Combined Presentation: Quality Software

Week One: Managing Complexity in Software Engineering

Date: October 16th

🧠 Software Engineering = Managing Complexity

Software engineering is about designing systems that can handle complexity over time. The most difficult challenges often come from unknown unknowns — problems or requirements that aren't visible until they cause issues. These hidden risks can derail projects if not addressed early.

Quality software anticipates change and uncertainty through thoughtful design, planning, and the use of appropriate tools. The earlier complexity is identified, the better the system can be structured to handle it.

📏 Rules of the Game

To build quality software, follow two essential principles:

- KISS (Keep It Simple, Stupid): Simplicity makes code easier to read, test, and maintain. Complexity increases the risk of bugs and slows down development. Simple systems are more robust and easier to evolve.

📏 Rules of the Game

- No Surprises (especially in teams): Your teammates should never be confused or caught off guard by your code. This means:
  - Clear structure and naming: Code should be self-explanatory and follow agreed conventions.
  - Consistent delivery: Push updates regularly, follow version control best practices, and avoid last-minute changes that disrupt others.
  - Predictable behavior: Code should do what it says it does — no hidden logic or unexpected side effects.
  - Team success = readable, predictable, and consistently delivered code.

🛠️ Tools for Managing Complexity

Software engineers rely on key tools and concepts to tame complexity:

Tool	Purpose
Software Design	Organizes code into modules and interfaces
Process	Guides development through planning, building, testing, and reviewing
High-Level Languages	Simplify development and allow developers to focus on solving problems

🐾 HW4 Example: Animal Shelter API

- **High-Level Language Use**: The API is built entirely in PHP, a high-level language that handles server-side logic and abstracts away low-level operations. JavaScript is used solely on the client side to access and interact with the API. SQL is used for structured data storage and querying.
- **KISS**: The implementation of each endpoint is kept simple and direct — each function does one thing clearly and predictably, making the code easy to read, debug, and extend.

Week Two: Milestones in Software Development

Date: October 23rd

📅 Milestones = Planning for Progress

Milestones are key checkpoints in a project that help you track progress and stay aligned. They break large goals into manageable pieces and give structure to your workflow.

Why milestones matter:

- They define what needs to be done and by when
- They help identify delays early
- They make steady progress visible
- They reduce last-minute stress

Even though the final deadline might be weeks away, setting earlier planned milestones helps you stay on track and avoid surprises.

🔁 Development Models: Waterfall → Spiral → Agile

Software development models have evolved in response to the growing complexity of projects and the need for flexibility:

Waterfall

- ✅ Clear documentation and planning
- ❌ Inflexible to change; problems often discovered too late
- 🔄 **Limitation**: Once a phase is complete, it's hard to go back — making it risky for dynamic projects

🔁 Development Models: Waterfall → Spiral → Agile

Spiral

- ✅ Better at managing uncertainty
- ❌ Complex and resource-intensive
- 🔄 **Evolution**: Recognized the need for revisiting earlier decisions and adapting as new information emerged

🔁 Development Models: Waterfall → Spiral → Agile

Agile

- ✅ Fast response to change, continuous delivery, customer involvement
- 🔄 **Result**: Agile became the dominant model because it's built for uncertainty, iteration, and real-world feedback

🧪 Ugly Languages Can Be Great

In software engineering, practicality beats purity. A language doesn’t need to be elegant — it needs to work.

Lesson: Use the language that solves the problem, integrates with your tools, and is supported by your environment. Beauty is optional — usefulness is not.

🐾 HW4 Example: Milestones in Action

Submission Deadline: November 7th, 2025

Even though the final deadline is November 7th, breaking the work into smaller milestones with planned dates are helping me stay on track and avoid last-minute stress.

🧩 Milestones in HW4

Milestone	What	When (Plan)
1	Create SQL database and access through PHP	October 26th, 2025
2	Create API endpoints (no authentication)	October 31st, 2025
3	Create API endpoints (with authentication)	November 2nd, 2025
4	Create frontend to access API endpoints	November 6th, 2025

🧪 "Ugly" Languages Used in HW4

- **PHP**: Often criticized for its quirks, but it’s easy to deploy and works well for server-side scripting.
- **SQL**: Verbose and rigid, but essential for defining and querying structured data in your database.
- **JavaScript**: Messy at times, but it’s the backbone of interactive web interfaces and frontend logic.

These languages may not be beautiful, but they’re reliable and practical.

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.

Week Four: Advanced Design Principles

Date: November 6th

🧱 Composition: The Fifth Pillar of OOP

Beyond encapsulation, inheritance, polymorphism, and abstraction, **composition** is often considered a fifth pillar of object-oriented programming.

Definition:
Composition means building complex objects by **combining simpler ones**. One class contains other classes as part of its state, modeling a **"has-a" relationship** (e.g., a Car *has an* Engine, an Animal *has traits*).

Benefits:

- ✅ Promotes flexibility and avoids rigid hierarchies
- ✅ Encourages modularity, reuse, and adaptability
- ✅ Preferred over inheritance for long-term scalability
- ✅ Behaviors can be combined like interchangeable building blocks

💡 Interpretation: Why Composition Matters

Inheritance can be useful, but it often locks systems into rigid structures.
Composition provides flexibility by allowing behaviors to be combined or replaced without disrupting the overall design.
This makes systems easier to adapt when requirements change.

