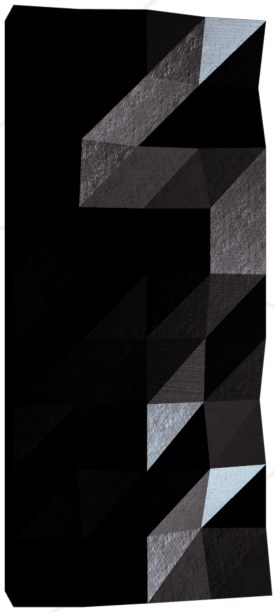

DataDeck

Master the Art of Abstract Card Architecture

Summary: Welcome to DataDeck - the ultimate card game engine! Master Python's abstract classes and interfaces by building a modular trading card system. Learn to create flexible, extensible game architectures through progressive challenges covering abstract base classes, multiple inheritance, and interface composition.

Version: 1.1

Contents

Chapter I

Foreword

Welcome, aspiring Deck Architect, to the world of DataDeck!

Picture this: You're designing the next legendary trading card game. Think Magic: The Gathering meets Hearthstone meets Pokémon - but with a twist! Your cards aren't just static images; they're dynamic data entities that can evolve, combine, and interact in countless ways. But here's the challenge: How do you create a system flexible enough to handle thousands of different card types while maintaining clean, maintainable code?

The secret lies in **Abstract Programming Patterns**! Just like how every trading card game has fundamental card types (Creatures, Spells, Artifacts), your code needs foundational blueprints that define how ALL cards behave.

Think of **abstract base classes** as the legendary card templates of programming. Whether you're creating a Fire Dragon, Lightning Bolt, or Healing Potion, they all follow the same core pattern: **cost, effect, activation, resolution**. Your abstract classes ensure every card in your deck follows these universal rules!

Interfaces are like the special abilities that cards can have. A card might be **Flyable** (can bypass ground defenses), **Stackable** (effects can combine), or **Tradeable** (can be exchanged between players). Python's ABC module lets you mix and match these abilities like building the perfect deck!

In this activity, you'll build "DataDeck" - a modular card game engine that could power the next generation of digital trading card games. You'll discover how abstract patterns make complex game systems manageable, extensible, and fun to develop. By the end, you'll think like a senior game architect, designing systems that can evolve with new card sets and mechanics!

Chapter II

AI Instructions

Context

During your learning journey, AI can assist with many different tasks. Take the time to explore the various capabilities of AI tools and how they can support your work. However, always approach them with caution and critically assess the results. Whether it's code, documentation, ideas, or technical explanations, you can never be completely sure that your question was well-formed or that the generated content is accurate. Your peers are a valuable resource to help you avoid mistakes and blind spots.

Main message

- Use AI to reduce repetitive or tedious tasks.
- Develop prompting skills — both coding and non-coding — that will benefit your future career.
- Learn how AI systems work to better anticipate and avoid common risks, biases, and ethical issues.
- Continue building both technical and power skills by working with your peers.
- Only use AI-generated content that you fully understand and can take responsibility for.

Learner rules:

- You should take the time to explore AI tools and understand how they work, so you can use them ethically and reduce potential biases.
- You should reflect on your problem before prompting — this helps you write clearer, more detailed, and more relevant prompts using accurate vocabulary.
- You should develop the habit of systematically checking, reviewing, questioning, and testing anything generated by AI.
- You should always seek peer review — don't rely solely on your own validation.

Phase outcomes:

- Develop both general-purpose and domain-specific prompting skills.
- Boost your productivity with effective use of AI tools.
- Continue strengthening computational thinking, problem-solving, adaptability, and collaboration.

