Laborator 9 – Clase abstracte si interfete

Tema: Adventure Game

Autor: ing. Eduard Donea

Poveste

Intr-o lume plina de creaturi misterioase, coridoare intunecate si comori ascunse, un erou solitar porneste intr-o aventura necunoscuta. Pentru a supravietui, acesta trebuie sa interactioneze cu diverse elemente ale mediului, sa deschida usi, sa colecteze obiecte, sa discute cu personaje si sa infrunte inamici periculosi. Sistemul pe care il dezvoltam astazi este o simulare simplificata a unui joc de aventura, care demonstreaza clar utilizarea conceptelor de **clase abstracte**, **interfete** si **polimorfism la timpul de executie**.

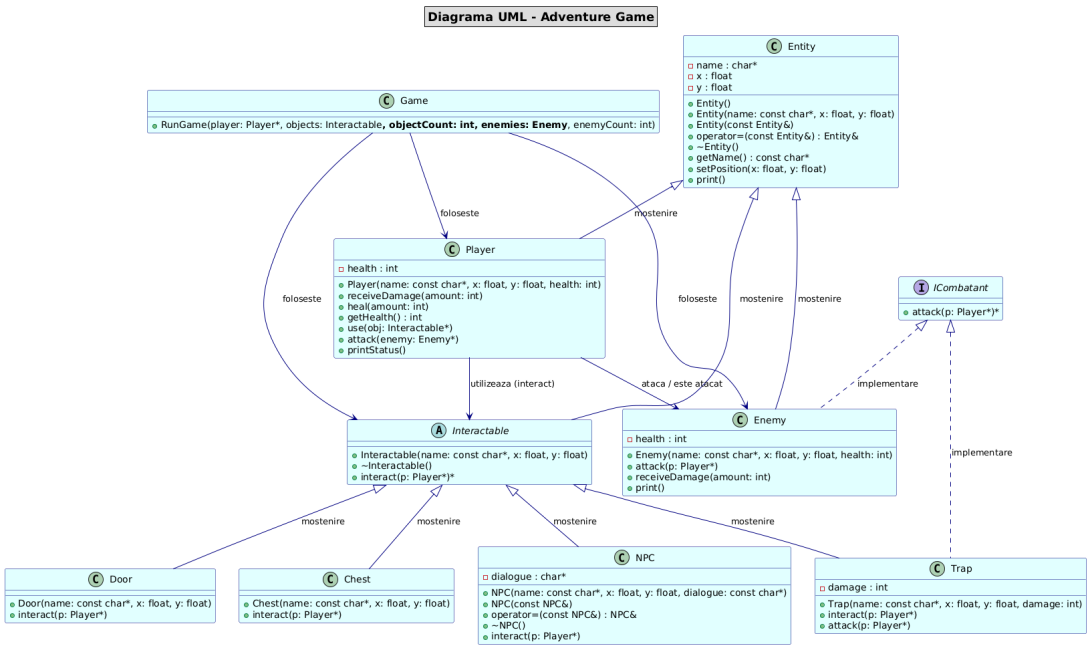
Scopul laboratorului

- intelegerea conceptului de **clasa abstracta** si **functie virtuala pura**;
- implementarea unei **interfete** in stil C++;
- folosirea polimorfismului la runtime folosind pointeri la clasa de baza;
- aplicarea mostenirii si diferentierea relatiei `is-a` fata de utilizare (`uses`);
- organizarea corecta a fisierelor intr-un proiect modular C++.

Structura fișierelor

- **Entity.h / Entity.cpp** – clasa de baza pentru toate entitatile;
- **Interactable.h / Interactable.cpp** – clasa abstracta pentru obiectele cu care Player-ul poate interactiona;
- **ICombatant.h** – interfata pentru entitati care pot ataca;
- **Player.h / Player.cpp** – clasa Player;
- **Enemy.h / Enemy.cpp** – clasa inamicilor;
- **Door.h / Door.cpp** – usa interactiva;
- **Chest.h / Chest.cpp** – cufar cu obiecte;
- **NPC.h / NPC.cpp** – personaj non-jucator;
- **Trap.h / Trap.cpp** – capcana ce cauzeaza damage;
- **Game.h / Game.cpp** – logica jocului (RunGame este complet implementata);
- **main.cpp** – fisierul principal de testare (cu TODO-uri).

