Universitatea Tehnică a Moldovei

Facultatea Calculatoare Informatică și Microelectronică

Departamentul Inginerie Software și Automatică

**RAPORT**

Disciplina: Tehnici și Mecanizme de Proiectare Software

A efectuat: Gorgos Ioan st. gr. TI-202

A verificat: Gaidau Mihai

Chişinău – 2023

**Сuprins**

**1 Ce reprezintă S.O.L.I.D……………………………………..…3**

* 1. **Single Responsability Principle.....................................................3**
  2. **Open-Close Principle……………………………………………..3**
  3. **Liskov substitution principle…………………………………….4**
  4. **Interface segregation principle…………………………………..4**
  5. **Dependency inversion....................................................................5**

1. **Ce reprezintă design patterns...............................................................6**
   1. **Familiile de Design Patterns...........................................................6**
2. **Realizarea proectului……………………………………………7**
   1. **Tema aleasă.....................................................................................7**
   2. **Realizarea proectului…………………………………………...10**
3. **Concluzie…………………………………………………………………23**
4. **Ce reprezintă S.O.L.I.D ?**

S.O.L.I.D este un acronim pentru primele 5 principii ale design-ului orientat pe obiecte (OOD), dupa cum spune Robert C. Martin, cunoscut si ca Uncle Bob. Aceste principii, combinate, ajuta programatorii în dezvoltarea de software ajutandu-î să își facă aplicația ușor de mentenat și extins. De asemenea, ajută atât la scrierea de cod cât mai curat intr-un timp cât mai scurt.

S.O.L.I.D vine de la:

* S - Single responsability principle;
* O - Open-Close Principle;
* L - Liskov substitution principle;
* I - Interface segregation principle;
* D - Dependency Injection Principle.

În continuare le vom cerceta pe rând pe fiecare:

* 1. **Single Responsability Principle**

• Orice modul sau clasă trebuie să încapsuleze o singură funcționalitate;

• O clasă sau funcție, nu ar trebui să rezolve sau sa trateze mai mult decât un singur scop deoarece aceasta ar introduce cuplarea între cele două funcționalități.

* 1. **Open-Close Principle**
* Acest principiu exprimă faptul că o clasă sau o funcție, trebuie să fie deschisă pentru extensie dar închisă pentru modificare;
* De exemplu, dacă avem o bibliotecă ce conține un set de clase ce trebuie extinse ar fi mai corect dacă am avea inițial un set de clase abstracte. Astfel în loc să modificăm clasele concrete din cadrul bibliotecii, putem extinde clasele abstracte;
* Conform prof. Bertrand Meyer acest principiu se rezumă la 2 reguli:
  + Un modul se declara open dacă acesta se poate extinde. De exemplu, se pot adăuga membrii noi unei clase, prin moștenire;
  + Un modul se declară close dacă poate fi utilizat de alte module. Aceasta înseamnă că a fost clar definit, iar informația esențială este declarată private.
  1. **Liskov substitution principle**
* Acest principiu spune că, într-un program dacă S este subtip al lui T, atunci obiectele de tip T pot fi înlocuite cu obiecte de tip S, fără modificarea funcționalităților esențiale.
* Acest principiu impune unele condiții în scrierea metodelor:
  + Contravarianța argumentelor metodelor unui subtip. Aceasta înseamnă impunerea unei relații inverse de ordine între părinte și copil;
  + Covarianța tipurilor pe care le returnează un subtip;
  + Subtipurile nu aruncă excepții. Excepția de la regulă este atunci când acele excepții sunt subtipuri ale celor aruncate de metodele tipului părinte.
* Altfel spus noile clase extinse din părinte trebuie să fie capabile înlocuiască clasa de bază în toate funcțiile sale fără a fi nevoie să aducem modificări nicăieri.
  1. **Interface segregation principle**
* Acest principiu expune ideea că niciun client nu are voie să fie forțat să depindă de metodele pe care nu le folosește;
* Aceasta se realizează prin ”spargerea” interfețelor în module mai mici pe care clasele au opțiunea de a le implementa sau nu;
* Acest mod va ajuta și la ideea de decuplarea funcționalităților atunci când se dorește re-factorizarea produsului.

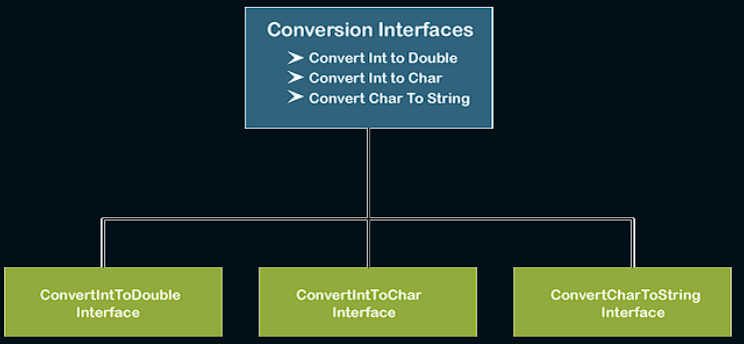
****

Figure 1.Crearea a trei interfețe în loc de una

* 1. Dependency inversion

Acest principiu se referă la o anumită formă de a decupla module software. Principiul spune că:

* Modulele de nivel înalt nu trebuie să depindă de modulele de nivel jos;
* Ambele module trebuie să depindă de abstractizări;
* Abstractizările nu trebuie să depindă de detalii ci detaliile trebuie să depindă de abstract.