Comments and examples:

- You'll regularly encounter situations — exams, evaluations, and more — where you must demonstrate real understanding. Be prepared, keep building both your technical and interpersonal skills.
- Explaining your reasoning and debating with peers often reveals gaps in your understanding. Make peer learning a priority.
- AI tools often lack your specific context and tend to provide generic responses. Your peers, who share your environment, can offer more relevant and accurate insights.
- Where AI tends to generate the most likely answer, your peers can provide alternative perspectives and valuable nuance. Rely on them as a quality checkpoint.

✓ Good practice:

> I ask AI: "How do I test a sorting function?" It gives me a few ideas. I try them out and review the results with a peer. We refine the approach together.

✗ Bad practice:

> I ask AI to write a whole function, copy-paste it into my project. During peer-evaluation, I can't explain what it does or why. I lose credibility — and I fail my project.

✓ Good practice:

> I use AI to help design a parser. Then I walk through the logic with a peer. We catch two bugs and rewrite it together — better, cleaner, and fully understood.

✗ Bad practice:

> I let Copilot generate my code for a key part of my project. It compiles, but I can't explain how it handles pipes. During the evaluation, I fail to justify and I fail my project.

Chapter III

Introduction

Welcome to DataDeck: Master the Art of Abstract Card Architecture!

You've conquered Python's basics, mastered data structures, and learned to organize code with classes. Now it's time to discover the architectural patterns that power enterprise-level game engines and data processing systems!

In this activity, you'll build DataDeck - a comprehensive trading card game engine. Each exercise introduces advanced object-oriented concepts through familiar card game mechanics:

- **Exercise 0**: Card Foundation - Master abstract base classes
- **Exercise 1**: Deck Builder - Implement concrete card types
- **Exercise 2**: Ability System - Design multiple interfaces
- **Exercise 3**: Game Engine - Build complex card interactions
- **Exercise 4**: Tournament Platform - Master interface composition

Think of this as building the "engine" of a trading card game - the core system that makes all the magic happen behind the scenes!

PREREQUISITES: This activity requires solid mastery of Python classes, inheritance, exception handling, and data structures. You should be comfortable with object-oriented programming concepts before tackling abstract patterns.

Focus on understanding **why** abstract patterns matter, not just **how** to implement them. A great game architect understands the design principles behind the code.

Chapter IV

Common Instructions

IV.1 General Rules

- Your project must be written in **Python 3.10 or later**.
- Your project must adhere to the **flake8** coding standard.
- Your functions should handle exceptions gracefully to avoid crashes.
- Use **type hints** for all function signatures and class methods.
- Focus on demonstrating abstract programming patterns clearly.
- All card processing should be done in-memory (no file I/O required).

IV.2 Authorized Imports

- **abc** module (Abstract Base Classes) - Essential for this project
- **typing** module - For advanced type hints
- **random** module - For card shuffling and effects
- **enum** module - For card types and rarities
- **datetime** module - For game timestamps
- Standard library modules as needed

IV.3 Forbidden

- External libraries (no pip install)
- File I/O operations (focus on in-memory processing)
- Complex game logic (keep card effects simple)
- Using eval() or exec()

IV.4 Project Structure

IMPORTANT - Repository Structure: Your Git repository must have the following structure:
Repository structure:

```
your-repo/
|- __init__.py      (REQUIRED)
|- ex0/
|  |- __init__.py   (REQUIRED)
|  |- Card.py
|  |- CreatureCard.py
|  '- main.py       (REQUIRED)
|- ex1/
|  |- __init__.py   (REQUIRED)
|  '- ...
|- ex4/
   |- __init__.py   (REQUIRED)
   '- ...
```

The `__init__.py` file at the repository root is **MANDATORY** for Python to recognize your exercises as packages and enable absolute imports.

Execution: All exercises must be executed from the repository root using: `python3 -m exN.main` (where N is the exercise number).
Example: `python3 -m ex0.main`

Imports: Use absolute imports between exercises:

- `from ex0.Card import Card`
- `from ex1.SpellCard import SpellCard`
- **Never** use relative imports like `from ..ex0.Card import Card`

IV.5 Helper Tools

Card Generator Available (Optional): A card generator utility is provided in the project attachments (card_generator.tar.gz) to help you during development. This is **NOT required for submission** - it's just a helper tool.