Figura 1 – Diagrama UML a sistemului Adventure Game



Fișierul `Entity.h`

```h
#ifndef ENTITY_H
#define ENTITY_H

#include <iostream>
#include <cstring>

class Entity {
protected:
    char* name;
    float x, y;

public:
    Entity();
    Entity(const char* name, float x, float y);
    Entity(const Entity& e);
    Entity& operator=(const Entity& e);
    virtual ~Entity();

    const char* getName() const;
    void setPosition(float x, float y);
    void print() const;
};

#endif
```

Explicație: Clasa de bază `Entity`

Clasa `Entity` reprezintă fundamentul pentru toate obiectele care există în lumea jocului. Ea conține atribute comune tuturor entităților, precum: **numele** și **poziția** acestora în spațiu (`x`, `y`).

Scopul clasei este de a evita duplicarea codului: atât `Player`, cât și `Enemy`, `Door`, `NPC`, `Chest` sau `Trap` moștenesc `Entity` și primesc automat aceste atribute și metode.

Clasa se ocupă și de gestionarea memoriei pentru nume (`char*`), implementând constructori, destructor, copiere și operatorul de atribuire. Astfel se asigură un comportament corect în tot proiectul.

Fișierul `Entity.cpp`

```cpp
#include "Entity.h"

Entity::Entity() {
    // TODO: finalizati constructorul
}

Entity::Entity(const char* n, float xPos, float yPos) {
    // TODO: finalizati constructorul
}

Entity::Entity(const Entity& e) {
    // TODO: copiere profunda pentru name + x,y
}

Entity& Entity::operator=(const Entity& e) {
    // TODO: evitati auto-atribuirea, refaceti deep copy
}

Entity::~Entity() {
    // TODO: eliberati memoria
}

const char* Entity::getName() const {
    // TODO: returnati numele
}

void Entity::setPosition(float newX, float newY) {
    // TODO: setati coordonatele
}

void Entity::print() const {
    // TODO: afisati numele si pozitia entitatii
}
```

Fișierul `Interactable.h`

```h
#ifndef INTERACTABLE_H
#define INTERACTABLE_H

#include "Entity.h"

class Player; // Forward declaration

class Interactable : public Entity {
public:
    Interactable(const char* name, float x, float y);
    virtual ~Interactable();
    virtual void interact(Player* p) = 0; // metoda virtuala pura
};

#endif
```

Explicație: Clasa abstractă `Interactable`

Clasa `Interactable` reprezintă un tip special de entitate care poate fi **folosită, activată sau declanșată** de către jucător. Ea este definită ca **abstractă** deoarece conține cel puțin o metodă virtuală pură, ceea ce înseamnă că nu poate fi instanțiată direct.

Scopul clasei este de a oferi un **contract comun** tuturor obiectelor din joc care necesită o interacțiune — precum uși, cufere, capcane sau NPC-uri. Fiecare clasă derivată trebuie să implementeze propria versiune a metodei `interact()`, definind modul concret în care jucătorul interacționează cu acel obiect.

Acest mecanism permite utilizarea polimorfismului: jocul poate lucra cu pointeri la `Interactable` fără să știe tipul concret al obiectului, iar comportamentul specific este determinat la timpul de execuție.

Explicație: Forward Declaration

Un *forward declaration* este o declarație prin care anunțăm compilatorului existența unei clase înainte de utilizare, fără a avea încă definiția completă. Se folosește pentru a evita includerile circulare și pentru a reduce dependențele dintre fișiere atunci când folosim DOAR pointeri sau referințe.

Fișierul `Interactable.cpp`

```cpp
#include "Interactable.h"

Interactable::Interactable(const char* n, float x, float y) {
    // TODO: finalizati constructorul
}

Interactable::~Interactable() {}
```

Fișierul `ICombatant.h`

```h
#ifndef ICOMBATANT_H
#define ICOMBATANT_H

class Player; // Forward declaration

class ICombatant {
public:
    virtual ~ICombatant() = 0; // destructor virtual pur
    virtual void attack(Player* p) = 0; // metoda virtuala pura
};

#endif
```

Explicație: Interfața `ICombatant`

În C++, o „interfață” este o clasă care conține doar metode *virtuale pure* (adică metode declarate cu `= 0`) și nu are implementări. Scopul ei este de a impune un anumit comportament claselor care o moștenesc.

