Cuprins

[Lista abrevierilor 2](#_Toc75450674)

[Abstract 2](#_Toc75450675)

[Introducere 3](#_Toc75450676)

[1 Istoria scurtă a dezvoltării 6](#_Toc75450677)

[1.1 Încerări inițiale 6](#_Toc75450678)

[1.2 Corona și Lua: etapa 2 7](#_Toc75450679)

[1.3 Rescrierea în C 8](#_Toc75450680)

[1.4 Unity și Godot 9](#_Toc75450681)

[1.5 Generarea codului 10](#_Toc75450682)

[2 Prezentarea generală a sistemei 11](#_Toc75450683)

[2.1 Prezentarea generală a mecanicilor jocului 11](#_Toc75450684)

[2.2 Prezentarea generală a design-ului sistemului. 14](#_Toc75450685)

[2.3 O direcție greșită? 17](#_Toc75450686)

[2.4 ECS (Entitate-Component-Sistem) 22](#_Toc75450687)

[3 Subiectele tehnice 25](#_Toc75450688)

[3.1 Grila 25](#_Toc75450689)

[3.2 Chain-urile 32](#_Toc75450690)

[3.3 Entități și Componente 34](#_Toc75450691)

[3.4 Acționare și bucla de joc 42](#_Toc75450692)

[3.5 Registru 49](#_Toc75450693)

[3.6 Generarea codului 56](#_Toc75450694)

[Concluziile 66](#_Toc75450695)

[References 66](#_Toc75450696)

# Lista abrevierilor

|  |  |
| --- | --- |
| Acronim | Semnificație |
| ECS | Entity-Component-System |
| API | Application Programming Interface |
| Necrodancer | Crypt of the Necrodancer |
| MVC | Model-View-Controller |
| POO | Programarea orientată pe obiecte |
| Roslyn | .NET Compiler Platform (Roslyn este un pseudonim) |
| T4 | Text Template Transformation Toolkit |
| ID | Identificator |
| IA | Inteligența Artificială |
| PVP | Player-vs-Player |
| MOBA | Mutiplayer Online Battle Arena |
| GC | Garbage Collector |

# Abstract

Cu colegul meu Țurcanu Critian am decis să creăm un joc Roguelike, numit **Hopper**, băzat pe mecanici din *Crypt of the Necrodancer*. În partea intorductivă explic de ce am inițiat acest proiect. În prima parte teoretică ce urmează, descriu baza experimentală a lucrării, adică prin ce stadii de dezvoltarea a trecut jocul, ce idei am explorat, și ce instrumente am încercat. În a doua parte teoretică aduc prezentarea generală a mecanicilor jocului, propun unele idei ce se referă la structura jocului și la soluționarea unelor probleme globale de arhitectura jocului. Ultima parte conține informații tehnice: detalii la design-ul sistemei cu exemple abundente din codul sursă, explicații despre funcționarea internă a jocului, prezint cum am evitat boilerplate-ul și duplicarea codului prin generarea codului cu *Roslyn* și *T4*.

# Introducere

Ideea jocului este inspirată de jocul **Crypt of the Necrodancer** (mai departe voi referi la jocul acesta ca simplu Necrodancer), care este jocul meu preferat. Denumirea **Hopper** provine din faptul că animația mișcării a caracterelor este un salt (*hop*, în limba engleză)

**Motivarea proiectului**

Am hotărât să încep acest proiect, deoarece există unele probleme cu Necrodancer, jocul după care mă orientez. Aceste probleme sunt următoarele:

1. Modarea este în esență imposibilă. Sunt posibile numai modurile care schimb aspectul vizual al jocului. Adăugarea mecanicilor sau a tipurilor de inamici nu este posibilă.
2. Necrodancer nu este disponibil pe Android. Am dorit să pot să rulez acest joc pe dispozitive mobile.
3. Codul nu este public.

**Scopurile pentru proiect**

Jocul s-a planificat să fie open-source și băzat pe aceleași mecanici. Este desemnat satisfacerii personale, și comunității online care l-ar continua mai departe, cu sau fără ajutorul meu.

Scopurile provin direct din problemele menționate mai sus.

1. Să dezvolt o joc cu mecanici asemănători cu cele din Necrodancer.
2. Să dezvolt un sistem informatic scalabil cu mai multe proprietăți și un API ușor de utilizat pentru a ușura dezvoltarea modurilor.
3. Să public codul pe github cu o licența permisivă și să invit dezvoltători de moduri când API-ul devine destul de matur.
4. Să pot rula jocul pe dispozitive mobile cu sistemul de operare Android.

