1 Warm-up: UML & Concepts

1.1 Vocabulary check

Solution

#	Statement	T/F	Justification
a	An abstract class can have concrete methods.	**T**	Abstract classes may mix abstract and concrete methods. E.g., our `Shape` has concrete `__str__`.
b	`@abstractmethod` prevents instantiation.	**T**	Only if the class inherits from `ABC`. Calling `Shape()` raises `TypeError`.
c	A class with $\geqslant 1$ pure virtual function is abstract.	**T**	By definition in C++.
d	Multiple inheritance is supported in both Python and C++.	**T**	Python uses MRO (C3 linearization); C++ uses virtual inheritance to resolve the diamond problem.
e	The Strategy pattern uses inheritance to swap algorithms.	**T**	Strategies share a common abstract interface; concrete strategies inherit from it. The context class delegates to the strategy via composition.
f	Python's `@property` is an example of the Decorator pattern.	**F**	`@property` is a descriptor protocol, not the GoF Decorator pattern. The `@` syntax is called "decorator" in Python, but the GoF Decorator pattern wraps objects to add behavior.

Common Mistakes

(f) is the trickiest. Students confuse Python's `@decorator` syntax with the GoF Decorator *design pattern*. Clarify: Python decorators wrap *functions*, the GoF pattern wraps *objects* to add responsibilities. They share a name but are different concepts.

1.2 Draw a UML class diagram

Solution

Expected diagram should show:

- `Vehicle` (abstract, with `speed: float`, `move()`)
- `Car` and `Bicycle` inheriting from `Vehicle` (open triangle arrows)
- `Car` has `num_doors: int`

- `Engine` class (with `horsepower: int`) connected to `Car` via composition (filled diamond)
- `GPS` class connected to `Vehicle` via aggregation (open diamond)

Grading Notes

Check for: correct arrow types (open triangle for inheritance, filled diamond for composition, open diamond for aggregation), attributes in the right classes, and correct direction of arrows.

2 Shape Hierarchy

2.1 Abstract base class

Solution

Provided in the lab sheet. No additional work needed.

2.2 Concrete shapes

Solution

```python
class Circle(Shape):
    def __init__(self, radius: float):
        self.radius = radius

    def area(self) -> float:
        return math.pi * self.radius ** 2

    def perimeter(self) -> float:
        return 2 * math.pi * self.radius


class Rectangle(Shape):
    def __init__(self, width: float, height: float):
        self.width = width
        self.height = height

    def area(self) -> float:
        return self.width * self.height

    def perimeter(self) -> float:
        return 2 * (self.width + self.height)


class Triangle(Shape):
    def __init__(self, a: float, b: float, c: float):
        self.a, self.b, self.c = a, b, c

    def area(self) -> float:
        s = (self.a + self.b + self.c) / 2
        return math.sqrt(s * (s-self.a) * (s-self.b) * (s-self.c))

    def perimeter(self) -> float:
        return self.a + self.b + self.c
```

Common Mistakes

- Forgetting to call `math.sqrt` in Heron's formula (returning $s(s-a)(s-b)(s-c)$ instead).
- Not validating the triangle inequality. Acceptable for this exercise, but mention it.

2.3 Polymorphism test

Solution

```python
def total_area(shapes: list) -> float:
    return sum(s.area() for s in shapes)

shapes = [Circle(5), Rectangle(3, 4), Triangle(3, 4, 5)]
print(f"Total area: {total_area(shapes):.2f}")
# Expected: 78.54 + 12.00 + 6.00 = 96.54
```

Explanation

This demonstrates **polymorphism**: `total_area` doesn't need to know the concrete type of each shape. It calls `area()` on each, and Python dispatches to the correct method at runtime.

2.4 Square as a special Rectangle

Solution

```python
class Square(Rectangle):
    def __init__(self, side: float):
        super().__init__(side, side)

sq = Square(5)
print(sq.area())       # 25.0
print(sq.perimeter())  # 20.0
print(isinstance(sq, Rectangle))  # True
print(isinstance(sq, Shape))      # True
```

Grading Notes

Students must use `super().__init__(side, side)` and not redefine `area()` or `perimeter()` — those are inherited from `Rectangle`.