Interfața `ICombatant` definește metoda virtuală pură `attack()`, obligând orice clasă derivată (precum `Enemy` sau `Trap`) să implementeze propriul comportament de atac.

Acest lucru permite polimorfismul la runtime: putem apela `attack()` pe pointeri la interfață, iar funcția corespunzătoare obiectului real va fi executată.

Fișierul `ICombatant.cpp`

```cpp
#include "ICombatant.h"

ICombatant::~ICombatant() {}
```

Fișierul `Player.h`

```h
#ifndef PLAYER_H
#define PLAYER_H

#include "Entity.h"
#include "Interactable.h"
#include "Enemy.h"

class Interactable; // Forward declaration
class Enemy; // Forward declaration

class Player : public Entity {
private:
    int health;

public:
    Player(const char* name, float x, float y, int health);

    void receiveDamage(int amount);
    void heal(int amount);
    int getHealth() const;

    void use(Interactable* obj);
    void attack(Enemy* enemy);

    void printStatus() const;
};

#endif
```

Explicație: Clasa `Player`

Clasa `Player` reprezintă personajul principal controlat de utilizator în cadrul jocului. Ea moștenește funcționalitatea de bază din clasa `Entity` (nume și poziție), la care adaugă atribute specifice unui erou, precum **viața (health)** și acțiuni caracteristice unui personaj jucător.

Rolul clasei este de a centraliza toate comportamentele asociate jucătorului: primirea de damage, vindecarea, interacțiunea cu obiectele din joc și atacarea inamicilor. În acest fel, `Player` devine punctul principal prin care logica jocului se desfășoară, fiind implicat în toate tipurile de interacțiuni cu mediul.

Clasa favorizează utilizarea polimorfismului prin metode precum `use()` sau `attack()`, care acceptă obiecte derivate din clase abstracte sau interfețe. Astfel, comportamentul concret depinde de tipul obiectului cu care jucătorul interacționează, contribuind la flexibilitatea întregului sistem.

Fișierul `Player.cpp`

```cpp
#include "Player.h"

Player::Player(const char* n, float xPos, float yPos, int h) {
    // TODO: finalizati constructorul
}

void Player::receiveDamage(int amount) {
    // TODO: scadeti viata si afisati mesajul
}

void Player::heal(int amount) {
    // TODO: cresteti viata si afisati mesajul
}

int Player::getHealth() const {
    // TODO: returnati viata
}

void Player::use(Interactable* obj) {
    // TODO: apelati functia virtuala interact(this)
}

void Player::attack(Enemy* enemy) {
    // TODO: afisati mesaj si apelati receiveDamage() pe inamic
}

void Player::printStatus() const {
    // TODO: afisati HP-ul playerului
}
```

Fișierul `Enemy.h`

```h
#ifndef ENEMY_H
#define ENEMY_H

#include "Entity.h"
#include "ICombatant.h"
#include "Player.h"

class Player; // Forward declaration

class Enemy : public Entity, public ICombatant {
private:
    int health;

public:
    Enemy(const char* name, float x, float y, int health);

    void attack(Player* p) override;
    void receiveDamage(int amount);
    void print() const;
};

#endif
```

Explicație: Clasa `Enemy`

Clasa `Enemy` reprezintă entitățile ostile pe care jucătorul le poate întâlni în timpul aventurii. La fel ca `Player`, ea moștenește atributele și comportamentele de bază din clasa `Entity`, însă este extinsă cu elemente specifice inamicilor, precum **viața proprie** și capacitatea de a provoca damage.

Această clasă implementează interfața `ICombatant`, ceea ce o obligă să definească metoda `attack()`, asigurând un comportament concret în situații de luptă. Astfel, orice instanță de `Enemy` poate ataca direct jucătorul, determinând logica de combat.

Prin integrarea sa în sistemul de polimorfism, `Enemy` permite jocului să folosească pointeri la interfață sau la clasa de bază pentru a gestiona diferite tipuri de adversari, fiecare cu propriul comportament, fără a necesita cunoașterea tipului concret la compilare. Acest lucru oferă flexibilitate în extinderea jocului cu noi tipuri de inamici.

