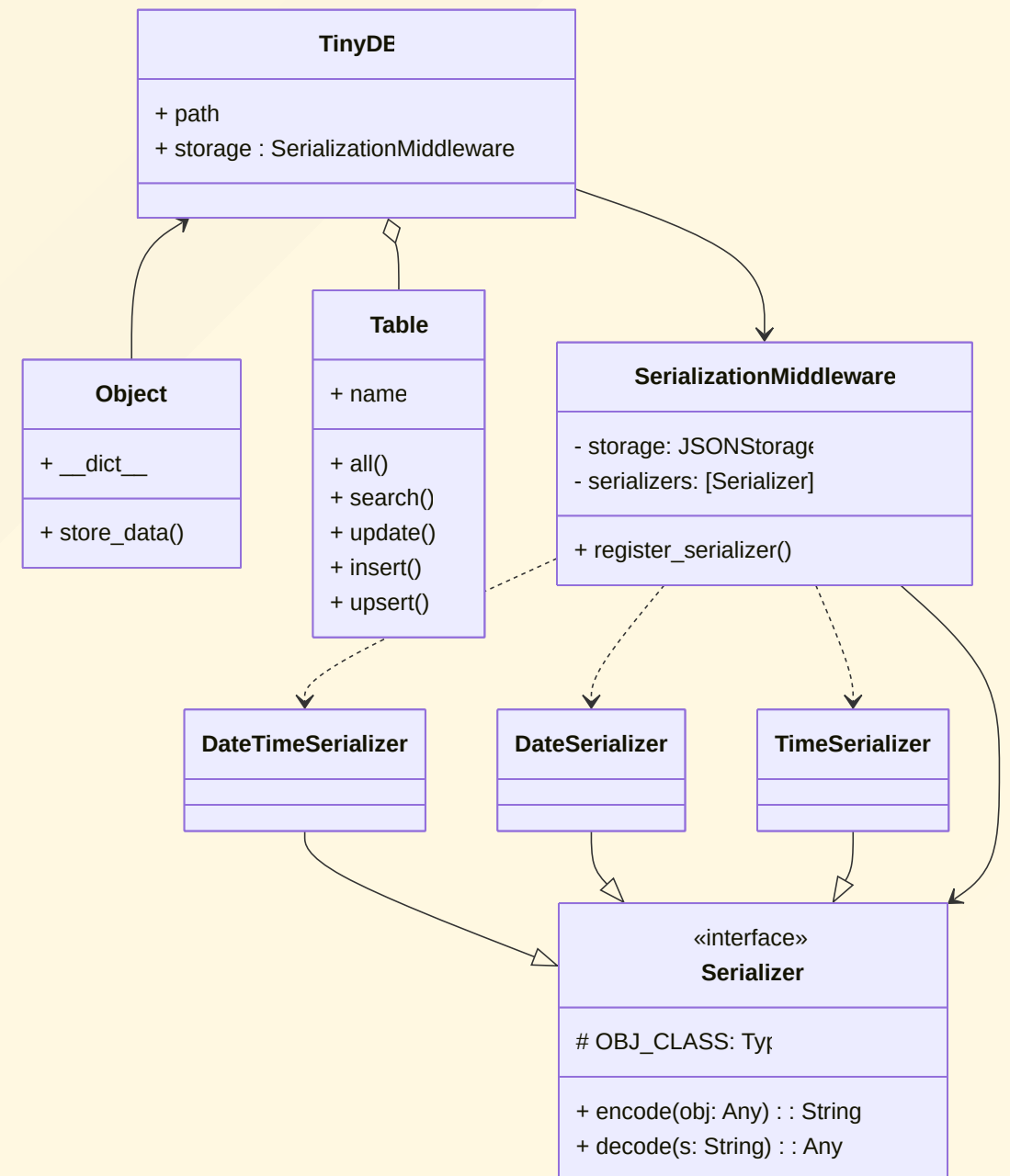
User & Device

- Bei genauerem Hinsehen finden wir viele Parallelen zwischen **User** und **Device**, die nur teilweise im Code abgedeckt sind.
- Beide verfügen über eine eindeutige ID
- Beide können über die ID geladen werden
- Beide können gespeichert werden
- Für beide können wir alle Einträge aus der Datenbank laden
- Aktuell können nur User gelöscht werden, dies könnte aber für beide sinnvoll sein