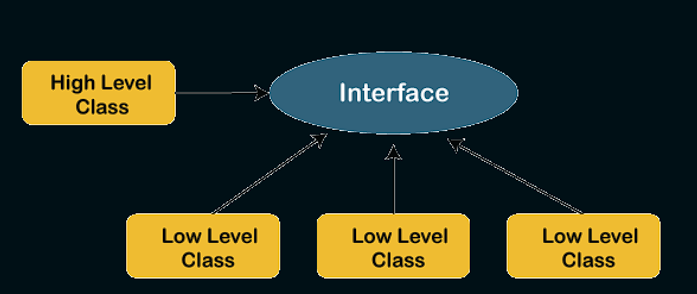


Figure 2. vizualizarea principiului Dependency Inversion

1. **Ce reprezintă Design Pattern?**

sunt soluții tipice la problemele frecvente în proiectarea software. Sunt ca niște planuri prefabricate pe care le puteți personaliza pentru a rezolva o problemă recurentă de design în codul dvs.

Nu puteți doar să găsiți un model și să-l copiați în programul dvs., așa cum puteți face cu funcții sau biblioteci disponibile. Modelul nu este o bucată specifică de cod, ci un concept general pentru rezolvarea unei anumite probleme. Puteți urmări detaliile modelului și puteți implementa o soluție care se potrivește realităților propriului program.

Modelele sunt adesea confundate cu algoritmii, deoarece ambele concepte descriu soluții tipice la unele probleme cunoscute. În timp ce un algoritm definește întotdeauna un set clar de acțiuni care pot atinge un anumit obiectiv, un model este o descriere la nivel mai înalt a unei soluții. Codul aceluiași model aplicat la două programe diferite poate fi diferit.

O analogie cu un algoritm este o rețetă de gătit: ambele au pași clari pentru a atinge un obiectiv. Pe de altă parte, un model seamănă mai mult cu un plan: puteți vedea care sunt rezultatul și caracteristicile sale, dar ordinea exactă de implementare depinde de dvs.

* 1. **Familiile de Design Patterns**
* Creational - Modelele de design creațional oferă diverse mecanisme de creare a obiectelor, care cresc flexibilitatea și reutilizarea codului existent.

Exemple de Design Pattern Creaționale:

* + Factory Method;
  + Abstract Factory;
  + Builder;
  + Singleton;
  + Prototype.
* Structural - Modelele de proiectare structurală explică modul de asamblare a obiectelor și claselor în structuri mai mari, păstrând în același timp aceste structuri flexibile și eficiente.

Exemple de Design Pattern Structural:

* Adapter;
* Bridge;
* Composite;
* Decorator;
* Facade;
* Fleyweight;
* Proxy;
* Behavioral – această familie de design pattern-uri sunt preocupate de algoritmi și de atribuirea responsabilităților între obiecte.

**3 Realizarea proectului**

**3.1 Tema aleasă pentru proectul de an**

În calitate de team a fost ales realizarea unui mini joc, cu respectarea principiilor solid și implementarea design pattern-urilor.

Prezentarea rezultatului:

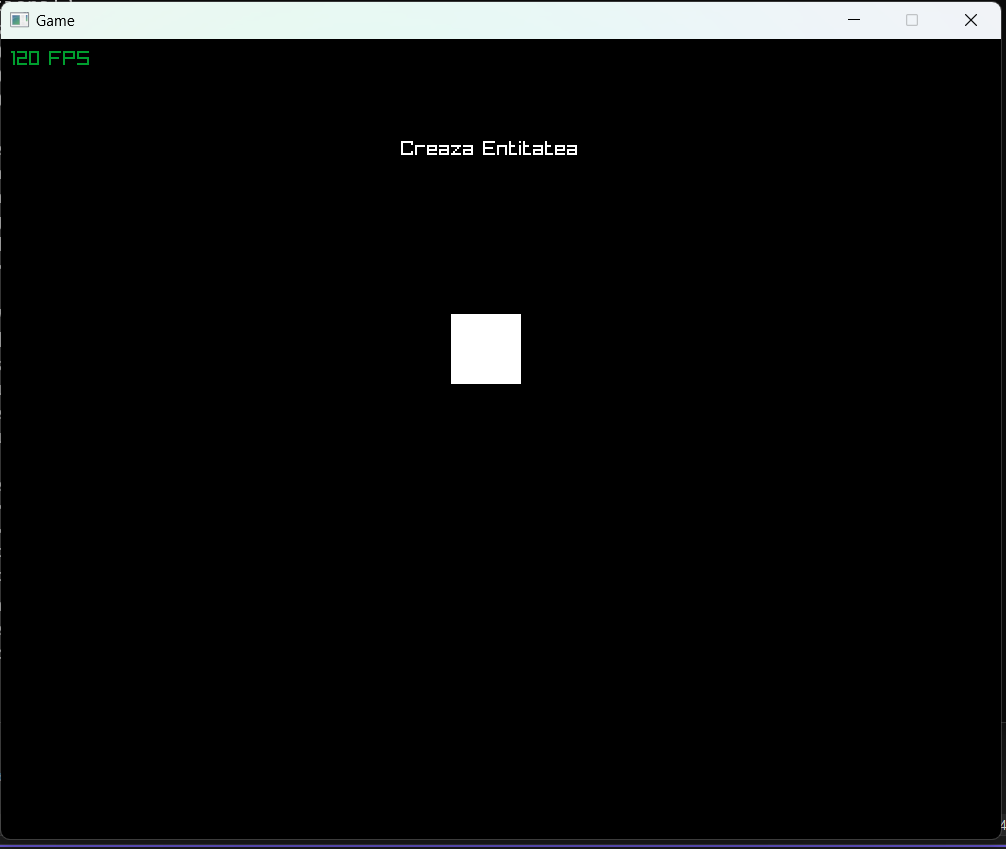


Figure 3.Vizualizarea primei etape

În figura 3 este reprezentată prima etapă al jocului respective, în care este necesar de setat dimensiunile bolcului nostru, în dependență de mărime variază viteza și voluml vieții.