Fișierul `Enemy.cpp`

```cpp
#include "Enemy.h"

Enemy::Enemy(const char* n, float xPos, float yPos, int h) {
    // TODO: finalizati constructorul
}

void Enemy::attack(Player* p) {
    // TODO: implementati atacul asupra playerului
}

void Enemy::receiveDamage(int amount) {
    // TODO: scadeti viata inamicului si afisati mesajul
}

void Enemy::print() const {
    // TODO: afisati informatii despre inamic
}
```

Fișierul `Door.h`

```h
#ifndef DOOR_H
#define DOOR_H

#include "Interactable.h"
#include "Player.h"

class Door : public Interactable {
public:
    Door(const char* name, float x, float y);
    void interact(Player* p) override;
};

#endif
```

Explicație: Clasa `Door`

Clasa `Door` reprezintă un obiect interactiv simplu din joc, derivat din clasa abstractă `Interactable`. Rolul său este de a modela o ușă care poate fi deschisă de jucător atunci când acesta interacționează cu ea.

Deoarece moștenește `Interactable`, clasa este obligată să implementeze metoda `interact()`, care definește comportamentul concret al ușii la momentul utilizării. În acest caz, interacțiunea se rezumă la un mesaj sau o acțiune simplă prin care ușa este „deschisă”, permițând progresul jucătorului în aventură.

`Door` nu are atribute suplimentare față de clasa de bază, fiind un exemplu ideal de obiect cu comportament specific, dar structură minimală, folosit pentru a ilustra principiile de polimorfism și moștenire abstractă.

Fișierul `Door.cpp`

```cpp
#include "Door.h"

Door::Door(const char* n, float xPos, float yPos) {
    // TODO: finalizati constructorul
}

void Door::interact(Player* p) {
    // TODO: afisati mesaj despre usa deschisa
}
```

Fișierul `Chest.h`

```cpp
#ifndef CHEST_H
#define CHEST_H

#include "Interactable.h"
#include "Player.h"

class Chest : public Interactable {
public:
    Chest(const char* name, float x, float y);
    void interact(Player* p) override;
};

#endif
```

Explicație: Clasa `Chest`

Clasa `Chest` reprezintă un obiect interactiv derivat din clasa abstractă `Interactable`. Acesta modelează un cufăr pe care jucătorul îl poate deschide în timpul aventurii. `Chest` oferă un comportament concret pentru metoda `interact()`, stabilind ce se întâmplă atunci când jucătorul utilizează cufărul.

În această implementare, interacțiunea cu un cufăr aduce un beneficiu direct jucătorului — de obicei vindecare sau un efect pozitiv similar. Prin urmare, clasa este un exemplu de obiect prietenos din joc, care contribuie la progresul și supraviețuirea jucătorului.

`Chest` demonstrează clar modul în care clasele derivate își definesc propriul comportament specific, păstrând în același timp structura și contractul impus de clasa abstractă `Interactable`.

Fișierul `Chest.cpp`

```cpp
#include "Chest.h"

Chest::Chest(const char* n, float xPos, float yPos) {
    // TODO: finalizati constructorul
}

void Chest::interact(Player* p) {
    // TODO: afisati mesaj si apelati p->heal()
}
```

Fișierul `NPC.h`

```h
#ifndef NPC_H
#define NPC_H

#include "Interactable.h"
#include "Player.h"

class NPC : public Interactable {
private:
    char* dialogue;

public:
    NPC(const char* name, float x, float y, const char* dialogue);
    NPC(const NPC& n);
    NPC& operator=(const NPC& n);
    ~NPC();

    void interact(Player* p) override;
};

#endif
```

Explicație: Clasa `NPC`

Clasa `NPC` (Non-Player Character) reprezintă personaje din joc care nu sunt controlate de jucător, dar care pot interacționa cu acesta prin dialog sau mesaje. Ea moștenește comportamentul de bază din `Interactable`, fiind astfel obligată să implementeze metoda `interact()`, care definește felul în care personajul reacționează atunci când jucătorul interacționează cu el.

