Lab Session 2

Advanced OOP, Design Patterns & Architecture

The Art of Computer Programming 2 — Week 2 — February 27, 2026

Objectives

By the end of this lab, you should be able to:

- Build a class hierarchy with inheritance and override methods in Python and C++
- Use abstract classes to define interfaces and enforce contracts
- Overload operators to create natural-feeling mathematical types
- Apply the Strategy pattern to swap algorithms at runtime
- Read and draw simple UML class diagrams

Duration: 1h30
Languages: Python (primary), C++ (bonus)

Mode: Pen & paper (Ex. 1) Computer (Ex. 2–5)

1 Warm-up: UML & Concepts (15 min)

Pen and paper — no computer needed.

1.1 Vocabulary check

For each statement, answer **True** or **False** and briefly justify.

#	Statement	T/F?
a	An abstract class can have concrete (non-abstract) methods.	
b	In Python, `@abstractmethod` prevents instantiation of the class.	
c	In C++, a class with at least one pure virtual function is abstract.	
d	Multiple inheritance is supported in both Python and C++.	
e	The Strategy pattern uses inheritance to swap algorithms.	
f	Python's `@property` decorator is an example of the Decorator pattern.	

1.2 Draw a UML class diagram

Consider the following system:

- A `Vehicle` has a `speed` (float) and a method `move()`.
- `Car` and `Bicycle` are vehicles. A `Car` has `num_doors` (int).
- A `Car` has an `Engine` (composition). An `Engine` has `horsepower` (int).
- A `GPS` can be associated with any `Vehicle` (aggregation).

Draw the UML class diagram showing: classes (with at least one attribute and one method each), inheritance, composition, and aggregation arrows.

2 Shape Hierarchy (25 min)

Build a class hierarchy for geometric shapes in Python.

2.1 Abstract base class

Create an abstract class `Shape` using the `abc` module:

```python
from abc import ABC, abstractmethod
import math

class Shape(ABC):
    @abstractmethod
    def area(self) -> float:
        """Return the area of the shape."""
        pass

    @abstractmethod
    def perimeter(self) -> float:
        """Return the perimeter of the shape."""
        pass

    def __str__(self) -> str:
        return (f"{self.__class__.__name__}: "
                f"area={self.area():.2f}, "
                f"perimeter={self.perimeter():.2f}")
```

2.2 Concrete shapes

Implement the following concrete classes that inherit from `Shape`:

(a) `Circle(radius)` — area $= \pi r^2$, perimeter $= 2\pi r$

(b) `Rectangle(width, height)` — area $= wh$, perimeter $= 2(w + h)$

(c) `Triangle(a, b, c)` — area via Heron's formula:

$$s = \frac{a+b+c}{2}, \qquad \text{area} = \sqrt{s(s-a)(s-b)(s-c)}$$

Hint

You can validate your triangle by checking that `Triangle(3,4,5).area()` returns 6.0.

2.3 Polymorphism test

Write a function `total_area(shapes: list[Shape]) -> float` that returns the sum of all areas. Test it with a list containing at least one of each shape type.

2.4 Square as a special Rectangle

Create a `Square(side)` class that inherits from `Rectangle`. Its constructor should call `super().__init__(side, side)`.

Hint

Verify that `isinstance(Square(5), Rectangle)` returns `True`.

3 Operator Overloading: Vector Arithmetic (25 min)

3.1 The `Vector` class

Implement a `Vector` class that supports the following operations:

```python
class Vector:
    def __init__(self, *components):
        """Initialize with variable number of components."""
        pass

    def __add__(self, other):
        """Vector addition: v1 + v2."""
        pass

    def __sub__(self, other):
        """Vector subtraction: v1 - v2."""
        pass

    def __mul__(self, scalar):
        """Scalar multiplication: v * 3."""
        pass

    def __rmul__(self, scalar):
        """Right scalar multiplication: 3 * v."""
        pass

    def __eq__(self, other):
        """Equality: v1 == v2."""
        pass

    def __len__(self):
        """Dimension: len(v)."""
        pass

    def __getitem__(self, index):
        """Indexing: v[0]."""
        pass

    def __repr__(self):
        """String representation."""
        pass

    def dot(self, other):
        """Dot product."""
        pass

    def norm(self):
        """Euclidean norm."""
        pass
```

3.2 Test your implementation

Verify the following:

```python
v1 = Vector(1, 2, 3)
v2 = Vector(4, 5, 6)

assert v1 + v2 == Vector(5, 7, 9)
assert v1 - v2 == Vector(-3, -3, -3)
assert v1 * 2  == Vector(2, 4, 6)
assert 2 * v1  == Vector(2, 4, 6)
assert v1.dot(v2) == 32
assert len(v1) == 3
assert v1[0] == 1
print(f"norm = {v1.norm():.4f}")  # 3.7417
```

3.3 Matrix–vector product

Recall

A matrix–vector product $y = Ax$ is defined as: $y_i = \sum_j A_{ij} x_j$.