Realizarea acestor lucruri ar fi valorată doar de o nișă foarte specifică de jucători și modatori, deci proiectul să nu fie considerat în special inovativ sau actual. Scopul proiectului nu este să primim vreun profit material, ci să experimentăm și să învățăm.

**Benificiile adăugătoare**

Aparte de satisfacere de programator personală, beneficiile acestui proiect includ faptul că am primi experiență în:

1. dezvoltarea jocurilor.
2. domeniul menținerii proiectelor complexe.
3. în colaborarea și comunicarea cu comunitatea, artiștii și alți dezvoltatori.

În plus, proiectul este destul de mare și complex și ar fi valorat înalt printre alte proiecte personale de către angajatori.

**Lucru în echipă**

Eu sunt programator, nu sunt artist sau designer.

Îmi place a programa sisteme complexe și instrumente, însă eu nu am capacitatea de a proiecta jocurile singur, nici nu vreau să fac acest lucru. Scopurile *mele personale* pentru acest proiect erau să construiez o bază, un *Core* (nucleu) al jocului, bazându-se pe care alții ar putea adăuga mai multe idei. Nu am ca scop să creez un joc *complet* în a acest proiect, nici să lucrez asupra graficii (desenarea sprite-urilor, crearea animațiilor, iluminației, interfeței de utilizator, etc.). Aș dori să accentuez faptul că partea mea în acest proiect este să construiez acea bază, acea interfață de interacțiune cu lumea logică și cu caracterele, instrumentele pentru crearea obiectelor și inamicilor noi etc.

Pe de altă parte, al doilea programator, Cristian Țurcanu, cu care lucram în echipă, a elaborat sistemul de interacțiune cu jucătorul. El a legat sistemul meu cu motorul de joc pentru a prezenta graficii și a înregistra input-ul utilizatorului.

**Suportul metodologic și teoretico-științific**

Nu menționez nici o lucrare științifică în lucrarea mea. Nu este o eroare în bibliografie, ci incompetența mea în timpul realizării lucrării și codului de a căuta informații utile referitor la programarea jocurilor video. Prin urmare, sursele de metode sunt fie experiența în programare adunată pe parcursul studiilor și implementării proiectelor personale, fie ideile generale de proiectare a arhitecturii sistemelor informatice, absorbate pe parcursul studiilor și rezolvării unor tipuri de probleme, fie pur și simplu experimente ceva reușite. În orice caz, sursele ideilor sunt fie inexistente, fie pierdute, iar uneori și ale noastre originale.

**O descriere foarte scurtă a jocului**

Jocul care am vizat să-l dezvolt împrumută mecanicile sale de bază de la **Crypt of the Necrodancer**.

Necrodancer este un joc **Dungeon Crawler, Roguelike**. Explorați un donjon generat aleatoriu, combatând inamicii și bosurile în proces. Este un joc băzat pe turnuri, adică Dumneavoastră și inamicii Dumneavoastră pot să facă o acțiune (mișcare, atacă, aruncarea unei vraji magice, deschiderea unui cufăr, etc.) doar o singură dată în fiecare tur.

Îmi place conceptul de Roguelike în general — faptul că deveniți mai puternici când progresați mai adânc în donjon. Îmi place și să joc jocuri video de acest fel. Printre ele: **The Binding of Isaac**, **Into the Breach**, **The Darkest Dungeon**, **Slay the Spire** și **One Step From Eden** sunt cele care mi-au plăcut în special.

Faptul care îl distinge pe Necrodancer este o întorsătură inteligentă de mechanici, anume faptul că *puteți face acțiuni doar după ritmul muzicii*. Datorită acestui fapt, jocul este clasificat ca un joc **Roguelike Rhythm**.

Faptul că aveți timp limitat excepțional distinge jocul de la celelalte jocuri de genul Roguelike. Tehnic, jocul este bazăt pe tururi, însă datorită acestei mecanici este și cu mersul repede. Cu toate că aveți *ceva* timp să vă calculați următoarea acțiune, este imposibil să luați în considerare totul, cum ați putea face în șah.