3 Operator Overloading: Vector Arithmetic

3.1 The `Vector` class

Solution

```python
import math

class Vector:
    def __init__(self, *components):
        self._data = list(components)

    def __add__(self, other):
        return Vector(*(a + b for a, b in zip(self._data, other._data)))

    def __sub__(self, other):
        return Vector(*(a - b for a, b in zip(self._data, other._data)))

    def __mul__(self, scalar):
        return Vector(*(x * scalar for x in self._data))

    def __rmul__(self, scalar):
        return self.__mul__(scalar)

    def __eq__(self, other):
        return self._data == other._data

    def __len__(self):
        return len(self._data)

    def __getitem__(self, index):
        return self._data[index]

    def __repr__(self):
        return f"Vector({', '.join(map(str, self._data))})"

    def dot(self, other):
        return sum(a * b for a, b in zip(self._data, other._data))

    def norm(self):
        return math.sqrt(self.dot(self))
```

Common Mistakes

- Forgetting `__rmul__`: then `3 * v` fails while `v * 3` works.
- Using `__str__` instead of `__repr__`: both are acceptable, but `__repr__` is more standard for objects.
- Not unpacking the generator in `Vector(*(...))` — this creates a single-element vector containing a generator.

3.2 Matrix–vector product

Solution

```python
class Matrix:
    def __init__(self, *rows):
        self.rows = list(rows)  # each row is a Vector

    def __matmul__(self, vec):
        """Matrix @ Vector product."""
        return Vector(*(row.dot(vec) for row in self.rows))

    def __repr__(self):
        return "Matrix(\n  " + ",\n  ".join(repr(r) for r in self.rows) + "\n)"

# Test
I = Matrix(Vector(1, 0), Vector(0, 1))
v = Vector(3, 4)
assert I @ v == Vector(3, 4)

A = Matrix(Vector(1, 2), Vector(3, 4))
assert A @ Vector(1, 1) == Vector(3, 7)
```

Explanation

The `@` operator in Python (PEP 465) is specifically designed for matrix multiplication. It calls `__matmul__`. This is exactly how NumPy implements `A @ x`.

4 Strategy Pattern: Sorting Strategies

4.1 Concrete strategies

Solution

```python
from abc import ABC, abstractmethod

class SortStrategy(ABC):
    @abstractmethod
    def sort(self, data: list) -> list:
        pass

class InsertionSort(SortStrategy):
    def sort(self, data: list) -> list:
        arr = data[:]
        for i in range(1, len(arr)):
            key = arr[i]
            j = i - 1
            while j >= 0 and arr[j] > key:
                arr[j + 1] = arr[j]
                j -= 1
            arr[j + 1] = key
        return arr
```

```python
class MergeSortStrategy(SortStrategy):
    def sort(self, data: list) -> list:
        if len(data) <= 1:
            return data[:]
        mid = len(data) // 2
        left = self.sort(data[:mid])
        right = self.sort(data[mid:])
        return self._merge(left, right)

    def _merge(self, left, right):
        result = []
        i = j = 0
        while i < len(left) and j < len(right):
            if left[i] <= right[j]:
                result.append(left[i]); i += 1
            else:
                result.append(right[j]); j += 1
        result.extend(left[i:])
        result.extend(right[j:])
        return result

class PythonBuiltinSort(SortStrategy):
    def sort(self, data: list) -> list:
        return sorted(data)
```

4.2 Context class and test

Solution

```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
import random
data = [random.randint(0, 1000) for _ in range(100)]
expected = sorted(data)

sorter = Sorter(InsertionSort())
assert sorter.sort(data) == expected

sorter.set_strategy(MergeSortStrategy())
assert sorter.sort(data) == expected

sorter.set_strategy(PythonBuiltinSort())
assert sorter.sort(data) == expected

print("All strategies produce the same result!")
```

4.3 Benchmark

Solution

```python
import time
import random

strategies = [
    ("InsertionSort", InsertionSort()),
    ("MergeSort",     MergeSortStrategy()),
    ("Python sorted", PythonBuiltinSort()),
]

for n in [1000, 10000]:
    data = [random.randint(0, 10**6) for _ in range(n)]
    print(f"\n--- n = {n} ---")
    for name, strat in strategies:
        start = time.time()
        strat.sort(data)
        elapsed = time.time() - start
        print(f"  {name:20s}: {elapsed:.4f}s")
```

Expected output (approximate):

Strategy	$n = 10^3$	$n = 10^4$
InsertionSort	$\sim$0.02 s	$\sim$2.0 s
MergeSort	$\sim$0.003 s	$\sim$0.04 s
Python sorted	$\sim$0.0001 s	$\sim$0.002 s

Analysis: InsertionSort is $O(n^2)$ so it scales quadratically ($10\times$ input $\Rightarrow$ $100\times$ time). MergeSort is $O(n \log n)$. Python's `sorted()` uses Timsort (also $O(n \log n)$) but is implemented in C, so it's much faster than pure-Python merge sort.