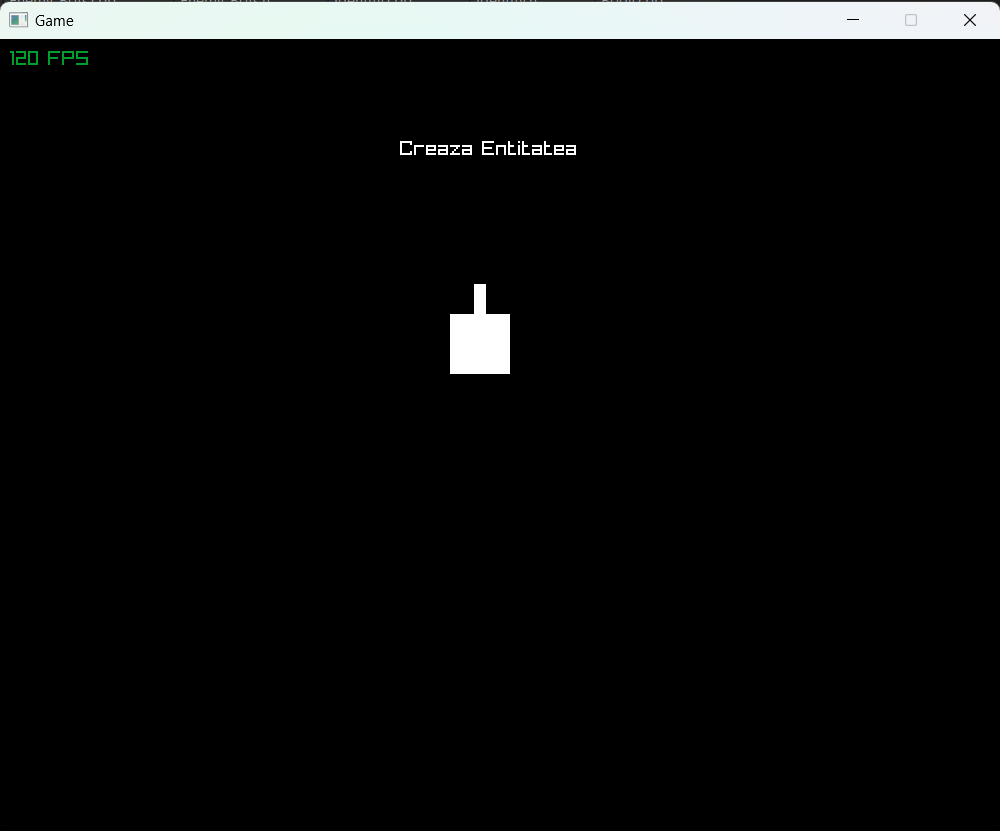


Figure 4.Alegerea primului model de arma

Figura 5 este specificat primul model de armă, care are o putere mai mare comparative cu a doua, care va fi reprezentată în următoarea figură.

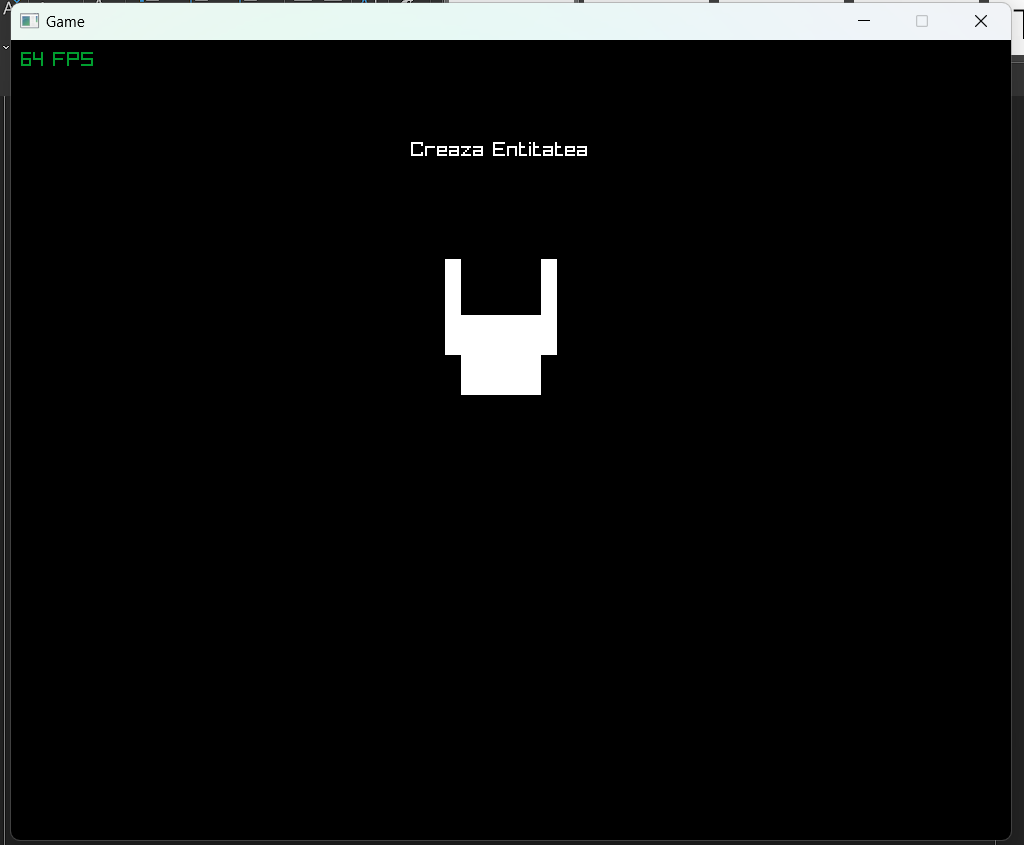


Figure 5.Alegerea modelului doi de armă

Figura 4 este specificat al doilea model de armă, care are o puteri mai mică comparativ cu prima, dar în schimb permite tragerea concomitent a două gloanțe.