Încă, este important să prevedeți consecințele acțiunilor Dumneavoastră și să planificați aproximativ ce se va întâmpla în următoarele tururi, însă, ținând cont că timpul dintre bătăile ritmului pentru a se gândi este atât de limitat, reacția tot joacă un rol mare. Acest joc învață să puteți opri la un anumit moment să luați o acțiune vrednică care mai degrabă nu va fi cea optimă. Această idee este într-un mod similară la ideea șahului cronomerat, unde timpul Dumneavoastră este o resursă de utilizat competent, deocamdată ticăitul ceasului ar putea să vă agite.  
Distanța scurtă dintre bătăile uneori se pare, de asemenea, stresantă, dar se simte bine să prinzi unele momente intense, în care reușeșți să respingi o hoardă de dușmani, de exemplu cu o vrajă magică bine aruncată sau cu o lovitură de armă abilă.

**Cuvinte cheie**

Necrodancer, game architecture, ECS, code generation, game modding, Godot, Roslyn, MVC.

# 1 Istoria scurtă a dezvoltării

Am început să lucrez asupra acestui proiect aproape 2 ani în urmă. Pe parcursul acestor 2 ani, a fost aruncat și rescris, complet sau parțial, de aproape 5 ori.

Aș zice că este greu să știi ce să faci în așa proiect chiar de la început, chiar aș zice imposibil. Cu sarcini complexe fără cerințe definite în întregime, rar faceți lucrurile bine prima dată. Codul este rescris, ideile devin mai clare, ariile noi sunt explorate și abandonate. Să scieți un joc, asemănător, nu este liniar.

Cu toate că am știut de la început conceptul general pe care am vrut să-l urmăresc, și mecanicii de bază deja clare, nu am știut cum să structurez jocul corect, în ceea ce privește codul și design-ul sistemei. Deci, trebuiam să încerc mai multe idei pentru a ajunge la acele momente mai insteresante pe care le am astăzi.

## 1.1 Încerări inițiale

Inițial, încercam să programez jocul în motorul de joc *Corona*, în limbajul de programare *Lua*. Permite exportarea pentru sisteme de operare de dispozitive mobile și pe desktop. A se vedea repertoriul pe github [după acest link](Anton, n.d.).

Însă, înțelegerea mea a structurii acestor jocuri, cum ele lucrează pe partea sistemei, era slabă atunci.

Design-ul și realizarea unui joc simplu este cu totul diferit de problema pe care am întâlnit-o eu. Dacă proiectați un joc care poate să aibă mii de efecte diferite, de mecanici și entități, posibil expandată de către moduri, nu puteți ține cont pentru fiecare interacțiune cu niște if-uri, aveți nevoie de un sistem mai abstract și complex, care permite utilizarea unui oricare fel de polimorfism. Nu am realizat acest lucru înaite de acest proiect, însă l-am realizat după această primă încercare. În părțile de mai târziu voi analiza aceasta mai detaliat.

Această încercare inițială la realizarea jocului mi-a adus înțelegerea faptului că jocurile video complexe nu sunt doar o mulțime de if-uri. Ele necesită creativitatea și competența.

Codul inițial a fost aruncat și rescris de la început în a doua versiune, încă pe Corona.

## 1.2 Corona și Lua: etapa 2

Lua este un limbaj de programare foarte simplist: nu există tipurile, modulele sau clasele. Dynamic method dispatch, încă, poate fi simulat prin metatabelele (moștenirea prototipică). Încă, nu există tablouri: și tablouri și dicționari sunt reprezentate prin așa numite tabele (perechile cheiea-valoare).

Cea mai mare problemă cu Lua este lipsa tipurilor și, ca rezultat, lipsa analizei statice. Combatați bug-urile proaste, ca o eroare runtime din cauza unei greșeli în numele variabilei, în fiecare zi. Aceste bug-uri sunt dificil de observat.

Am ajuns destul de departe cu Lua, am dezvoltat destul de multe capacități.

În acel timp, am ajuns la ideea de a utiliza **chain-urile** pentru implementarea event-urilor. În scurt, chain-urile în interpretarea mea sunt *responsibility chain-uri* care fac ceva cu context-ul transmis lor, ca stiva de funcții middleware pe backend care modific în secvența obiectele request și response. La orice etapă, propagarea context-ului poate fi oprită de către una din funcții (handler), pentru a evita execuția handler-urilor ce urmează. În plus, fiecărui handler este asociată o prioritate, după care ele rămân sortate în structura subiacentă a datelor. Chain-urile vor fi exemplificate mai bine în partea tehnică ce urmează.

Această idee a devenit esențială pentru modul în care eu am reușit să administrez mișcarea, atacarea, primirea daunei, etc.