If you want to use it, extract it in your repository:

```
$ tar -xzf card_generator.tar.gz
$ mkdir -p tools
$ mv card_generator.py tools/
$ touch tools/__init__.py
```

You can then use it to generate sample cards for testing your implementations. Import with: `from tools.card_generator import CardGenerator`

Note: The `tools/` directory is for development only and should NOT be included in your Git submission.

Decorators Policy: While `@abstractmethod` decorators are commonly used with abstract classes, they are **optional** for this activity. You can implement abstract classes using the `abc` module with or without decorators - both approaches are acceptable and will be evaluated equally.

IV.6 The DataDeck Architecture

In this activity, you'll build a modular card game engine. Each exercise represents a key architectural component:

- **Foundation Layer**: Abstract base classes that define card contracts
- **Implementation Layer**: Concrete card types with specific behaviors
- **Ability Layer**: Interfaces for special card abilities
- **Engine Layer**: Game mechanics and card interaction systems
- **Platform Layer**: Advanced composition and tournament management

Think of this as building the "engine" of a trading card game - each layer adds functionality and intelligence to create engaging card interactions.

Chapter V

Exercise 0: Card Foundation

	Exercise0
ex0	
Directory: *ex*0/	
Files to Submit: `__init__.py, Card.py, CreatureCard.py, main.py`	
Authorized: `abc, typing, enum, print()`	

Foundation Layer: Building the Universal Card Blueprint.

In trading card games, thousands of different cards exist - creatures, spells, artifacts, and more. But how do you ensure all these different card types can work together in the same game? The answer: **abstract base classes**!

`Your Mission: Create the foundational blueprint that defines how ALL cards in DataDeck must behave. This is like creating the "universal card template" that ensures any card can be played in your game engine.`

Implementation:

- `__init__.py` - Package initialization file
- `Card.py` - The abstract foundation class
- `CreatureCard.py` - Your first concrete card type
- `main.py` - Demonstration script (required for all exercises)

Technical Requirements:
Class Signatures:

```
# Card (Abstract Base Class)
def __init__(self, name: str, cost: int, rarity: str)
def play(self, game_state: dict) -> dict
def get_card_info(self) -> dict
def is_playable(self, available_mana: int) -> bool

# CreatureCard (Concrete Implementation)
def __init__(self, name: str, cost: int, rarity: str, attack: int, health: int)
def play(self, game_state: dict) -> dict
def attack_target(self, target) -> dict
```

Card.py - Create an abstract base class with:

- Inherit from `ABC` (Abstract Base Class)
- Constructor: `__init__(self, name: str, cost: int, rarity: str)`
- Abstract method: `play(self, game_state: dict)`
- Concrete method: `get_card_info(self)`
- Concrete method: `is_playable(self, available_mana: int)`

CreatureCard.py - Create a concrete implementation that:

- Inherits from `Card`
- Adds attack and health attributes
- Implements the abstract `play` method
- Adds `attack_target` method for creature combat
- Validates that attack and health are positive integers

Expected Output Example:

```
$> python3 -m ex0.main

=== DataDeck Card Foundation ===

Testing Abstract Base Class Design:

CreatureCard Info:
{'name': 'Fire Dragon', 'cost': 5, 'rarity': 'Legendary',
'type': 'Creature', 'attack': 7, 'health': 5}

Playing Fire Dragon with 6 mana available:
Playable: True
Play result: {'card_played': 'Fire Dragon', 'mana_used': 5,
'effect': 'Creature summoned to battlefield'}

Fire Dragon attacks Goblin Warrior:
Attack result: {'attacker': 'Fire Dragon', 'target': 'Goblin Warrior',
'damage_dealt': 7, 'combat_resolved': True}

Testing insufficient mana (3 available):
Playable: False

Abstract pattern successfully demonstrated!
```

How do **abstract base classes** ensure consistency across different card types? What happens if you try to create a Card directly without implementing required methods?