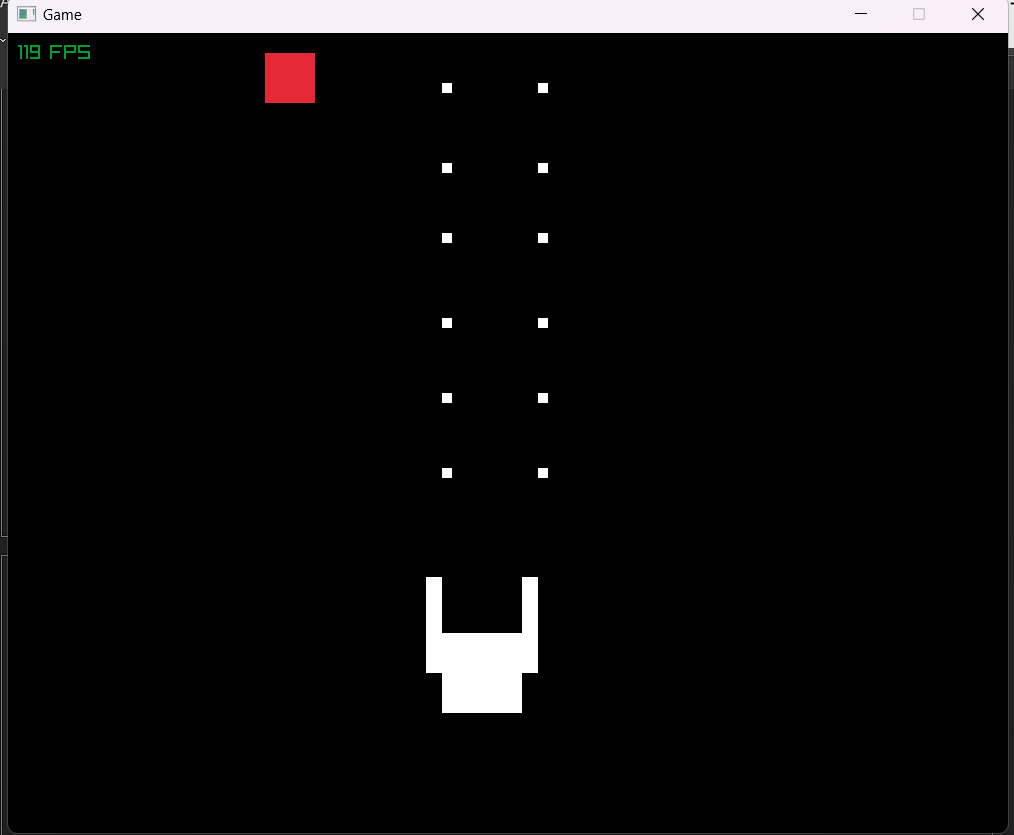


Figure 6.Reprezentarea procesului de joc

În figura 6 este prezentat însuși procesul de joc, unde blocul roșu este oponentul(bot) care urmează a fi distrus, iar blocul alb reprezintă blocul nostru construit la etapele precedente. Cu ajutorul tastelor stânga(left) și dreapta(right) se poate de mutat poziția blocului pe axa x, iar cu tasta spațiu(space) se pate de lansat glaonțe.

Pentru a înțelege și a descrie design paternurile, vom reprezenta structura proectului.

**3.2 Realizarea proectului**

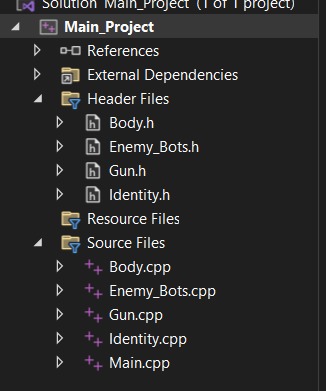


Figure 7.Structura proectului

În figura 7 este reprezentată structura proectului, este specificat toatea head-urile, descrierea acestora și fișierul main. Pentru realizarea acestui joc a fost folosite următoarele paternuri: Build, factory method, bridge, adapter. În continuare vom descrie fiecare patern în parten mai detaliat. Dar mai întâi vom afisa funcția main în care tote componentele noastre sunt strânse.



Figure 8.Funcția main

Această este funcția main în care se crează fereastra noastră de dialog, în interiorul căreia se realizează toată acțiunile. Se poate observa o buclă while, în corpul căreia are loc toate acțiunele care vor urma, de la alegerea armei, până la însuși procesul de joc. Se poate observa un if care are ca parametru variabila bool menu, aceasta specifică la ce stadie suntem acum, adică cream blocul nostru și arma sau deja suntem la etapa de joc. Se poate observa un alt if, care verifică dacă timing este mai mic sau egal cu 0, aceasta este folosită pentru a identifica dacă a trecut 3 secunde de la momentul creării blocului până la moemntul de start al jocului, în cazul în care nu a trecut 3 secunde, se afișează mesajul ,,jocul incepe in … secunde,,.



Figure 9. Funcția begim

În figura 9 este prezentată funcția begin, în momentul de start al jocului se apelează această funcție, care ne permite să creăm identiatea noastră. În primele linie de cod este specificat afisarea mesajului ,,Creaza Entiatea,, apoi urmează un if și verirică dacă x este egal cu zero, aceest lucru permite determinarea etapei de creare a entității, dacă acest este 0 atungi set urmează setarea dimensiunii blocului nostru cu ajutorul tastelor Left(stânga) și right(Dreapta), apoi pentru a trece mai departe se tastează enter.

