Week Three: Designing with Purpose

Date: October 30th

🧱 Object-Oriented Programming (OOP)

OOP organize software around objects — bundles of data and behavior that modeled real-world things.

Its core pillars provide powerful benefits:

- **Encapsulation** → Keeps internal details private, exposing only what was necessary
  - ✅ Makes code easier to understand and safer to change
- **Inheritance** → Allows new classes to reuse and extend existing ones
  - ✅ Reduces duplication and promotes consistency

🧱 Object-Oriented Programming (OOP)

- **Polymorphism** → Lets different objects respond to the same interface in different ways
    - ✅ Enables flexibility and clean abstractions
- **Abstraction** → Hides complex implementation behind simple interfaces
    - ✅ Helps developers focus on what an object does, not how it does it

Together, these principles support modular, reusable, and maintainable systems.

🧩 UML: Unified Modeling Language

UML is used as a visual language for designing software systems. It helps sketch out how parts of the system relate before writing a single line of code.

It is especially useful for planning object-oriented systems, clarifying relationships, and communicating design decisions with others.

🛠️ Software Design: Modules and Interfaces

Good design makes software easier to understand, extend, and maintain.

- **Modules** are the core components of the system (e.g., animals, adoptions, users)
- **Interfaces** are the API endpoints that allow other parts of the system — like the frontend — to interact with those modules

Designing with clear boundaries between modules and interfaces helps isolate complexity and improve flexibility.

🔄 DRY: Don’t Repeat Yourself

- Repetition leads to bugs, inconsistencies, and wasted effort
- Instead of duplicating logic, shared functionality is abstracted into reusable functions or components
- Validation logic, UI elements, and utility functions are reused across modules

DRY code is easier to test, debug, and evolve. It reflects thoughtful, professional engineering.

🐾 HW4 Example: Modules and Interfaces

The backend for HW4 is organized into three distinct modules, each tied to a database table and exposed through a clean set of API endpoints:

🧱 Modules (Database Tables)

Table	Purpose
animals	stores data about each animal (e.g., name, type, age, traits)
adoptions	tracks adoption requests and their status
users	manages user credentials and profiles

Each table functions as a module.

🔌 Interfaces (API Endpoints)

Animals

Method	Endpoint
GET	/animals
GET	/animals/:id
POST	/animals
PATCH	/animals/:id
DELETE	/animals/:id
GET	/animals/search?traits=...

🔌 Interfaces (API Endpoints)

Adoptions

Method	Endpoint
POST	/adoptions
PATCH	/adoptions/:id
GET	/adoptions/:id

🔌 Interfaces (API Endpoints)

Users

Method	Endpoint
POST	/register
POST	/login

This separation was key to good design: each module encapsulate its own responsibilities, while the interfaces expose only what was necessary for other parts of the system to interact with it cleanly and predictably.