În acest timp, am relizat că, ca sistemul să fie destul de robust, am nevoie de componente dinamice. Mai mult despre ele în secțiunile ce urmează.

Această etapă a proiectului este documentată destul de bine, o valoare esențială, dacă chiar nu aveți tipuri în limbajul Dumneavoastră. Am scris niște articole în limba engleză ce descriu unele mecanici din sistem. Le puteți [găsi aici.](https://antonc9018.github.io/Dungeon-Hopper-Docs/) Unele idei documentate aici s-au tradus aproape intact în versiunea nouă a codului.

## 1.3 Rescrierea în C

În fine, m-am săturat de faptul că Lua nu are tipuri. Am decis să rescriu întregul proiect în C#. De ce C#? Nu-mi păsa ce motor de joc voi utiliza la final, deci m-am concentrat pe partea logică a proiectului, adică dezvoltarea sistemei. Am știut că există **Unity** și **Godot** care suportă C# ca limbajul său de scripting. Deci, ideea era să scriu nucleul jocului independent de grafică. Acest concept are numele **MVC (Model-View-Controller)** sau **MVVM (Model-View-ViewModel)**. Cu așa sistem, ar fi posibil să creez “scripturi de vizualizare” în orice motor de joc ce suportă C#.

Această idee nu este nimic nou, și de fapt m-am gândit la ea de la început. Însă, înainte de a începe a utiliza C#, ideea referitor la modul în care comunicare dintre ele trebuia să fie organizată era vagă în mintea mea. Înainte de tranziția această de la Lua, eu nici nu am încercat să creez un sistem destul de robust pentru a putea dirija acest proces. Am programat doar un prototip simplu pentru testare și am lăsat așa.

Deci, versiunea jocului pe C# inițială era dezvoltată doar în consolă. Atunci nici nu am știut cum să scriu teste automatizate, sau mai bine spus nu m-am apucat de aceasta atunci. Testele erau manuale, băzate pe inspecție. Am avut un script care ar afișa o mulțime de informație referitor la cum obiectele au interacționat în joc, pe când eu aș citi aceste loguri pentru a mă asigura dacă elementul nou a funcționat corect.

În această versiune inițială în C#, eu de fapt am tradus codul din Lua în C# și am îmbunătățit unele idei. Codul a devenit mai robust, dar nu destul de robust.

Am avut o mulțime de probleme de întreținere care au încetinit programarea și au făcut-o anevoioasă. Pe scurt, am folosit fabrici pentru a construi tipurile mele de entități, precum și builder-uri pentru chain-urile lor inițiale. Problema este că, dacă creați și utilizați fabricile manual, aveți acest cuplaj strâns dintre fabrica și tipul de entititate pe care îl produce. Deci, când schimbam entitatea, trebuia să mă întorc și să schimb și fabrica. Când schimba modul în care funcționează chain-urile, trebuia să mă întorc și să văd dacă builder-ul funcționează corect.

Am avut conceptele de *tinker* și *retoucher* (ambele termeni sunt inventate) care ambele au existat numai pentru a ușura procesul de adăugarea și scoaterea handler-urilor la sau de la chain-uri. Unica diferența dintre ele era că retoucher-urile se utilizau pentru *tipuri* de entități (pe factori), iar tinker-urile pe *instanțe* de entități. Au făcut același lucru, fiind diferiți doar în ce tip de container au vizat. Faptul că ele au făcut același lucru a implicat duplicarea codului și unele probleme de întreținere. Însă, nu primim nimic dacă le diferențiem, deoarece ele deja fac același lucru.

Într-un fel, am realizat aceasta, însă nu am știut cum s-o rezolv.

O altă problemă de întreținere era cod *boilerplate*.

De fapt nicidecum nu puteți evita această problemă în întregime în C# curat. Știu că există reflexia, însă ea este nesigură și predispusă la erori, în plus foarte lentă. Am utilizat reflexia pentru a evita boilerplate-ul în unele locuri, de exemplu adăugarea obiectelor de statuturi, care sunt în esența niște structuri cu numeri întrege.

Un alt instrument pe care am încercat să-l utilizez erau interfețe generice. Acestea lucrez într-o măsură, însă prea mult complic codul.

Pe toate aceste probleme le-am rezolvat în mare parte doar recent, utilizând generarea codului.

## 1.4 Unity și Godot

Peste niște luni după rescierea proiectului în C#, baza de cod a devenit destul de matură pentru a încerca să fac un *view* pe Unity sau Godot.