Grading Notes

- Students should observe the $O(n^2)$ vs $O(n \log n)$ difference clearly at $n = 10^4$.
- Accept any reasonable timing methodology (`time.time()`, `timeit`).
- Bonus point if they note that Python's `sorted()` is C-optimized Timsort.

5 Putting It All Together: Mini Blackjack

5.1 `Card` class

Solution

```python
class Card:
    SUITS = ["Hearts", "Diamonds", "Clubs", "Spades"]
    RANKS = ["2","3","4","5","6","7","8","9","10","J","Q","K","A"]
    VALUES = {"2":2,"3":3,"4":4,"5":5,"6":6,"7":7,
              "8":8,"9":9,"10":10,"J":10,"Q":10,"K":10,"A":11}

    def __init__(self, rank: str, suit: str):
        assert rank in self.RANKS, f"Invalid rank: {rank}"
        assert suit in self.SUITS, f"Invalid suit: {suit}"
        self.rank = rank
        self.suit = suit

    @property
    def value(self) -> int:
        return self.VALUES[self.rank]

    def __repr__(self) -> str:
        return f"{self.rank} of {self.suit}"

    def __lt__(self, other) -> bool:
        return self.value < other.value

    def __eq__(self, other) -> bool:
        return self.rank == other.rank and self.suit == other.suit
```

5.2 Deck class

Solution

```python
import random

class Deck:
    def __init__(self):
        self.cards = [Card(r, s)
                      for s in Card.SUITS
                      for r in Card.RANKS]

    def shuffle(self):
        random.shuffle(self.cards)

    def draw(self) -> Card:
        return self.cards.pop()

    def __len__(self) -> int:
        return len(self.cards)
```

5.3 Hand class

Solution

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

    def add(self, card: Card):
        self.cards.append(card)

    @property
    def score(self) -> int:
        total = sum(c.value for c in self.cards)
        aces = sum(1 for c in self.cards if c.rank == "A")
        while total > 21 and aces > 0:
            total -= 10
            aces -= 1
        return total

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

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

Explanation

The `score` property handles aces greedily: start with all aces valued at 11, then downgrade one at a time to 1 if the total exceeds 21. This guarantees the best possible score.

Common Mistakes

- Counting aces *after* the while loop instead of before (variable is consumed).
- Only handling a single ace: the loop must handle multiple aces (e.g., A + A + 9 = 21, not 31).
- Using `card.value == 11` instead of `card.rank == "A"`: fragile if the property changes.

5.4 Player strategies (Strategy pattern)

Solution

```python
from abc import ABC, abstractmethod

class PlayerStrategy(ABC):
    @abstractmethod
    def should_hit(self, hand: Hand, dealer_visible: Card) -> bool:
        pass

class DealerStrategy(PlayerStrategy):
    """Standard dealer: hit until score >= 17."""
    def should_hit(self, hand: Hand, dealer_visible: Card) -> bool:
        return hand.score < 17

class ConservativeStrategy(PlayerStrategy):
    """Hit until 15, but stand if dealer shows a weak card (<= 6)."""
    def should_hit(self, hand: Hand, dealer_visible: Card) -> bool:
        if dealer_visible.value <= 6:
            return hand.score < 12
        return hand.score < 15

class AggressiveStrategy(PlayerStrategy):
    """Hit until 18, regardless of dealer card."""
    def should_hit(self, hand: Hand, dealer_visible: Card) -> bool:
        return hand.score < 18
```

Grading Notes

Accept any reasonable strategy logic. The key points are:

- Abstract class with `@abstractmethod`
- Each strategy implements `should_hit` differently
- Strategies use the `hand` score and optionally the dealer's visible card

5.5 `Player` class

Solution

```python
class Player:
    def __init__(self, name: str, strategy: PlayerStrategy):
        self.name = name
        self.strategy = strategy
        self.hand = Hand()

    def play(self, deck: Deck, dealer_visible: Card):
        while self.strategy.should_hit(self.hand, dealer_visible):
            self.hand.add(deck.draw())

    def reset(self):
        self.hand = Hand()

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

5.6 Game simulation

Solution

