Laborator 9 – Clase abstracte si interfete

Tema: Mystery House - Phasmofobia Remake 👻

Autor: ing. Eduard Donea

Poveste

Într-o casă bântuită, plină de camere întunecate, zgomote ciudate și prezențe nevăzute, un investigator paranormal pășește în necunoscut pentru a colecta indicii și a identifica tipul de fantomă care locuiește acolo. Pe parcurs, acesta trebuie să interacționeze cu obiecte misterioase, să deschidă uși încuiate, să evite atacurile spiritelor și să supraviețuiască suficient de mult pentru a rezolva cazul. Tema de astăzi reprezintă o simulare simplificată inspirată din jocul **Phasmophobia**, construită pentru a demonstra folosirea **claselor abstracte**, **interfetelor**, **mostenirii**, **polimorfismului la runtime** și **gestionării memoriei** într-un proiect modular orientat-obiect.

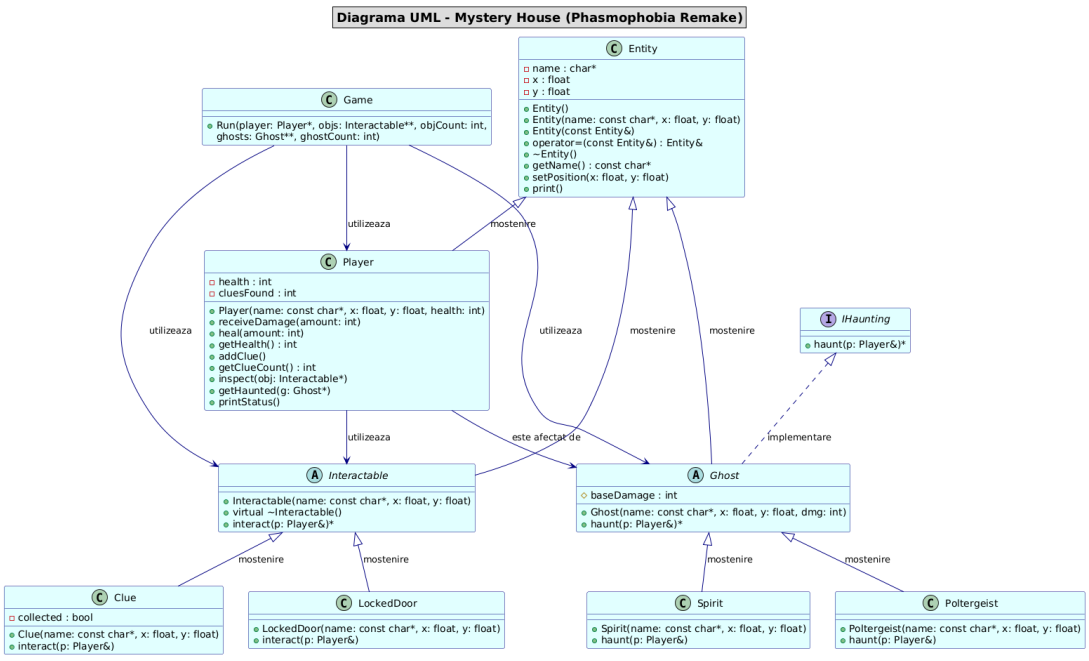
Scopul laboratorului

- intelegerea conceptului de clasa abstracta si metoda virtuala pura;
- implementarea unei interfete in C++;
- folosirea pointerilor catre clase abstracte pentru polimorfism la timpul executiei;
- demonstrarea mostenirii si a relatiilor dintre clase;
- aplicarea conceptelor in cadrul unui sistem 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;
- **IHaunting.h / IHaunting.cpp** – interfata pentru entitatile care pot ataca jucatorul;
- **Player.h / Player.cpp** – clasa jucatorului;
- **Ghost.h / Ghost.cpp** – clasa abstracta pentru toate tipurile de fantome;
- **Spirit.h / Spirit.cpp** – fantomă simplă;
- **Poltergeist.h / Poltergeist.cpp** – fantomă agresivă;
- **Clue.h / Clue.cpp** – obiect interactiv ce reprezintă un indiciu colectabil;
- **LockedDoor.h / LockedDoor.cpp** – ușă încuiată ce poate fi deschisă;
- **Game.h / Game.cpp** – logica principală a jocului;
- **main.cpp** – fișierul principal pentru simulare.