Spre deosebire de alte obiecte interactive, `NPC` stochează și un text de dialog, pe care îl afișează jucătorului în timpul interacțiunii. Acest lucru îl face util pentru transmiterea de indicii, informații narative sau mesaje care îmbogățesc atmosfera jocului.

Clasa gestionează manual memoria pentru textul dialogului, demonstrând importanța implementării constructorilor, destructorilor și a copierii corecte (deep copy) în aplicațiile orientate pe obiecte.

Fișierul `NPC.cpp`

```cpp
#include "NPC.h"

NPC::NPC(const char* n, float xPos, float yPos, const char* d) {
    // TODO: finalizati constructorul
}

NPC::NPC(const NPC& n) : Interactable(n) {
    // TODO: deep copy pentru dialogue
}

NPC& NPC::operator=(const NPC& n) {
    // TODO: deep copy + auto-atribuire
}

NPC::~NPC() {
    // TODO: eliberati memoria
}

void NPC::interact(Player* p) {
    // TODO: afisati dialogul pentru player
}
```

Fișierul `Trap.h`

```h
#ifndef TRAP_H
#define TRAP_H

#include "Interactable.h"
#include "ICombatant.h"
#include "Player.h"

class Trap : public Interactable, public ICombatant {
private:
    int damage;

public:
    Trap(const char* name, float x, float y, int damage);

    void interact(Player* p) override;
    void attack(Player* p) override;
};

#endif
```

Explicație: Clasa `Trap`

Clasa `Trap` reprezintă o capcană din joc, adică un obiect interactiv care provoacă damage jucătorului atunci când acesta interacționează cu el. Moștenește atât `Interactable`, cât și interfața `ICombatant`, fiind astfel obligată să definească un comportament concret pentru metoda `interact()` și pentru metoda `attack()`.

Rolul său este de a introduce un element de risc în aventură, demonstrând cum moștenirea multiplă permite combinarea comportamentelor unui obiect static cu cele ale unui atacator.

Fișierul `Trap.cpp`

```cpp
#include "Trap.h"

Trap::Trap(const char* n, float xPos, float yPos, int d) {
    // TODO: finalizati constructorul
}

void Trap::interact(Player* p) {
    // TODO: afisati mesaj si apelati attack()
}

void Trap::attack(Player* p) {
    // TODO: damage aplicat playerului prin p->receiveDamage()
}
```

Fișierul `Game.h`

```h
#ifndef GAME_H
#define GAME_H

#include "Player.h"
#include "Interactable.h"
#include "Enemy.h"

class Game {
public:
    void RunGame(Player* player, Interactable** objects, int objectCount,
                 Enemy** enemies, int enemyCount);
};

#endif
```

Explicație: Clasa `Game`

Clasa `Game` are rolul de a coordona întregul flux al aventurii și reprezintă elementul central al logicii jocului. Spre deosebire de celelalte clase din proiect, care modelează obiecte, entități și interacțiuni locale, `Game` este responsabilă pentru **organizarea secvențelor** în care aceste elemente sunt folosite. Ea oferă o structură de tip „motor de joc" simplificat, în care obiectele, capcanele și inamicii sunt traversați în ordine, iar jucătorul este ghidat prin fiecare etapă a aventurii.

Metoda principală, `RunGame()`, primește jucătorul, lista de obiecte interactive și lista de inamici, apoi controlează modul în care aceștia sunt procesati: jucătorul interacționează pe rând cu fiecare obiect, iar inamicii apar periodic pe parcurs. În cadrul acestei metode se execută toate interacțiunile: deschiderea ușilor, colectarea cufărelor, dialogurile cu NPC-urile, declanșarea capcanelor și luptele cu inamicii.

Clasa `Game` demonstrează clar utilitatea polimorfismului, deoarece lucrează exclusiv cu pointeri către clase de bază (`Interactable` și `Enemy`), fără să cunoască tipurile concrete ale obiectelor din joc. Comportamentul fiecărui obiect este determinat la timpul de execuție prin mecanismul de legare dinamică, ceea ce permite extinderea ulterioară a sistemului fără a modifica logica de joc.

Prin faptul că structura aventurii este complet definită în interiorul clasei `Game`, iar celelalte clase oferă doar comportamente locale, modelul rezultat separă clar **logica globală a jocului** de **detaliile fiecărei entități**. Această separare este un principiu important în proiectarea sistemelor orientate pe obiecte.