Inițial, am făcut un demo mic pe [Unity](https://github.com/AntonC9018/hopper-unity). Demo-ul a prezentat un view care a lucrat cu interfețe și care pe urmă a fost reutilizat pentru același [demo în Godot](https://github.com/AntonC9018/hopper-godot) (nu este veriunea curentă). Această dată am proiectat un prototip mai robust pentru view, însă oricum unele proprietăți lipseau și nu era plăcut să lucrez cu el. Deficiențiele lui au fost adresate în versiunea nouă, în mare parte dezvoltată de către colegul meu, Țurcanu Cristian, care o descrie mai detaliat în [lucrarea sa](file:///E:\Coding\C%23\Paper\citation_needed).

Mediul concret al motorului de joc, cum s-a menționat anterior, nu-mi afectează procesul de lucru într-un mod semnificativ. Din această cauză voi omite descrierea motoarelor acestea. Însă colegul meu a lucrat mai apropiat cu motorul de joc, deci vă încurajez să vedeți [lucrarea lui](file:///E:\Coding\C%23\Paper\citation_needed) pentru mai multe detalii referitor la motoare de joc. Partea mea în dezvoltarea a jocului era să derivez un API esențial, ce am făcut independent de oarecare motor de joc. Pentru mine, motorul de joc numai mi-a oferit o modalitate de a vizualiza ce face codul meu. Acest mod de a vizualiza ce face programul uneori poate ajuta în identificarea și înțelegerea unor bug-uri. Explicarea fenomenului este că oameni înțeleg input-ul vizual mai intuitiv decât log-urile în consola sau stiva de apeluri; uneori problema este mai aparentă dacă o observați în acțiune.

## 1.5 Generarea codului

Începând cu luna aprilie, am lucrat asupra generării codului pentru a elimina boilerplate-ul și pentru a face procesul de dezvoltare mai puțin strângenitor.

### 1.5.1 Motive pentru generarea codului

Generarea codului este esențială, deoarece induce experimentarea. Când eu văd un pattern care nu poate fi ușor exploatat ușor fără generarea codului, au pot să fac rapid un modul prototip pentru generator de cod care ar exploata ideea. Dacă văd că este util, îl utilizez în continuare. Dacă nu, anularea lui poate fi atinsă pur și simplu prin omiterea unui pas în generator de cod. Eu nu am avea nevoie să frunzăresc zeci de fișieri sau să retrag un git commit.

Generarea codului previne repetarea codului boilerplate în zeci de fișiere, în același timp furnizând orice cod viitor cu unele capacități implicite din start. Este mai ușor de administrat, deoarece singurul lucru catre trebuie să schimbe pentru a afecta zeci de clase care au utilizat o capacitate particulară a generatorului de cod este doar regulile după care acel cod este construit. Este mai ușor de adăugat capacitățile noi, din aceeași cauză. Dă documentarea automată. Imaginați-vă păstrarea documentării la zi în toate acele fișiere.

### 1.5.2 Instumente în scurt

Utilizez **T4**, scurt pentru **Text Template Transformation Toolkit**, pentru a crea șabloane pentru a genera fișiere sursă adăugătoare. Utilizez **Roslyn**, pentru analiza codului sursă. Aș marca clasele mele în codului sursă cu atributuri specifice pentru a permite generarea anumitului cod când generatorul este pornit.

Abordarea mea la analiza codului este una simplistă. Eu nu monitorizez codul live, printr-o conexiune la language server. Când generatorul de cod este pornit, el șterge toate fișierile generate anterior și analizează întregul proiect din nou, generând toate fișierile din nou. Da, această abordare este foarte lentă dar este cu mult mai ușoară de implementat. Cea mai lentă parte a procesului este citirea și analiza fișierilor sursă, deci precis ar putea fi optimizată cu un language server.

### 1.5.3 Fluxul meu de lucru

Procesul meu de tranformare a codului repetativ în cod generat este aproximativ următorul:

1. Când scriu cod observ un pattern care poate fi exploatat de către generatorul de cod.
2. Dacă pattern-ul nu este destul de clar, aștept până când un pattern asemănător apare într-o altă bucată de cod, până când problema devine destul de clară ca să pot propune o soluție generală.
3. Încerc să rezolv problema fără a genera codul, cât mai simplu, printr-o abstracție.
4. Dacă nu pot rezolva astfel, pornesc generarea codului pentru ideea dată.