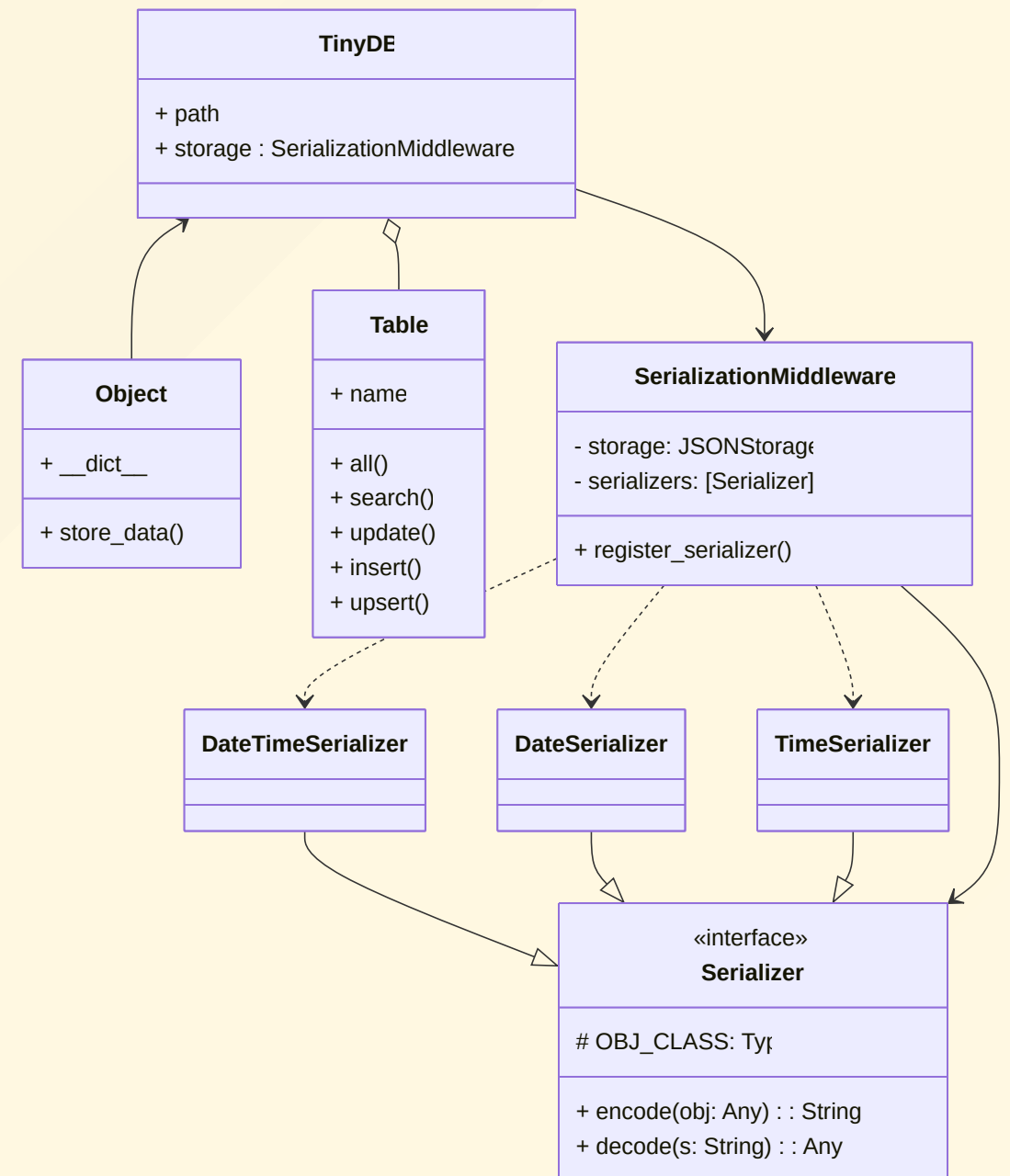
Allgemeine Architektur

- Beliebige Objekte können mittels **tinydb** einfach in einer **JSON**-Datei gespeichert werden
- Hierbei wird die **__dict__**-Darstellung des Objekts genutzt
- Hierzu nutzen wir z.B. die **upsert()**-Methode indem wir diese in der **store_data()** unseres Objekts aufrufen
- Dafür müssen wir eine Datenbankverbindung aufbauen, indem wir uns auf ein **TinyDB**-Objekt beziehen und eine **Table** darin auswählen



Allgemeine Architektur

- **TinyDB** stellt die auf *Folie 9* angezeigten Methoden bereit
- Für primitive Datentypen erfolgt die Umwandlung von Attribut zu serialisierter JSON-Darstellung automatisch
- Für spezielle Datentypen wie **Date** und **DateTime** müssen wir **encode()** und **decode()** in einer **SerializationMiddleware** konfigurieren



Allgemeine Architektur

- Dies erfolgt, indem wir dem Objekt vom Typ **SerializationMiddleware** geeignete **Serializer** hinzufügen
- Die so konfigurierte **SerializationMiddleware** wird dann dem **TinyDB**-Objekt im Parameter **storage** übergeben