În linia de cod 59 este specificată verificarea de tastare a enter-ului, dacă aceasta a fost tastată se apelează metoda ProductBody, al obiectului ident, care permite crearea blocului în Identity(ateste nuanțe urmează a fi descrise), apoi x se incrementează cu o unitate. Dacă x este egal cu 1, atunci urmează setarea armei dorite, apoi iarăși urmează tasta enter care incrementază x cu unu. În momentul în care x este egal cu 2 variabila menu se transformă în false și în funcția main are loc startul jocului.

În continuare vom cerceta implementarea design pattern-ului Factory, care a fost folosit pentru a crea arma dorită.

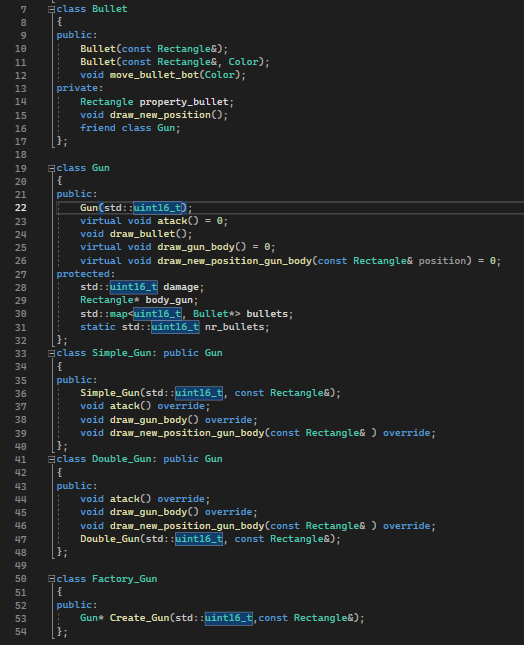


Figure 10.Factory method pentru clasa Gun

În figura 10 sunt prezentate clasele Simple\_gun și Double\_Gun care realizează interfața Gun, tot aici se poate observa clasa Factory\_Gun care are o singură metodă Create\_Gun și care întoarce o instanță nouă de arma aleasă. Clasa Bullet este utilizată pentru a crea identitatea glonțului. Pentru a înțelege mai exact funcționalitatea. Mai jos este prezentată diagrama uml

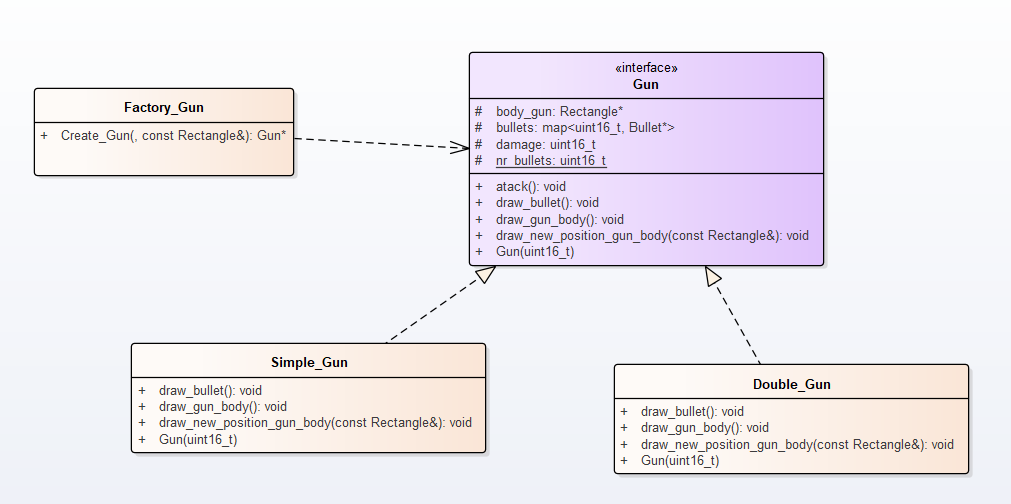


Figure 11.Diagrama UML pentru Factory Method

În figura 11 este prezentată diagram UML pentru Factory Method pentru Gun. Aici se poate observa mai detaliat căci clasa Gun reprezintă o interfață care are un set de metode și atriubte. Această interfață este realizată de Simple\_Gun și Double\_Gunm, aceasta se poate observa prin relația de implementare între acestea . Tot în această se poate observa clasa Factory\_Gun, care are o relație de dependeță cu interfața Gun, se mai poate observa că aceasta are o methodă create\_gun care preia ca parametru id-ul tipului de armă care dorim să o creăm și mărimele blocului nostrum creat.



Figure 12.Realizarea claselor pentru Gun

În figura 12 este prezentat o parte din implementarea claselor de la figura 10, se poate observa descrierea constructorilor bullet, simple\_gun, double\_gun, gun. Tot aici se poate observa descrierea funcționalității fiecărei funcții în parte. Pentru a înțelege mai bine cum funcționează acestea, codul este încărcat pe github și se poate de cercetat mai detaliat.

În continuare se va cerceta paternul Bridge, care ne permite să decuplăm o clasă foarte mare în componente mai mici pentru a avea o modularitatea a claselor și o flexibilitate mai mare.

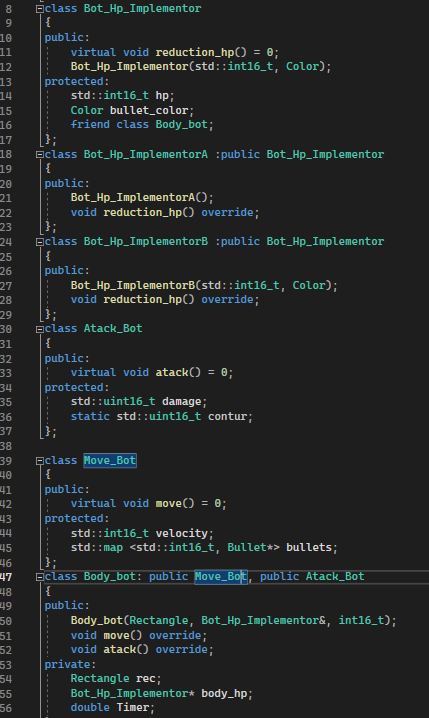


Figure 13.Implementarea design patternului Bridge

În figura 13 este prezentat identitatea pentru boți, care pentru realizarea funcționalității acesteia a fost implementa design pattern-ului Bridge. Se poate observa căci există intefața Bot\_Hp\_Implementor care este utilizată pentru a indica nivelul de viață al botului, această intefață este realizată de clasele Bot\_Hp\_ImplementorA și Bot\_Hp\_ImplementorB, toti aici se poate de menționat căci în clasa Body\_bot, constructorul preia prin referință obiectul clasei Bot\_Hp\_Impplemetor, se poate de menționat faptul căci aici se implementează principiul Dependece Inversion din SOLID.