Chapter VI

Exercise 1: Deck Builder

	Exercise1
ex1	
Directory: *ex*1/	
Files to Submit: `__init__.py, SpellCard.py, ArtifactCard.py, Deck.py, main.py`	
Authorized: `abc, typing, enum, random, print()`	

Implementation Layer: The Deck Builder System.

Your card foundation is solid! Now build different card types that can all work together in the same deck. Create a deck builder that can manage creatures, spells, and artifacts using the same interface.

Your Mission: Build multiple concrete card types that all implement your abstract interface. Create a deck management system that can handle any card type using your foundation.

Implementation:

- `SpellCard.py` - Instant magic effects
- `ArtifactCard.py` - Permanent game modifiers
- `Deck.py` - Deck management system
- `main.py` - Demonstration script

Import your Card from ex0 using: `from ex0.Card import Card`. Each exercise must include a main.py file that demonstrates the functionality. **Important:** Create an `__init__.py` file in each exercise directory to make it a Python package.

Technical Requirements: Class Signatures:

```
# SpellCard (Concrete Implementation)
def __init__(self, name: str, cost: int, rarity: str, effect_type: str)
def play(self, game_state: dict) -> dict
def resolve_effect(self, targets: list) -> dict

# ArtifactCard (Concrete Implementation)
def __init__(self, name: str, cost: int, rarity: str, durability: int, effect: str)
def play(self, game_state: dict) -> dict
def activate_ability(self) -> dict

# Deck (Management Class)
def add_card(self, card: Card) -> None
def remove_card(self, card_name: str) -> bool
def shuffle(self) -> None
def draw_card(self) -> Card
def get_deck_stats(self) -> dict
```

SpellCard.py:

- Processes instant magical effects
- Has effect_type attribute (damage, heal, buff, debuff)
- Implements `resolve_effect` for spell mechanics
- Spells are consumed when played (one-time use)

ArtifactCard.py:

- Represents permanent game modifiers
- Has durability attribute (how long it lasts)
- Has effect attribute describing the artifact's permanent ability
- Implements `activate_ability` for ongoing effects
- Artifacts remain in play until destroyed

Deck.py - Deck management class:

- Method: `add_card(card: Card)`
- Method: `remove_card(card_name: str)`
- Method: `shuffle()`
- Method: `draw_card()`
- Method: `get_deck_stats()`

Expected Output Example:

```
$> python3 -m ex1.main

=== DataDeck Deck Builder ===

Building deck with different card types...
Deck stats: {'total_cards': 3, 'creatures': 1, 'spells': 1,
'artifacts': 1, 'avg_cost': 4.0}

Drawing and playing cards:

Drew: Lightning Bolt (Spell)
Play result: {'card_played': 'Lightning Bolt', 'mana_used': 3,
'effect': 'Deal 3 damage to target'}

Drew: Mana Crystal (Artifact)
Play result: {'card_played': 'Mana Crystal', 'mana_used': 2,
'effect': 'Permanent: +1 mana per turn'}

Drew: Fire Dragon (Creature)
Play result: {'card_played': 'Fire Dragon', 'mana_used': 5,
'effect': 'Creature summoned to battlefield'}

Polymorphism in action: Same interface, different card behaviors!
```

How does **polymorphism** enable the Deck to work with any card type?
What are the benefits of this design pattern for card game systems?

Chapter VII

Exercise 2: Ability System

	Exercise2
ex2	
Directory: *ex2/*	
Files to Submit: `__init__.py, Combatable.py, Magical.py, EliteCard.py, main.py`	
Authorized: `abc, typing, enum, random, print()`	

Ability Layer: Multiple Interface Design.