Figura 1 – Diagrama UML



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* n, float xx, float yy);
    Entity(const Entity& e);
    Entity& operator=(const Entity& e);
    virtual ~Entity();

    const char* getName() const;
    void setPosition(float xx, float yy);
    void print() const;
};

#endif
```

Explicație: Clasa de bază `Entity`

Clasa `Entity` este fundamentul tuturor obiectelor și creaturilor din joc. Ea conține **numele** entității și poziția `(x, y)` în lumea virtuală. Prin moștenirea acestei clase, toate celelalte elemente ale sistemului primesc automat această funcționalitate comună, evitând duplicarea codului.

Clasa gestionează memoria pentru nume și implementează constructori, destructor, constructor de copiere și operator de atribuire pentru a asigura o gestionare corectă a resurselor.

Fișierul `Entity.cpp`

```cpp
#include "Entity.h"

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

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

Entity::Entity(const Entity& e) {
    // TODO: finalizati constructorul
}

Entity& Entity::operator=(const Entity& e) {
    // TODO: finalizati operatorul de atribuire
}

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

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

void Entity::setPosition(float xx, float yy) {
    // TODO: setati x si y
}

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

Fișierul `Interactable.h`

```h
#ifndef INTERACTABLE_H
#define INTERACTABLE_H

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

class Player; // forward declaration

class Interactable : public Entity {
public:
    Interactable(const char* n, float x, float y);
    virtual ~Interactable();

    virtual void interact(Player* p) = 0; // metoda virtuala pura
};

#endif
```

Explicație: Clasa abstractă `Interactable`

Clasa `Interactable` modelează toate obiectele din casă cu care jucătorul poate interacționa: indicii, uși, obiecte paranormale etc. Este o **clasă abstractă** întrucât conține o metodă virtuală pură `interact()`.

Fiecare clasă derivată implementează propria versiune a interacțiunii, permițând folosirea polimorfismului la runtime. Astfel, jocul poate trata uniform toate obiectele, fără a cunoaște tipul lor real.

Explicație: Forward Declaration

Se folosește pentru a evita includerile circulare între clase atunci când folosim pointeri sau referințe către acestea. Compilatorului i se spune doar că o clasă există, fără detalii despre implementarea ei.

Fișierul `Interactable.cpp`

```cpp
#include "Interactable.h"

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

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

Fișierul `IHaunting.h`

```cpp
#ifndef IHAUNTING_H
#define IHAUNTING_H

class Player; // forward declaration

class IHaunting {
public:
    virtual ~IHaunting() = 0; // destructor virtual pur

    virtual void haunt(Player* p) = 0; // metoda virtuala pura
};

#endif
```

Explicație: Interfața `IHaunting`

Interfața definește comportamentul specific tuturor entităților capabile să atace jucătorul. Ea conține metoda virtuală pură `haunt()`, pe care fiecare tip de fantomă trebuie să o implementeze diferit.

Prin folosirea interfețelor, sistemul permite un design flexibil în care diferite tipuri de fantome pot coexista și pot fi tratate prin pointeri la aceeași bază.

Fișierul `IHaunting.cpp`

```cpp
#include "IHaunting.h"

IHaunting::~IHaunting() {}
```

Fișierul `Player.h`

```h
#ifndef PLAYER_H
#define PLAYER_H

#include "Entity.h"
#include "Interactable.h"
#include "Ghost.h"

class Interactable; // forward declaration
class Ghost; // forward declaration

class Player : public Entity {
private:
    int health;
    int cluesFound;

public:
    Player(const char* n, float x, float y, int hp);

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

    void addClue();
    int getClueCount() const;

    void inspect(Interactable* obj);
    void getHaunted(Ghost* g);

    void printStatus() const;
};

#endif
```

Explicație: Clasa `Player`

Reprezintă investigatorul paranormal, controlat de utilizator. Pe lângă poziție și nume (moștenite din `Entity`), jucătorul are **viață** și **număr de indicii colectate**.

Clasa implementează comportamente precum: a primi damage, a se vindeca, a interacționa cu obiecte și a fi afectat de fantome. Este punctul central al logicii jocului.

Fișierul `Player.cpp`