Pentru a întelege mai exact funcționalitatea, mai jos este prezentată diagrama UML pentru Bridge.

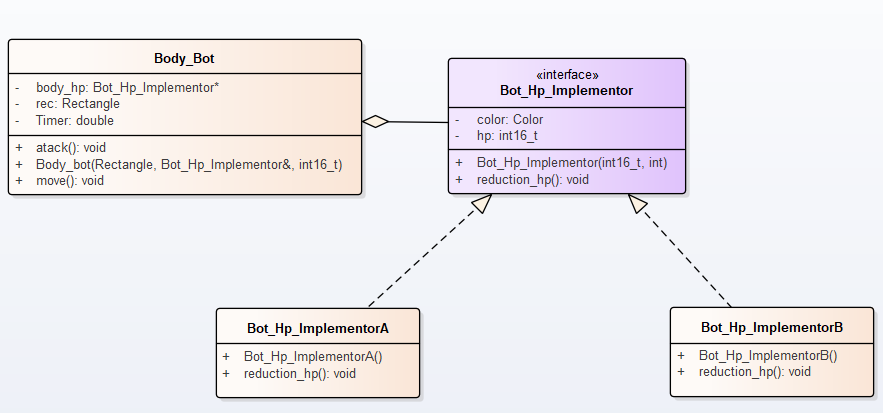


Figure 14.Diagrama UML pentru Bridge

În figura 14 se poate observa implementarea design pattern-ului Bgride. Clasa Bot\_Hp\_Implementor rerepzintă o intefață care este realizată de clasele Bot\_Hp\_ImplementorA și Bot\_Hp\_ImplementorB, tot aici se vede relația de agregare între clasele Body\_Bot și Bot\_Hp\_Implementor și care dă de înțeles căci clasa Body\_Bot folosește această clasa pentru necesitățile sale interne.

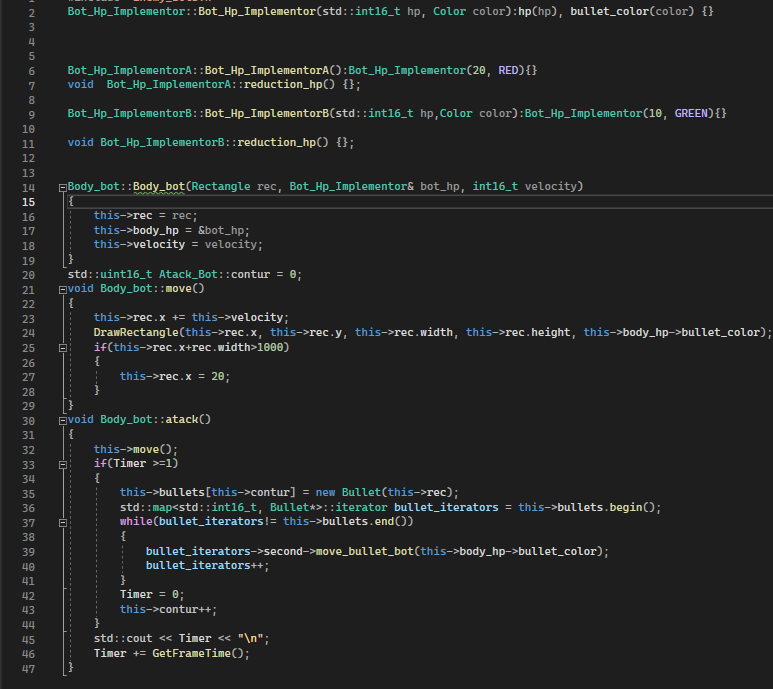


Figure 15.Realizarea design Patternului Bridge

În figura 15 este descris funcționalul, atât pentru boți, cât și deja realizarea însuși a desig patternului Bridge. Tot aici se observă descrierea constructorilor pentru clasele Bot\_Hp\_Implementor, Bot\_Hp\_ImplementorA, Bot\_Hp\_ImplementorB, Body\_Bot, este descrisă funcția pentru a ataca(atack) a botului, funcția pentru deplasare al botului, funcția de reducere al veții botului.

Următorul Design Pattern care îl vom studia este Build, folosit pentru a construi enitatea noastră,care va fi folosită pentru ca avatarul nostrum.

