The APIEC Framework

A unified framework for object-oriented design principles:

- **A**bstraction – Focus on essential features, hide complexity
- **P**olymorphism – Many forms, uniform interfaces
- **I**nheritance – Code reuse through hierarchies
- **E**ncapsulation – Hide internal state, expose interfaces
- **C**omposition – Build complex objects from simpler ones

Key Principle: Favor composition over inheritance for flexibility and reduced coupling.

APIEC principles work together to create maintainable, extensible software designs.

Encapsulation & Dependency Injection

Encapsulation Example:

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def get_balance(self):  # Controlled access
        return self.__balance
```

Dependency Injection Example:

```
class EmailService:
    def send(self, message):
        pass
```

Encapsulation & Dependency Injection Concepts

- **Encapsulation:** Encapsulation means keeping an object's internal data private and exposing only what's necessary through methods or an interface. This protects the integrity of the object's state. For example, rather than allowing direct access to a class's fields, we provide getter/setter methods or use properties. Encapsulation promotes modularity and makes it easier to change implementation without affecting other parts of the code.
- **Dependency Injection (DI):** DI is a design technique where an object's dependencies (other objects it needs to function) are provided from the outside rather than created internally. In practice, this often means **passing required components via constructors or setters**. DI supports the *Inversion of Control* principle – instead of a class controlling its dependencies, the control is inverted and handed to an external entity (like a framework or a factory). This leads to more testable and flexible code, since you can swap out implementations (e.g., inject a mock object in tests or a

Polymorphism & Composition

Polymorphism Example:

```
class Shape:
    def draw(self):
        pass

class Circle(Shape):
    def draw(self):
        print("Drawing circle")

class Square(Shape):
    def draw(self):
        print("Drawing square")

shapes = [Circle(), Square()]
for shape in shapes:
    shape.draw()  # Polymorphic behavior
```

Composition Example

Composition (favor over inheritance):

```
class Engine:
    def start(self):
        print("Engine starting")

class Wheels:
    def rotate(self):
        print("Wheels rotating")

class Car:  # Composition: Car HAS-A Engine and Wheels
    def __init__(self):
        self.engine = Engine()
        self.wheels = Wheels()

    def drive(self):
        self.engine.start()
        self.wheels.rotate()
```

Polymorphism & Composition Concepts

- **Polymorphism:** Literally "many forms," polymorphism in OOP allows treating objects of different classes through a uniform interface. For example, if `Shape` is a base class with a method `draw()`, polymorphism lets us call `draw()` on any subclass (`Circle`, `Square`, etc.) and get subclass-specific behavior. At runtime, the actual object's implementation is invoked. Polymorphism improves extensibility – new subclasses can be introduced without changing code that uses the base class interface.
- **Composition:** Composition is a design technique where a class is composed of one or more other classes, implying a strong *has-a* relationship. Instead of using inheritance to reuse code, a class can have instances of other classes as members. For example, a `Car` class might *compose* a `Engine` class and `Wheel` classes, rather than inheriting from them, since a car is not a type of engine but has an engine. Composition is often favored over inheritance for flexibility – you can change composed parts without

Use Case Diagrams

Example UML use case diagram (restaurant scenario): Each **actor** (user role) is connected to the **use cases** (ovals) they participate in.

- **Use Case Diagram** – a UML diagram to model **functional requirements** of the system. It shows *actors* (users or external systems) and *use cases* (services or functions the system provides to those actors).
- Use case diagrams provide a high-level overview of **who** uses the system and **what** they can do. For example, in an online shopping system, actors might be *Customer* and *Admin*, and use cases could include *Browse Products*, *Place Order*, *Manage Inventory*, etc.
- Each use case represents a distinct functionality or goal from the actor's perspective. Relationships in use case diagrams can include **<<extends>>** or **<<includes>>** (to show optional or common sub-flows). This diagram is a communication tool between

Software Testing

- **Software Testing** is the process of evaluating a software item to detect differences between **expected output and actual output** (i.e., finding defects). In practice, testing involves executing the software with sample inputs and verifying that outputs and behaviors match the requirements.
    - **2025 Market:** Software testing market valued at $87.42B in 2024, projected to reach $512.3B by 2033 (CAGR 21.71%)
- **Testing Levels:**
    - *Unit Testing* – testing individual components or functions in isolation for correct behavior.
    - *Integration Testing* – checking that different modules or services work together properly.
    - *System Testing* – validating the entire integrated system against the requirements.

Testing Example & TDD

Unit Test Example (pytest):

```
def calculate_total(price, quantity):
    return price * quantity

def test_calculate_total():
    assert calculate_total(10, 5) == 50
    assert calculate_total(0, 10) == 0
    assert calculate_total(7.5, 2) == 15.0
```

Approach: We follow good testing practices like writing test cases for both normal and edge conditions. Automated tests (using frameworks like `unittest` or `pytest` in Python) help catch regressions quickly. A solid test suite gives confidence to refactor code since you can verify nothing broke. In fact, one of the Agile practices is **Test-Driven Development (TDD)** – writing tests before implementing code to drive correct behavior and design.

SOLID Principles

- **SOLID Principles:** Five foundational design principles for maintainable OOP code:
    i. **S**ingle Responsibility Principle – a class should have only one reason to change (one responsibility).
    ii. **O**pen/Closed Principle – software entities should be open for extension but closed for modification.
    iii. **L**iskov Substitution Principle – objects of a superclass should be replaceable with objects of subclasses without breaking the system.
    iv. **I**nterface Segregation Principle – no client should be forced to depend on methods it doesn't use (split large interfaces into smaller, specific ones).
    v. **D**ependency Inversion Principle – high-level modules should not depend on low-level modules; both should depend on abstractions.
- Applying SOLID leads to cleaner, more flexible designs. For instance, adhering to SRP

SOLID Example: Single Responsibility

Violation (Multiple Responsibilities):

```
class User:
    def save_to_database(self):
        # Database logic here
        pass

    def send_email(self):
        # Email logic here
        pass
```

Following SRP (Single Responsibility):

```
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

class UserRepository:  # Handles DB operations
```

Design Patterns

- **Design Patterns:** A design pattern is a general, reusable solution to a commonly occurring problem in software design. It's a template for how to solve a problem that can be adapted to different situations, rather than a direct piece of code. Classic patterns (from the "Gang of Four" book) include creational patterns (e.g. Factory), structural patterns (e.g. Adapter), and behavioral patterns (e.g. Observer).
    - **2025 Reality:** Many patterns now built into modern languages, but remain essential as conceptual tools
    - 23 Gang of Four patterns still relevant as shared vocabulary among developers

Iterator Pattern Example

Iterator Pattern in Python:

```
class BookCollection:
    def __init__(self):
        self.books = []

    def add_book(self, book):
        self.books.append(book)

    def __iter__(self):  # Iterator protocol
        return iter(self.books)

    def __next__(self):
        # Custom iteration logic if needed
        pass

# Usage
collection = BookCollection()
for book in collection:  # Clean, standard iteration
```

Iterator Pattern Concepts

- *Iterator Pattern:* The Iterator is a behavioral pattern that **provides a standard way to access elements of a collection sequentially without exposing the collection's internal structure**. For example, Python's iterator protocol (`__iter__` and `__next__`) allows you to loop over objects like lists and custom collections in a uniform way. Design patterns like this improve interoperability and code clarity. Using known patterns can also make your design more communicative – other developers recognize the pattern and understand the solution approach more readily.