🧩 Composition vs Inheritance

Aspect	Inheritance (*is-a*)	Composition (*has-a*)
Definition	Subclass derives from parent, reusing properties/methods	Class built by including other objects as fields
Relationship	Hierarchical — “Dog is an Animal”	Structural — “Car has an Engine”
Flexibility	Rigid; parent changes ripple through subclasses	Flexible; parts can be swapped or modified
Coupling	Tight coupling between parent and child	Loose coupling; modules interact via interfaces

🧩 Composition vs Inheritance (cont.)

Aspect	Inheritance (*is-a*)	Composition (*has-a*)
Reuse	Code reuse via inheritance	Code reuse via delegation to contained objects
Risks	Fragile hierarchies, explosion of subclasses	More maintainable, avoids subclass explosion
Best Use	Clear hierarchical “is-a” relationships	Combining behaviors, avoiding duplication

💡 Interpretation: Why Favor Composition

Inheritance is appropriate for natural hierarchies, but it can make systems brittle.
Composition is more adaptable because modules interact through well-defined boundaries.
This approach reduces duplication and supports long-term maintainability.

🚫 Duplication and Coupling

- Duplication is bad → fix the **design**, not just the code
- **Coupling** → when two modules are tightly bound together, changes in one force changes in the other
- **Solid line = coupling** → direct dependency between modules
- **Dotted line = dependency injection** → reduces circular dependencies and improves adaptability

🧱 Coupling Explained

Coupling is the **degree of dependency between modules or classes**.

- **Tight coupling** → modules rely heavily on each other's internal details
- **Loose coupling** → modules interact only through well-defined interfaces

Examples of Coupling:

- A UI class directly accessing database queries (tight coupling)
- A payment service hard-coding calls to a specific bank API (tight coupling)
- A logging module used via an interface (loose coupling)
- Dependency injection frameworks (loose coupling)

💡 Interpretation: Coupling

Tightly coupled modules are fragile because a small change can break multiple parts of the system.
Loose coupling isolates complexity, allowing modules to evolve independently.
Interfaces and abstractions are the key to achieving loose coupling.

🧩 SOLID Principles

- **S: Single Responsibility** → one reason to change
- **O: Open/Closed** → open for extension, closed for modification
- **L: Liskov Substitution** → subclasses must not alter expected behavior
- **I: Interface Segregation** → no client should depend on unused methods
- **D: Dependency Inversion** → high-level modules depend on abstractions, not low-level details

💡 Interpretation: SOLID

SOLID principles keep code safe to change and easier to extend.
They reduce hidden dependencies, prevent fragile hierarchies, and encourage designs that adapt to new requirements without breaking existing functionality.

🐾 HW4 Example: Animal Shelter API

- **Coupling** → Each module exposes only what is necessary through its API. For example, the Adoptions module interacts with Animals via endpoints, not direct database access. This reduces tight coupling.
- **Single Responsibility** → Each module handles one concern: Animals manage traits, Adoptions track requests, Users manage credentials.
- **Interface Segregation** → Clients only depend on the endpoints they need. A user registering does not need adoption endpoints.
- **Open/Closed** → New endpoints can be added (e.g., search filters) without modifying existing ones.

🐾 HW4 Example: Animal Shelter API Testing

- Testing is performed by running a **bash script of unit tests**
- Unit tests validate each API endpoint (e.g., GET /animals, POST /adoptions)
- Ensures endpoints behave as expected and changes don’t break existing functionality
- ✅ Demonstrates iterative design: tests provide quick feedback and protect against regressions

Week Five: Software Evolution & Architecture

Date: November 13th

🎯 The 80:20 Rule

- Focus on discovering **unknown unknowns** early
- ✅ Effective design anticipates problems before they surface

💡 Interpretation: 80:20 Rule

Software engineering is not about removing every risk.
The goal is to identify the most significant risks as early as possible.
Addressing them sooner prevents costly rework later in the project.

🏗️ MVVM Architecture

- **Model–View–ViewModel** separates concerns:
  - Model → data and business logic
  - View → UI representation
  - ViewModel → mediator between model and view

💡 Interpretation: MVVM

Separating responsibilities makes systems easier to test and maintain.
Logic stays in the Model and ViewModel, while the View focuses only on presentation.
This division allows teams to work in parallel without interfering with each other.

🐞 Debugging Philosophy

Good software engineering isn’t just finding bugs — it’s designing code so bugs **reveal themselves**.

💡 Interpretation: Debugging

Well-structured systems expose problems instead of hiding them.
Clear error handling and strong typing make failures visible, so they can be fixed quickly.
This reduces time spent searching for hidden issues.

😀 Happy Engineers = Successful Engineers

The most successful software projects are built by teams that enjoy their work.

💡 Interpretation: Happiness

Satisfied engineers produce higher-quality work.
When frustration is reduced through good design and clear processes, teams are more productive and creative.
Positive environments lead to better long-term outcomes.

🐾 HW4 Example: Animal Shelter API

Applying Week 5 Principles:

- **MVVM**
  - **Model** → Database tables: Animals, Adoptions, Users
  - **View** → Frontend UI: animal listings, adoption forms, user registration
  - **ViewModel** → API endpoints: `/animals`, `/adoptions`, `/users`
    This separation ensures the UI does not directly manipulate the database, making the system easier to test and extend.

🐾 HW4 Example: Animal Shelter API

Applying Week 5 Principles:

- **Debugging Philosophy**
  Errors surface clearly through endpoints. For instance, if an adoption request references a non-existent animal, the API returns a clear error message. This design makes problems visible instead of hidden.

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.