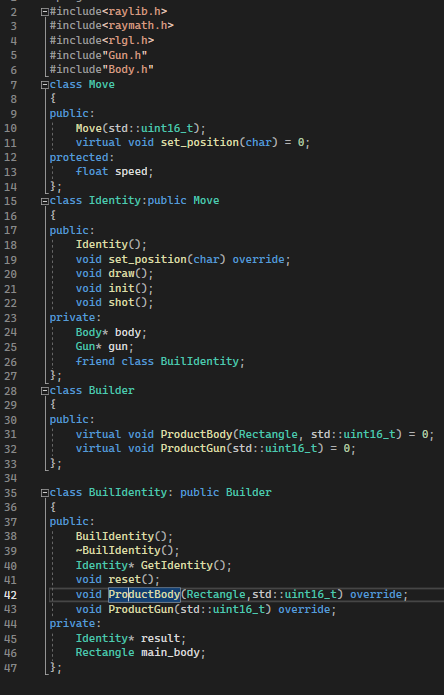


Figure 16.Realizarea design pattern-ului Build

În figura 16 sunt prezentate clasele folosite pentru realizarea avatarului utilizatorului și prin intermediul acestora este implementat design pattern-ul Build. Clasa Identity reprezintă rezultatul care este întors de design pattern-ul buil, clasa abstractă Builder reprezintă o interfață care este folosită pentru a indica ce tip de indeintate dorim să crem(ce componente pot exista), clasas BuildIdentity realizează clasa abstractă și ne permite să cream obiectul clasei identity cu parametrii setați.

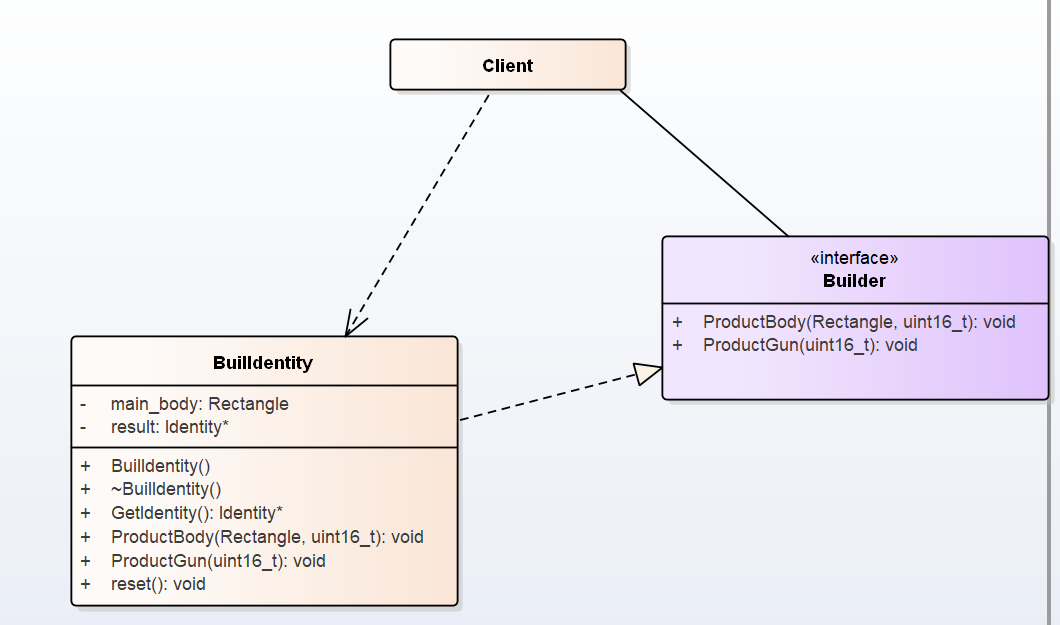


Figure 17.Diagrama UML pentru Build

În figura 17 este prezentată diagrama UML pentru build, spre deosebire de forma standartă a desig pattern-ului, aici lipsește clasa Director, care ste adițională. Aici se obsevă că există o clasă abstractă Builder și aceasta este realizată de clasa BuildIndentity, clasa BuildIndentity ne permite să creăm identatea noastră cu componentele dorite și să întoarcem obiectul indentității respective. Client reprezintă componenta care cere de la BuildIndentity să creeze o componentă nou.

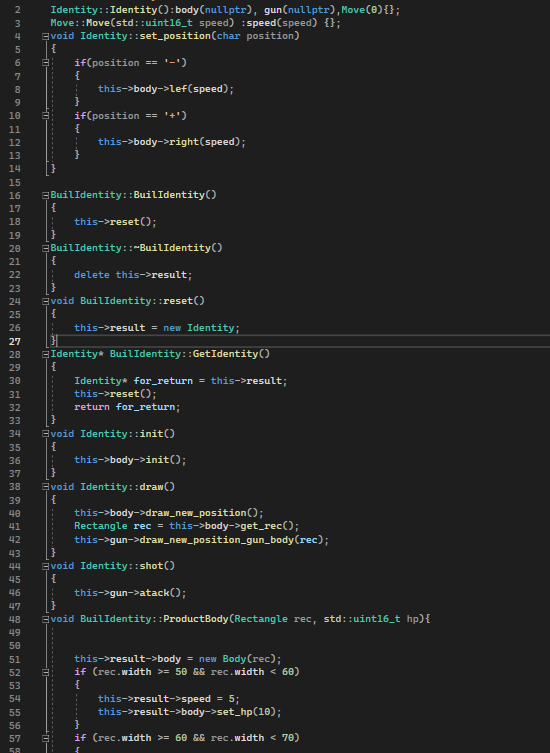


Figure 18.Reprezentarea codului pentru Build

În figura 18 este prezentat codul pentru design pattern Build, aici se poate observa realizarea constructorilor pentru clasa Move, BuilIdentity, realizarea funcțiilor, realizarea distructorilor. Pentru a înțelege mai bine codul, acesta va fi încărcat pe github.

În continuare va fi descris design paternul Adapter. Adaptorul este un model de design structural care permite obiectelor cu interfețe incompatibile să colaboreze.