```cpp
#include "Player.h"

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

void Player::receiveDamage(int dmg) {
    // TODO: scadeti HP si afisati mesaj
}

void Player::heal(int amount) {
    // TODO: cresteti HP si afisati mesaj
}

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

void Player::addClue() {
    // TODO: cresteti cluesFound
}

int Player::getClueCount() const {
    // TODO: returnati cluesFound
}

void Player::inspect(Interactable* obj) {
    // TODO: apelati obj->interact(this)
}

void Player::getHaunted(Ghost* g) {
    // TODO: apelati g->haunt(this)
}

void Player::printStatus() const {
    // TODO: afisati HP si numarul de indicii gasite
}
```

Fișierul `Ghost.h`

```cpp
#ifndef GHOST_H
#define GHOST_H

#include "Entity.h"
#include "IHaunting.h"
#include "Player.h"

class Player; // forward declaration

class Ghost : public Entity, public IHaunting {
protected:
    int baseDamage;

public:
    Ghost(const char* n, float x, float y, int dmg);

    virtual void haunt(Player* p) = 0; // metoda virtuala pura
};

#endif
```

Explicație: Clasa abstractă `Ghost`

Această clasă reprezintă fundamentul pentru toate tipurile de fantome. Ea moștenește `Entity` și implementează interfața `IHaunting`. Conține **damage-ul de bază** și definește un comportament comun pentru toți inamicii din casă.

Fiind abstractă, metodele specifice fiecărui tip de fantomă sunt implementate în clasele derivate.

Fișierul `Ghost.cpp`

```cpp
#include "Ghost.h"

Ghost::Ghost(const char* n, float x, float y, int dmg) {
    // TODO: finalizati constructorul
}
```

Fișierul `Spirit.h`

```h
#ifndef SPIRIT_H
#define SPIRIT_H

#include "Ghost.h"

class Spirit : public Ghost {
public:
    Spirit(const char* n, float x, float y);

    void haunt(Player* p) override;

};

#endif
```

Explicație: Clasa `Spirit`

Spiritul este o fantomă de intensitate redusă. Atacurile sale sunt mai slabe, iar comportamentul său reprezintă cel mai simplu tip de entitate ostilă din joc. Ea moștenește `Ghost` și implementează metoda `haunt()` în mod specific.

Fișierul `Spirit.cpp`

```cpp
#include "Spirit.h"

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

void Spirit::haunt(Player* p) {
    // TODO:
    // daca player are clues -> dmg = baseDamage / 2
    // altfel dmg = baseDamage
    // afisati mesajul de "cold" sau "frig" si aplicati damage jucatorului
}
```

Fișierul `Poltergeist.h`

```h
#ifndef POLTERGEIST_H
#define POLTERGEIST_H

#include "Ghost.h"
#include "Player.h"

class Poltergeist : public Ghost {
public:
    Poltergeist(const char* n, float x, float y);

    void haunt(Player* p) override;

};

#endif
```

Explicație: Clasa `Poltergeist`

Poltergeistul este o fantomă agresivă și puternică, capabilă să provoace damage mare jucătorului. Comportamentul său este mai violent decât cel al Spiritului, reprezentând un pericol semnificativ pentru supraviețuire.

Fișierul `Poltergeist.cpp`

```cpp
#include "Poltergeist.h"

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

void Poltergeist::haunt(Player* p) {
    // TODO:
    // afisati introducerea "objects fly..." sau echivalent
    // dmg = baseDamage
    // daca p->getClueCount() == 0 -> dmg *= 2
    // aplicati damage
}
```

Fișierul `Clue.h`

```h
#ifndef CLUE_H
#define CLUE_H

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

class Clue : public Interactable {
private:
    bool collected;

public:
    Clue(const char* n, float x, float y);

    void interact(Player* p) override;
};

#endif
```

Explicație: Clasa `Clue`

Reprezintă un obiect interactiv care poate fi colectat de jucător. Indiciile sunt esențiale pentru progres și pentru rezolvarea misterului. Interacțiunea crește numărul de indicii găsite.

Fișierul `Clue.cpp`

