Week Six: Software Engineering in Practice

Date: November 20th

🧩 The Role of a SWE

- Core job: **problem solver**
- Engineers balance technical decisions with business needs, ensuring solutions are practical and cost-effective
- Principle: **Buy, don’t build** when possible → saves time, reduces risk, leverages proven tools
- Building custom solutions is reserved for cases where existing options cannot meet requirements
- A SWE must also evaluate trade-offs: cost, scalability, maintainability, and long-term impact

💡 Interpretation: Buy vs Build

Reusing proven solutions saves time and reduces risk.
Engineers should focus their effort on solving unique problems rather than reinventing existing tools.
This principle keeps teams focused on innovation instead of duplicating commodity features.

🏛️ Two Levels of Design

- **Architecture** → defines the high-level structure of the system
  - Modules, boundaries, communication flows, and scalability considerations
  - Decisions about distributed systems, databases, and integration points
  - Guides how teams collaborate by clarifying responsibilities
- **Design** → focuses on implementation details
  - Object-oriented programming, UML diagrams, API contracts, SOLID principles, refactoring practices
  - Ensures that architecture is realized in a maintainable and testable way
  - Provides the craftsmanship needed to make architecture practical

💡 Interpretation: Architecture vs Design

Architecture provides the overall structure, while design focuses on the details of implementation.
Both are necessary: architecture without design is incomplete, and design without architecture lacks direction.
Together they ensure systems are both coherent and maintainable.

📐 Design Patterns

- Documented solutions to recurring problems in software engineering
- Provide a shared vocabulary for teams, improving collaboration and communication
- Increase consistency across projects by applying proven approaches
- Not mandatory, but widely adopted in successful organizations to reduce complexity and speed up development

💡 Interpretation: Design Patterns

Design patterns capture proven solutions to recurring problems.
They are not strict rules but shared practices that make complex systems easier to build and maintain.
Patterns help teams avoid reinventing solutions and reduce misunderstandings.

🔢 Type Systems and Reliability

- A significant portion of software bugs stem from type mismatches and invalid data handling
- Strong type systems enforce rules about data representation and usage
- Compile-time type checking prevents invalid states before execution
- Reliability improves when systems catch errors early, reducing runtime failures and costly debugging
- Languages like TypeScript, Rust, and Java emphasize type safety to improve developer confidence

💡 Interpretation: Type Systems

Strong type systems prevent common errors before code runs.

They enforce consistency and reduce the risk of subtle bugs that are difficult to trace.

By catching issues early, type systems improve reliability and reduce maintenance costs.

🐾 HW4 Example: Animal Shelter API

Applying Week 6 Principles:

- **Architecture vs Design**
  The shelter system is divided into three modules:
  - **Animals** → manages individual animals and their characteristics such as name, age, species, and status (available, adopted, pending).
  - **Adoptions** → handles requests, approvals, and adoption records.
  - **Users** → manages registration, authentication, and user roles.
    Architecture defines these boundaries and how modules interact, while design specifies details like UML diagrams, OOP structures, and API endpoints.
    This separation ensures clarity at both the strategic and tactical levels.

🐾 HW4 Example: Animal Shelter API

Applying Week 6 Principles:

- **Type Systems**
  The API enforces strict type rules:
  - Age must be an integer, preventing invalid inputs like strings or decimals.
  - Status must be an enum (e.g., pending, approved, adopted, rejected), ensuring consistency across the system.
  - Species must be a controlled string or enum, reducing ambiguity and errors.
    These constraints catch errors early, preventing invalid data from entering the system and reducing runtime bugs.