Card games need flexible ability systems! Time to build the Ability Layer using multiple interfaces that can be combined to create powerful, versatile cards with multiple special abilities.

Your Mission: Design multiple abstract interfaces that can be combined using multiple inheritance. Create cards that implement combat, magic, and utility abilities simultaneously.

Implementation:

- `Combatable.py` - Abstract combat interface
- `Magical.py` - Abstract magic interface
- `EliteCard.py` - Multiple inheritance implementation
- `main.py` - Demonstration script

Import Card from ex0 in your EliteCard.py using: from ex0.Card import Card. Each exercise must include a main.py file that demonstrates the functionality. **Important:** Create an __init__.py file in each exercise directory.

Technical Requirements: Class Signatures:

```
# Combatable (Abstract Interface)
def attack(self, target) -> dict
def defend(self, incoming_damage: int) -> dict
def get_combat_stats(self) -> dict

# Magical (Abstract Interface)
def cast_spell(self, spell_name: str, targets: list) -> dict
def channel_mana(self, amount: int) -> dict
def get_magic_stats(self) -> dict

# EliteCard (Multiple Inheritance: Card + Combatable + Magical)
def play(self, game_state: dict) -> dict
def attack(self, target) -> dict
def cast_spell(self, spell_name: str, targets: list) -> dict
```

Combatable.py - Abstract combat interface:

- Abstract method: `attack(self, target)`
- Abstract method: `defend(self, incoming_damage: int)`
- Abstract method: `get_combat_stats(self)`

Magical.py - Abstract magic interface:

- Abstract method: `cast_spell(self, spell_name: str, targets: list)`
- Abstract method: `channel_mana(self, amount: int)`
- Abstract method: `get_magic_stats(self)`

EliteCard.py - Multiple inheritance class:

- Inherits from `Card`, `Combatable`, AND `Magical`
- Implements ALL abstract methods from all three interfaces
- Represents powerful cards with multiple abilities
- Combines combat prowess with magical capabilities

Expected Output Example:

```
$> python3 -m ex2.main

=== DataDeck Ability System ===

EliteCard capabilities:
- Card: ['play', 'get_card_info', 'is_playable']
- Combatable: ['attack', 'defend', 'get_combat_stats']
- Magical: ['cast_spell', 'channel_mana', 'get_magic_stats']

Playing Arcane Warrior (Elite Card):

Combat phase:
Attack result: {'attacker': 'Arcane Warrior', 'target': 'Enemy',
'damage': 5, 'combat_type': 'melee'}
Defense result: {'defender': 'Arcane Warrior', 'damage_taken': 2,
'damage_blocked': 3, 'still_alive': True}

Magic phase:
Spell cast: {'caster': 'Arcane Warrior', 'spell': 'Fireball',
'targets': ['Enemy1', 'Enemy2'], 'mana_used': 4}
Mana channel: {'channeled': 3, 'total_mana': 7}

Multiple interface implementation successful!
```

How do **multiple interfaces** enable flexible card design? What are the advantages of separating combat and magic concerns?

Chapter VIII

Exercise 3: Game Engine

	Exercise3
ex3	
Directory: *ex3/*	
Files to Submit: `__init__.py, GameStrategy.py, CardFactory.py, AggressiveStrategy.py, FantasyCardFactory.py, GameEngine.py, main.py`	
Authorized: `abc, typing, enum, random, print()`	

Engine Layer: Strategy and Factory Patterns.

Time to build the Game Engine - the brain of DataDeck that orchestrates complex card interactions using advanced abstract patterns.

Your Mission: Create a sophisticated game system using the Abstract Factory Pattern combined with Strategy Pattern. Build a system that can create different card types and apply different game strategies dynamically.