# 2 Prezentarea generală a sistemei

## 2.1 Prezentarea generală a mecanicilor jocului

După cum s-a menționat anterior, mecanicile jocului sunt băzate pe cele din Necrodancer.

Jocul are loc într-o grilă de două dimensiuni a lumii, și este băzat pe tururi. Controlați un caracter poziționat pe o celulă în acea grilă. Fiecare tur, puteți face o acțiune, de exemplu, să vă mișcați într-o direcție ortogonală la o celulă adiacentă, să atacați un inamic pe o celulă adiacentă, să săpați un obstacol sau să faceți o acțiune specială, de exemplu să aruncați o vrajă. Încă, este posibil să săriți peste un tur, fără a face nimic.

După ce v-ați efectuat acțiunea, toți inamicii primesc posibilitatea de a-și face acțiunea, unu câte unu. Ce acțiune ar fi selectată depinde de IA a lor (algoritm pentru selectarea următoarei acțiuni) și de fapt orice acțiune poate fi selectată, de la ataca sau mișcare simplă până la aruncarea unei vraje.

Mai sunt lucruri care au loc după aceasta, însă le vom discuta mai târziu.

### 2.1.1 Tipurile de acțiuni

Un concept important care trebuie să-l adresez este faptul că jucătorul poate selecta dintre următoarele două tipuri de acțiuni:

1. *Acțiuni-vector* (directed actions), governate de către input-uri direcționale (săgețile). Acestea includ atacarea, mișcarea și săparea într-o direcție specificată.
2. *Acțiuni speciale* (în mare parte undirected actions), ca aruncarea unei vraje sau activarea unui item. Aceste acțiuni sunt executate prin apasarea unei cheie desemnate, fără a necesita un input direcțional simultan (săgeata). Acele acțiuni speciale care totuși necesită o direcție pentru a lucra corect, ca, de exemplu, aruncarea unei sfere de foc într-o direcție specifică, pot utiliza orientarea curentă a caracterului, sau pot obține direcția într-un alt mod.

Lucrul cel mai important de înțeles este faptul că jucătorul nu poate controla ce acțiune exact va fi executată după furnizarea unui input vectorial. Mai precis, toate acțiunile posibile vor fi încercate în ordinea și prima acțiune care a reușit să se execute termină procesul de execuție.

De exemplu, dacă jucătorul dă input-ul sus, la început caracterul va încerca să atace în direcția sus, pe urmă, dacă atacul nu întâlnește nici un inamic, caracterul va încerca să sape obstacol din sus, pe urmă, dacă nu întâlnește nici un obstacol, caracterul ar mișca sus.

Acest lucru este opus acțiunilor speciale care sunt de obicei mapate direct la un anumit efect sau un anumit item. De exemlu, zicem, jucătorul furnizează inputul S care este mapat la ridicarea scutului. Deci, apăsarea lui S mereu ar executa această acțiune concretă (în general, dar sunt unele excepții).

Ca rezultat acestul model, jucătorul poate executa orice valabilă la moment acțiune în orice moment al timpului prin apăsarea cel mult unei cheie.

### 2.1.2 Scopul

Jucătorul se confruntă cu problema de a completa un nivel generat aleator. Nivelurile consistă dintr-un set de cameri conectate, fiecare cameră conținând inamici. Există o cameră finală cu o ușă (sau o trapă, sau o scară) la următorul nivel. Când jucătorul învinge un număr de așa niveluri, el confruntă un bos. Învingerea bosului ori permite jucătorul să se procede la următorul nivel, sau rezultă într-o victorie generală.

Nivelurile devin progresiv mai complicate. În special, monștri au mai multe puncte de sănătate, apar tipuri de monștri noi și mai complexe, numărul de pericoli, ca țepușe sau bălți de apă, crește, etc. În același timp, jucătorul primește itemi când învinge etajul, care dă abilități noi pasive sau active, sporește statuturile. Deci, jucătorul tot devine mai puternic, progresând.

### 2.1.3 Itemii

Inventoriul jucătorului are niște sloturi de itemi, fiecare cu un rol asociat, de exemplu, arma, lopata, vraja sau o parte de armură, ca cizmele sau casca. Acele sloturi care pot fi activate sunt mapate la un input, adică furnizând acel input ar activa itemul în slotul corespunzător.

Jucătorul poate ridica itemi dacă pășește pe ele, astfel ele sunt plasate în slotul desemnat lor automat. Dacă în acel slot deja este un item, acel item ar fi schimbat la cel ridicat și plasat pe podea.