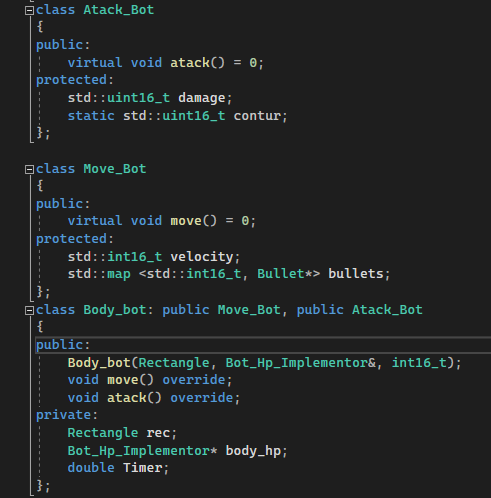


Figure 19.Interfetele pentru Adapter



Figure 20.Crearea obiectului Body\_bot cu interfata atack

În figura 19 este prezentat clasa Body\_bot care implementează interfața atack\_bot și Move\_Bot, iar în figura 20 este prezentat deja implementarea design pattern-ului Adapter prin crearea unei instanțe de tip Body\_bot și atribuită clasei de tip Atack\_Bot. La apelarea obiectului bot, va fi posibilă doar adresarea la metoda atack.

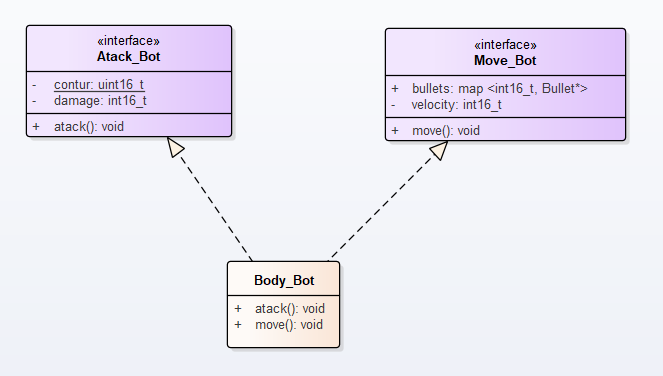


Figure 21.Diagrama UML pentru Adapter

În figura 21 este prezentată diagrama uml pentru design patternul Adapter, aici sunt două interfețe una Atack\_Bot și alta Move\_Bot, care sunt realizate în clasa Body\_Bot. clasa Body\_Bot realiezază override pentru funcțiile atack și move, apoi cum a fost specificat în figura 19 se crează obiectul Body\_Bot, de tip Atack\_Bot și care are doar o singură metodă atack.

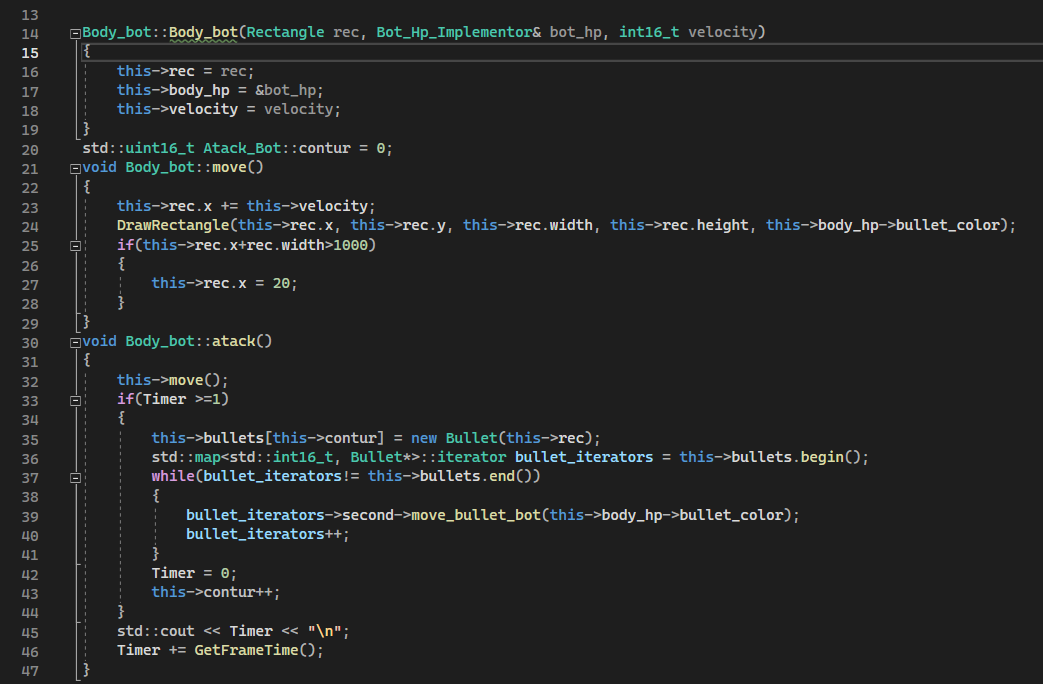


Figure 22.Realizarea funcțiilor pentru Adapter

În figura 22 sunt prezentate descrierele pentru funcțiile override al obiectului Body\_Bot moștenite de la Move\_Bot și Atack\_bot, se poate observa căci în metoda atack se apelează move, aceasta este realizat pentru a mișca bot-ul nostru doar prin apelarea din exterior al metodei atack.

**Concluzie**

În urma realizării acestui priect am făcut cunoșință așa numitele principii SOLID, am înteles care sunt avantajele acestor și de ce este o practică bună respectarea lor. Cercetarea principiilor a permis, atât scriere unui cod mai curat, mai clar și mai extensibil, cât și înțelegerea la un nivel mai superior al codului scris de alți develor**.** Tot în procesul de realizare a proectul am făcut cunoștință cu diferite desig pattern-uri, cum ar fi Build, Adapter, factory method, am înțeles avantajele acestora, unde sunt folosite, când este mai bine de le folosit, am aflat cum pot interacționa între ele. Pe lângă faptul căci am interacționat cu SOLID și desig patterns, mi-a făcut plăcere să interacționez cu libraria grafică raylib, care este destinată creării diferitor elemente grafice în C++ și cu ajutorul căreia se poate de realizat diferite jocuri.