Implementation:

- `GameStrategy.py` - Abstract strategy interface
- `CardFactory.py` - Abstract factory interface
- `AggressiveStrategy.py` - Concrete aggressive strategy
- `FantasyCardFactory.py` - Concrete factory implementation
- `GameEngine.py` - Game orchestrator
- `main.py` - Demonstration script

Import interfaces from previous exercises as needed: from ex0.Card import Card, from ex1.SpellCard import SpellCard, etc. Each exercise must include a main.py file that demonstrates the functionality. **Important:** Create an __init__.py file in each exercise directory.

Technical Requirements: Class Signatures:

```
# GameStrategy (Abstract Interface)
def execute_turn(self, hand: list, battlefield: list) -> dict
def get_strategy_name(self) -> str
def prioritize_targets(self, available_targets: list) -> list

# CardFactory (Abstract Factory Interface)
def create_creature(self, name_or_power: str | int | None = None) -> Card
def create_spell(self, name_or_power: str | int | None = None) -> Card
def create_artifact(self, name_or_power: str | int | None = None) -> Card
def create_themed_deck(self, size: int) -> dict
def get_supported_types(self) -> dict

# AggressiveStrategy (Concrete Strategy)
def execute_turn(self, hand: list, battlefield: list) -> dict
def get_strategy_name(self) -> str
def prioritize_targets(self, available_targets: list) -> list

# GameEngine (Game Orchestrator)
def configure_engine(self, factory: CardFactory, strategy: GameStrategy) -> None
def simulate_turn(self) -> dict
def get_engine_status(self) -> dict
```

GameStrategy.py - Abstract strategy interface:

- Abstract method: `execute_turn(self, hand: list, battlefield: list)`
- Abstract method: `get_strategy_name(self)`
- Abstract method: `prioritize_targets(self, available_targets: list)`

CardFactory.py - Abstract factory interface:

- Abstract method: `create_creature(self, name_or_power: str | int | None = None)`
- Abstract method: `create_spell(self, name_or_power: str | int | None = None)`
- Abstract method: `create_artifact(self, name_or_power: str | int | None = None)`
- Abstract method: `create_themed_deck(self, size: int)`
- Abstract method: `get_supported_types(self)`

AggressiveStrategy.py - Concrete strategy:

- Prioritizes attacking and dealing damage
- Plays low-cost creatures first for board presence

- Targets enemy creatures and player directly
- Returns comprehensive turn execution results

FantasyCardFactory.py - Concrete factory:

- Creates fantasy-themed creatures (Dragons, Goblins, etc.)
- Creates elemental spells (Fire, Ice, Lightning)
- Creates magical artifacts (Rings, Staffs, Crystals)
- Supports extensible card type registration

GameEngine.py - Game orchestrator:

- Method: `configure_engine(factory, strategy)`
- Method: `simulate_turn()`
- Method: `get_engine_status()`

Expected Output Example:

```
$> python3 -m ex3.main

=== DataDeck Game Engine ===

Configuring Fantasy Card Game...
Factory: FantasyCardFactory
Strategy: AggressiveStrategy
Available types: {'creatures': ['dragon', 'goblin'], 'spells': ['fireball'],
'artifacts': ['mana_ring']}

Simulating aggressive turn...
Hand: [Fire Dragon (5), Goblin Warrior (2), Lightning Bolt (3)]

Turn execution:
Strategy: AggressiveStrategy
Actions: {'cards_played': ['Goblin Warrior', 'Lightning Bolt'],
'mana_used': 5, 'targets_attacked': ['Enemy Player'],
'damage_dealt': 8}

Game Report:
{'turns_simulated': 1, 'strategy_used': 'AggressiveStrategy',
'total_damage': 8, 'cards_created': 3}

Abstract Factory + Strategy Pattern: Maximum flexibility achieved!
```

How do **Abstract Factory** and **Strategy** patterns work together? What makes this combination powerful for game engine systems?

Chapter IX

Exercise 4: Tournament Platform

	Exercise4
ex4	
Directory: *ex4/*	
Files to Submit: `__init__.py, Rankable.py, TournamentCard.py, TournamentPlatform.py, main.py`	
Authorized: `abc, typing, enum, random, print()`	

Platform Layer: Advanced Interface Composition.