Unii itemi pot să nu aibă un slot asociat. Așa itemi de obicei sporesc statuturile jucătorului, sau modifică subtil un comportament specific. De exemplu, ar putea fi un item care adaugă efectul pasiv de a dauna toți inamicii în jurul inamicului lovit.

Vom presupune, pentru simplitate, că nu pot exista două exact aceiași itemi ridicate de către jucător.

### 2.1.4 Inamicii

Fiecare inamic are un comportament clar-definit. Ei selectează acțiuni după o strategie ușor de înțeles pentru jucător.

De exemplu, un inamic simplu ar putea să aibă următoarea strategie: a sări peste o acțiune, după ce a ataca sau a se mișca în direcția jucătorului.

Acțiunile inamicilor trebuie să fie previzibile pentru jucător pentru a putea evalua repede o situație dată și a fi sigur în ce acțiune el va lua. Ideal nimic aleator sau neprevizibil nu trebuie să întâmple.

Fiecare inamic încă trebuie să aibă o metodă de a-l învinge, un pattern simplu de mișcări care jucătorul poate să urmărească și să câștige mereu. Bucuria jocului constă în studierea setului de mișcări al inamicului, găsirea unor pattern-uri și strategii de a-i învinge, și în evaluarea situației rapid, găsind o acțiune bună datorită înțelegerii bune a mecanicilor jocului, în cazul în care inamicii avansează în grupuri.

### 2.1.5 Limita de timp

După cum s-a menționat anterior, cea mai intrigantă idee este faptul că există o limită de timp pentru fiecare acțiune. Mai specific, acțiunile trebuie să fie selectate după ritmul muzicii (cu anumit spațiu de manevră).

Această detalie este esențială pentru design-ul jocului. Eu aș zice această mecanică este cea mai importantă mecanică din Necrodancer. Însă, ea este relativ independentă de celelalte mecanici ale jocului, ca deplasarea jucătorului în grilă și sistemul de itemi, și ea nu va figura în această lucrare. Acestă lucrare concentrează pe întrebările implementării altor părți ale jocului.

### 2.1.6 Mai multe idei

Când motorul este suficient de completat, va fi ușor să explorăm mai multe idei. Eu aș dori să încerc să transform jocul acesta într-o PVP arena, sau MOBA, lăsând mecanicii de bază și ideea să facem acțiuni după muzică neschimbate. Nu știu cât viabil aceasta ar fi, dar ideea îmi pare destul de intrigantă.

## 2.2 Prezentarea generală a design-ului sistemului.

Mă preocup în mare parte numai de motorul meu, adică cum ar funcționa logica, cum itemii, acțiunile, IA al inamicilor vor fi implementate, cu instrumentele de exemplu pentru generarea codului. Încă, sunt interesat să permit să extindem contentul existent prin moduri.

### 2.2.1 Cum să NU scrieți cod

Unul din cele mai importante subiecte în dezvoltarea jocurilor video este cum să arătăm frumos ce se face în joc pe ecran, cu animații, particule și sprite-urile corect arătate utilizatorului. O metodă de a face acest lucru este să ne referim la codul ce controlează **View**-ul, adică ce se vede pe ecran, direct în codul pentru logică (**Model**). De exemplu, cam așa (pseudocod pentru înțelegere, nu-i codul real din joc):

void Move(IntVector2 direction)  
{  
 if (!Grid.HasBlockAt(this.position + direction))  
 {  
 SetAnimation(Animation.Hopping);  
 TranslateSprite(  
 to: this.position + direction,   
 timeInMs: 500,   
 callback: () => SetAnimation(Animation.Idle));  
 this.position += direction;  
 }  
}

Însă, așa cod are niște defecte asociate cu el:

1. Logica jocului Dumneavoastră este cuplată strâns cu View-ul. Mixați codul care ar putea fi separat, astfel complicând procesul de a-l citi, înțelege și menține.
2. Codul este foarte instabil. Imaginați-vă penru un moment că jucătorul după ce s-a mișcat la o poziție nouă, a declanșat o capcană care l-a ucis. Aceasta ar trebuie să decalșe animația de moarte, însă animația inactivității (idle) setată în callback-ul este în curs de redare. Evident, acest exemplu este foarte simplificat, însă deja puteți vedea că setarea callback-urilor în așa mod este ceva inadmisibil. Aveți nevoie de un sistem mai complex pentru a administra aceasta.
3. Ce dacă jucătorul alunecă în loc de a sări? Atunci, o animație diferită trebuie să fie setată, nu Animation.Hopping, ci Animation.Sliding. Ați adăuga o verificare în funcția Move()? Dar ce dacă alunecarea vine dintr-un mod? Atunci, sistemul Dumneavoastră nu ar fi putut să aibă cunoștințe despre aceasta. Este clar, că așa strategie simplă nu va lucra aici.