Create a `Matrix` class that stores a list of `Vector` rows. Overload `__matmul__` (`@` operator) so that `A @ v` returns a `Vector`:

```python
A = Matrix(Vector(1, 0), Vector(0, 1))  # identity
v = Vector(3, 4)
assert A @ v == Vector(3, 4)
```

4 Strategy Pattern: Sorting Strategies (15 min)

4.1 Define the strategy interface

```python
from abc import ABC, abstractmethod

class SortStrategy(ABC):
    @abstractmethod
    def sort(self, data: list) -> list:
        """Return a new sorted list."""
        pass
```

4.2 Implement concrete strategies

Implement at least three sorting strategies:

(a) `InsertionSort` — $O(n^2)$

(b) `MergeSortStrategy` — $O(n \log n)$ (use your Week 1 implementation)

(c) `PythonBuiltinSort` — wraps Python's `sorted()`

4.3 Context class

Create a `Sorter` class:

```python
class Sorter:
    def __init__(self, strategy: SortStrategy):
        self._strategy = strategy

    def set_strategy(self, strategy: SortStrategy):
        self._strategy = strategy

    def sort(self, data: list) -> list:
        return self._strategy.sort(data)
```

Test by sorting the same list with all three strategies and verifying the results are identical.

4.4 Benchmark

Use `time.time()` to compare the three strategies on random lists of sizes $n = 10^3, 10^4$. Which strategy is fastest? Does this match the theory?

Hint

Use `import random; data = [random.randint(0, 10**6) for _ in range(n)]`.

5 Putting It All Together: Mini Blackjack (30 min)

This exercise ties together all the concepts from today's lecture: inheritance, abstract classes, operator overloading, and the Strategy pattern, in a single project: a simplified Blackjack simulator.

Recall

Blackjack rules (simplified): Each player tries to reach a hand score as close to **21** as possible without exceeding it. Face cards (J, Q, K) are worth 10; an Ace is worth 11 or 1 (whichever is better). The dealer must hit (draw) until reaching at least 17.

5.1 `Card` class

Implement a `Card` class:

```python
class Card:
    SUITS = ["Hearts", "Diamonds", "Clubs", "Spades"]
    RANKS = ["2","3","4","5","6","7","8","9","10","J","Q","K","A"]

    def __init__(self, rank: str, suit: str):
        ...

    @property
    def value(self) -> int:
        """Return the card value (Ace = 11 for now)."""
        ...

    def __repr__(self) -> str:
        """E.g. 'A of Spades'."""
        ...

    def __lt__(self, other) -> bool:
        """Compare by value."""
        ...

    def __eq__(self, other) -> bool:
        ...
```

5.2 `Deck` class

Implement a `Deck` that generates all 52 cards, supports `shuffle()` and `draw()` (returns and removes the top card), and `len(deck)` for remaining cards.

Hint

Use `random.shuffle` and a list comprehension with `Card.RANKS` and `Card.SUITS`.

5.3 `Hand` class

Implement a `Hand` class:

```python
class Hand:
    def __init__(self):
        self.cards = []

    def add(self, card: Card):
        """Add a card to the hand."""
        ...

    @property
    def score(self) -> int:
        """Compute the score.
        Aces count as 11 unless that would bust,
        in which case they count as 1."""
        ...

    @property
    def is_bust(self) -> bool:
        return self.score > 21

    def __repr__(self) -> str:
        return f"{self.cards} (score: {self.score})"
```

Important

The tricky part is `score`: sum all values (Ace = 11), then for each Ace, if the total exceeds 21, subtract 10. Test with: `A + 9 = 20`, `A + A + 9 = 21`, `A + 5 + 8 = 14`.

5.4 Player strategies (Strategy pattern)

Define an abstract strategy and two concrete strategies:

```python
from abc import ABC, abstractmethod

class PlayerStrategy(ABC):
    @abstractmethod
    def should_hit(self, hand: Hand, dealer_visible: Card) -> bool:
        """Return True if the player should draw another card."""
        pass

class DealerStrategy(PlayerStrategy):
    """Hit until score >= 17."""
    ...

class ConservativeStrategy(PlayerStrategy):
    """Hit until score >= 15, but stand if dealer shows <= 6."""
    ...

class AggressiveStrategy(PlayerStrategy):
    """Hit until score >= 18."""
    ...
```

5.5 `Player` class

Create a `Player` class that holds a `name`, a `Hand`, and a `PlayerStrategy`. Add a method `play(deck, dealer_visible_card)` that draws cards based on the strategy.

5.6 Game simulation

Write a `play_round(players, dealer)` function that:

1. Creates a `Deck` and shuffles it
2. Deals 2 cards to each player and the dealer
3. Each player plays according to their strategy (the dealer's first card is visible)
4. The dealer plays last (`DealerStrategy`)
5. Determines the winner(s): closest to 21 without busting

Run 10 000 rounds and report the win rate for each strategy.

Hint

Expected approximate win rates: Aggressive ≈ Conservative ≈ 40–45%. The dealer wins ties. The exact rates depend on your strategy thresholds.

6 Bonus: Shape Hierarchy in C++ (optional)

Bonus

Port your Python shape hierarchy to C++. Specifically:

(a) Define `Shape` as an abstract class with pure virtual `area()` and `perimeter()`.
(b) Implement `Circle`, `Rectangle`, and `Triangle`.
(c) Write a function `void print_info(const Shape& s)` that prints area and perimeter.
(d) In `main()`, create a `std::vector<Shape*>`, add shapes, and call `print_info` in a loop.
(e) Don't forget to `delete` your pointers (or use `std::unique_ptr`).

Hint

Remember: in C++, you need `virtual` keyword on methods you want to override, and `override` on derived class methods. Don't forget the `virtual` destructor!