Fișierul `Game.cpp`

```cpp
#include "Game.h"

void Game::RunGame(Player* player, Interactable** objects, int objectCount,
                   Enemy** enemies, int enemyCount) {

    std::cout << "\n=== GAME START ===\n";
    player->printStatus();
    std::cout << "\n";

    int enemyIndex = 0;

    for (int i = 0; i < objectCount; i++) {
        std::cout << "\n--- Step " << (i+1) << " ---\n";
        std::cout << player->getName() << " approaches: " <<
        ↪  objects[i]->getName() << "\n";

        player->use(objects[i]);

        if (player->getHealth() <= 0) {
            std::cout << "\n>>> " << player->getName() << " has died...\n";
            std::cout << "=== GAME OVER ===\n";
            return;
        }

        if (enemyIndex < enemyCount && (i % 2 == 1)) {
            std::cout << "\n!!! Enemy Encounter !!!\n";
            enemies[enemyIndex]->print();
            enemies[enemyIndex]->attack(player);

            if (player->getHealth() <= 0) {
                std::cout << "\n>>> " << player->getName() << " has been slain
                ↪  by an enemy...\n";
                std::cout << "=== GAME OVER ===\n";
                return;
            }

            player->attack(enemies[enemyIndex]);
            enemyIndex++;
        }

        player->printStatus();
    }

    std::cout << "\n=== ADVENTURE COMPLETE! ===\n";
    std::cout << player->getName() << " survives with HP: " <<
    ↪  player->getHealth() << "\n\n";
}
```

Fișierul `main.cpp`

```cpp
#include "Player.h"
#include "Enemy.h"
#include "NPC.h"
#include "Door.h"
#include "Chest.h"
#include "Trap.h"
#include "Game.h"

int main() {
    // TODO: creati un Player*

    // TODO: creati un vector de Interactable** objects si initializati-l

    // TODO: creati un vector de Enemy** enemies si initializati-l

    // TODO: apelati RunGame(...) din clasa Game cu parametrii corecti

    // TODO: eliberati memoria alocata (delete/destroy)

    return 0;
}
```

În caz că nu vă descurcați să scrieți de mână:
Puteți descărca proiectul pregătit (cu toate fișierle și TODO-urile incluse):
Descarcă arhiva de pe Google Drive

Cerințe (10p)

1. (2p) Implementati clasa `Entity` si gestionati corect memoria.
2. (2p) Implementati clasa abstracta `Interactable` si clasele derivate.
3. (2p) Implementati interfata `ICombatant` si clasele care o folosesc.
4. (2p) Implementati clasele `Player`, `Enemy` si comportamentul lor.
5. (1p) Creati o simulare completa in `main.cpp` folosind metoda din clasa `Game`.
6. (1p) Eliberati memoria si verificati totul cu `valgrind`.

Compilare si rulare

```
g++ -g *.cpp -o lab9
valgrind ./lab9
```

Bonus GitHub Classroom

Studentii pot obtine pana la **3p bonus** daca incarca toate laboratoarele pana la finalul semestrului, respectiv **1p bonus** daca incarca mai mult de jumatate dintre ele.

Pasii pentru incarcarea laboratorului:

1. Accesati cursul de pe OCW – Programare Orientata pe Obiecte, sectiunea **Resurse** → **Bonus GitHub laborator**.
2. Acolo veti gasi linkul **GitHub Classroom** pentru Laboratorul 9:

   `https://classroom.github.com/a/1HmGkW56`

3. Dupa acceptare, veti primi propriul repository, ex. `laborator-9-NumePrenume`.
4. Clonati repository-ul local:

```
git clone
↪  git@github.com:Laborator-POO-2025-2026/laborator-9-NumePrenume.git
cd laborator-9-NumePrenume
```

5. Dupa ce ati adaugat fisierele, urmeaza comenzile:

```
git add .
git commit -m "Mesaj de commit"
git push origin main
```

Atenție: laboratoarele copiate între studenți sunt considerate nevalide, iar niciunul dintre cei doi participanți nu va primi bonusul. Repository-urile sunt monitorizate automat de sistemul GitHub Classroom.