```python
def play_round(players: list, dealer: Player) -> str | None:
    """Play one round. Return the name of the winner, or None (dealer wins)."""
    deck = Deck()
    deck.shuffle()

    # Reset hands
    for p in players:
        p.reset()
    dealer.reset()

    # Deal 2 cards each
    for _ in range(2):
        for p in players:
            p.hand.add(deck.draw())
        dealer.hand.add(deck.draw())

    dealer_visible = dealer.hand.cards[0]

    # Players play
    for p in players:
        p.play(deck, dealer_visible)

    # Dealer plays
    dealer.play(deck, dealer_visible)

    # Determine winner
    dealer_score = dealer.hand.score
    best_name = None
    best_score = dealer_score if not dealer.hand.is_bust else 0

    for p in players:
        if not p.hand.is_bust and p.hand.score > best_score:
            best_score = p.hand.score
            best_name = p.name

    return best_name  # None = dealer wins
```

```python
# --- Simulation ---
N = 10_000
players = [
    Player("Conservative", ConservativeStrategy()),
    Player("Aggressive",   AggressiveStrategy()),
]
dealer = Player("Dealer", DealerStrategy())

wins = {p.name: 0 for p in players}
wins["Dealer"] = 0

for _ in range(N):
    winner = play_round(players, dealer)
    if winner is None:
        wins["Dealer"] += 1
    else:
        wins[winner] += 1

print(f"Results over {N} rounds:")
for name, count in wins.items():
    print(f"  {name:15s}: {count:5d} wins  ({100*count/N:.1f}%)")
```

Typical output:

```
Results over 10000 rounds:
  Conservative   :  2805 wins  (28.1%)
  Aggressive     :  3312 wins  (33.1%)
  Dealer         :  3883 wins  (38.8%)
```

Note: Exact numbers vary due to randomness. The dealer has an advantage because they play last and win ties. The aggressive strategy tends to slightly outperform conservative because standing too early often loses to the dealer.

Grading Notes

- Full credit if the simulation runs, strategies behave differently, and results are discussed.
- The exact win rates don't matter — accept any reasonable numbers.
- Bonus if students add additional strategies or a `set_strategy` method.
- Common issue: forgetting to `reset()` hands between rounds.

6 Bonus: Shape Hierarchy in C++

Solution

```cpp
#include <iostream>
#include <cmath>
#include <vector>
#include <memory>

class Shape {
public:
    virtual double area() const = 0;
    virtual double perimeter() const = 0;
    virtual ~Shape() = default;
};

void print_info(const Shape& s) {
    std::cout << "Area: " << s.area()
              << ", Perimeter: " << s.perimeter() << "\n";
}

class Circle : public Shape {
    double r;
public:
    Circle(double r) : r(r) {}
    double area() const override { return M_PI * r * r; }
    double perimeter() const override { return 2 * M_PI * r; }
};
```

```cpp
class Rectangle : public Shape {
protected:
    double w, h;
public:
    Rectangle(double w, double h) : w(w), h(h) {}
    double area() const override { return w * h; }
    double perimeter() const override { return 2 * (w + h); }
};

class Square : public Rectangle {
public:
    Square(double s) : Rectangle(s, s) {}
};

class Triangle : public Shape {
    double a, b, c;
public:
    Triangle(double a, double b, double c) : a(a), b(b), c(c) {}
    double area() const override {
        double s = (a + b + c) / 2;
        return std::sqrt(s * (s-a) * (s-b) * (s-c));
    }
    double perimeter() const override { return a + b + c; }
};

int main() {
    std::vector<std::unique_ptr<Shape>> shapes;
    shapes.push_back(std::make_unique<Circle>(5));
    shapes.push_back(std::make_unique<Rectangle>(3, 4));
    shapes.push_back(std::make_unique<Triangle>(3, 4, 5));
    shapes.push_back(std::make_unique<Square>(7));

    for (const auto& s : shapes)
        print_info(*s);

    return 0;
}
```

Grading Notes

Key points to check:

- `virtual` on base class methods, `override` on derived
- Virtual destructor on `Shape`
- Use of `std::unique_ptr` or manual `delete`
- Correct use of `const` on methods that don't modify state
- `protected` on `Rectangle` members so `Square` can access them