```cpp
#include "Clue.h"

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

void Clue::interact(Player* p) {
    // TODO: verificati daca collected este true
    // TODO: daca e false:
    //       - afisati mesaj
    //       - apelati p->addClue()
    //       - setati collected = true
}
```

Fișierul `LockedDoor.h`

```h
#ifndef LOCKEDDOOR_H
#define LOCKEDDOOR_H

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

class LockedDoor : public Interactable {
public:
    LockedDoor(const char* n, float x, float y);

    void interact(Player* p) override;
};

#endif
```

Explicație: Clasa `LockedDoor`

Modelează o ușă încuiată din casă. Interacțiunea cu ea permite jucătorului accesul către noi zone ale nivelului. Este derivată din `Interactable` și implementează propriul comportament la deschidere.

Fișierul `LockedDoor.cpp`

```cpp
#include "LockedDoor.h"

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

void LockedDoor::interact(Player* p) {
    // TODO:
    // daca p->getClueCount() >= X
    //      -> afisati "unlocked"
    // altfel:
    //      -> afisati mesaj cu spaima
    //      -> aplicati damage de Y HP
}
```

Fișierul `Game.h`

```h
#ifndef GAME_H
#define GAME_H

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

class Game {
public:
    void Run(Player* p, Interactable** objs, int objCount, Ghost** ghosts, int
    ↪  gCount);
};

#endif
```

Explicație: Clasa `Game`

Coordonează întregul flux al aventurii. Metoda `Run()` stabilește secvența de interacțiuni: obiecte, indicii, uși și apariții ale fantomelor.

Clasa funcționează ca un mini „engine”, folosind polimorfismul pentru a trata uniform toate tipurile de obiecte și entități.

Fișierul `Game.cpp`

```cpp
#include "Game.h"

void Game::Run(Player* p, Interactable** objs, int objCount, Ghost** ghosts,
↪  int gCount) {

    std::cout << "\n===== MYSTERY HOUSE INVESTIGATION =====\n";
    p->printStatus();

    int ghostIndex = 0;

    for (int i = 0; i < objCount; i++) {
        std::cout << "\n[STEP " << i+1 << "] Approaching: " <<
        ↪  objs[i]->getName() << "\n";
        p->inspect(objs[i]);

        if (p->getHealth() <= 0) {
            std::cout << "\n>>> You succumbed to terror...\n";

            return;
        }

        if (ghostIndex < gCount && (i % 2 == 1)) {
            std::cout << "\n!!! GHOST ENCOUNTER !!!\n";
            ghosts[ghostIndex]->haunt(p);

            if (p->getHealth() <= 0) {
                std::cout << "\n>>> The ghost claims your soul...\n";

                return;
            }

            ghostIndex++;

        }

        p->printStatus();
    }

    std::cout << "\n===== INVESTIGATION COMPLETE =====\n";
    std::cout << p->getName() << " escaped with " << p->getHealth() << "
    ↪  HP.\n\n";
}
```

Fișierul `main.cpp`

```cpp
#include "Player.h"
#include "Clue.h"
#include "LockedDoor.h"
#include "Spirit.h"
#include "Poltergeist.h"
#include "Game.h"

int main() {

    // TODO 1: Creati un Player* cu nume, pozitie si HP initial

    // TODO 2: Creati un vector de Interactable** cu 3 obiecte:
    // Exemplu:
    //  - un Clue
    //  - un LockedDoor
    //  - un Clue

    // TODO 3: Creati un vector de Ghost** cu 2 fantome:
    // Exemplu:
    //  - un Spirit
    //  - un Poltergeist

    // TODO 4: Instantiati un Game g; si apelati:
    // g.Run(p, objects, <nr_objects>, ghosts, <nr_ghosts>);

    // TODO 5: Eliberati memoria

    return 0;

}
```

În caz că nu vă descurcați să scrieți de mână:
Puteți descărca proiectul pregătit (cu toate fișierele 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 `IHaunting` si clasele care o folosesc.
4. (2p) Implementati clasele `Player`, `Ghost`, `Spirit`, `Poltergeist`.
5. (1p) Creati o simulare completa in `main.cpp` folosind 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.
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. Repository-urile sunt monitorizate automat.