Time to combine everything you've learned into a unified tournament system that demonstrates mastery of abstract programming patterns.

Your Mission: Create a comprehensive tournament platform that combines abstract classes, multiple interfaces, and factory patterns into one cohesive system. Build a platform that can manage tournaments with different card types and strategies.

Implementation:

- `Rankable.py` - Simple ranking interface
- `TournamentCard.py` - Card with tournament capabilities
- `TournamentPlatform.py` - Platform management system
- `main.py` - Demonstration script

Import and combine classes from previous exercises: from ex0.Card import Card, from ex2.Combatable import Combatable, etc. Each exercise must include a main.py file that demonstrates the functionality. **Important:** Create an __init__.py file in each exercise directory.

Technical Requirements: Class Signatures:

```
# Rankable (Abstract Interface)
def calculate_rating(self) -> int
def update_wins(self, wins: int) -> None
def update_losses(self, losses: int) -> None
def get_rank_info(self) -> dict

# TournamentCard (Multiple Inheritance: Card + Combatable + Rankable)
def play(self, game_state: dict) -> dict
def attack(self, target) -> dict
def calculate_rating(self) -> int
def get_tournament_stats(self) -> dict

# TournamentPlatform (Platform Management)
def register_card(self, card: TournamentCard) -> str
def create_match(self, card1_id: str, card2_id: str) -> dict
def get_leaderboard(self) -> list
def generate_tournament_report(self) -> dict
```

Rankable.py - Simple ranking interface:

- Abstract method: `calculate_rating(self)`
- Abstract method: `update_wins(self, wins: int)`
- Abstract method: `update_losses(self, losses: int)`
- Abstract method: `get_rank_info(self)`

TournamentCard.py - Enhanced card class:

- Inherits from Card, Combatable, and Rankable
- Implements all abstract methods from all three interfaces
- Tracks tournament performance (wins, losses, rating)
- Processes tournament matches with ranking updates

TournamentPlatform.py - Platform management system:

- Method: `register_card(card: TournamentCard)`
- Method: `create_match(card1_id: str, card2_id: str)`
- Method: `get_leaderboard()`
- Method: `generate_tournament_report()`

Expected Output Example:

```
$> python3 -m ex4.main

=== DataDeck Tournament Platform ===

Registering Tournament Cards...

Fire Dragon (ID: dragon_001):
- Interfaces: [Card, Combatable, Rankable]
- Rating: 1200
- Record: 0-0

Ice Wizard (ID: wizard_001):
- Interfaces: [Card, Combatable, Rankable]
- Rating: 1150
- Record: 0-0

Creating tournament match...
Match result: {'winner': 'dragon_001', 'loser': 'wizard_001',
'winner_rating': 1216, 'loser_rating': 1134}

Tournament Leaderboard:
1. Fire Dragon - Rating: 1216 (1-0)
2. Ice Wizard - Rating: 1134 (0-1)

Platform Report:
{'total_cards': 2, 'matches_played': 1,
'avg_rating': 1175, 'platform_status': 'active'}

=== Tournament Platform Successfully Deployed! ===
All abstract patterns working together harmoniously!
```

How does **multiple inheritance** allow a class to implement several interfaces? What are the benefits of combining ranking capabilities with card game mechanics?

Chapter X

Submission

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double check the names of your files to ensure they are correct.

During evaluation, you may be asked to explain abstract classes, demonstrate polymorphism, or show how interfaces enable multiple inheritance. Focus on understanding the concepts, not just the implementation.

Keep card game logic simple - the focus is on demonstrating abstract programming patterns. Each exercise builds on the previous one, so make sure your imports work correctly.

Abstract programming patterns are the foundation of extensible software design. You've learned how to create systems that can grow and adapt to new card types and game mechanics!