Deci, defectele identificate sunt:

* cuplarea strânsă;
* probleme de întreținere;
* inflexibilitatea.

### 2.2.2 Separarea și event-urile este ideea cheie

Voi ilustra cum separarea componentelor și event-urile pot rezolva toate problemele constatate mai sus. Deci, în primul rând vom adresa problema cuplării strânse. Imaginați-vă două funcții, una responsibilă pentru mișcare, alta pentru animații.

void Move(IntVector2 direction)  
{  
 if (!Grid.HasBlockAt(this.position + direction))  
 {  
 this.position += direction;   
 }  
}  
  
void AnimateMove(IntVector2 newPosition)  
{  
 SetAnimation(Animation.Hopping);  
 TranslateSprite(  
 to: newPosition,   
 timeInMs: 500,   
 callback: () => SetAnimation(Animation.Idle));  
}

Codul nu este ideal, din punct de vedere a implementării (callback-urile, etc.), dar aici avem o problemă mai mare. Curent, nu există o modalitate de a comunica pentru aceste funcții. Apelarea lui AnimateMove() în Move() nu lucrează, deoarece aceasta ar însemna că doar am refactorizat codul legat de animație într-o funcție separată, însă ele au rămas cuplate strâns. Scopul nostru era să separăm codul pentru logică de la codul pentru view. Cum să facem acest lucru? Event-urile (semnale) la salvare!

Ideea este să definim o coadă de handler-uri, codul în care să fie executat după ce jucătorul se mișcă. Această coadă poate fi statică, configurabilă pentru tipuri diferite de entități. Încă în pseudocod:

static EventQueue<Handler> moveEvent;  
  
void Move(IntVector2 direction)  
{  
 if (!Grid.HasBlockAt(this.position + direction))  
 {  
 moveEvent.Dispatch(this, this.position + direction);  
 this.position += direction;  
 }  
}  
  
void AnimateMove(IntVector2 newPosition)  
{  
 SetAnimation(Animation.Hopping);  
 TranslateSprite(  
 to: newPosition,   
 timeInMs: 500,   
 callback: () => SetAnimation(Animation.Idle));  
}  
  
void Setup()  
{  
 moveEvent.AddHandler(AnimateMove);  
}

Acum, funcția Move() nu știe nimic despre view. Ea doar expediază event-ul fiecare dată când jucătorul se mișcă.

Însă, aceasta nu rezolvă problema cu, de exemplu, alunecarea. Jucătorul nu alunecă implicit. Alunecarea este un efect care poate fi aplicat în timpul rulării. Dacă am dori să animăm alunecarea corect, ar trebui să schimbăm această coadă în timpul rulării. Deci, vom crea o coadă de event-uri într-o proprietate a instanței jucătorului, nu doar a tipului jucătorului, ca să putem s-o modificăm în timpul rulării.

Acum, după ce le-am separat, putem rezolva și problemele de întreținere. Deoarece partea lui view va fi îmbunătățită și va devine un sistem complet și independent, această problemă tot poate fi rezolvată, cu ceva mai mult chibzuit.

## 2.3 O direcție greșită?

Deci, ideea mea inițială era că modelul va fi separat de la view-ul, dar nu am știut cum să fac acest lucru exact. Am știut despre event-uri și le-am utilizat, însă realizarea că ele pot fi utilizate pentru comunicarea dintre view-ul și model-ul atunci încă nu a venit la mine. Pur și simplu m-am gândit despre problema puțin diferit. M-am gândit că view-ul și model-ul sunt aceste două sisteme complet independente, view-ul fiind conectat cu model-ul printr-un pod minuscul. Aceasta poate lucra, însă nu este tare scalabil. În loc de această abordare, view-ul trebuie să fie conectat cu modelul într-un set lat de puncte de contact, prin event-uri, unde modelul nu ar cunoaște nimic despre view-ul.

### 2.3.1 Ideea istoriei

Inițial, mi-am imaginat modelul și view-ul să fie conectate prin *istoria*. Modelul ar împinge actualizările referitor la ce event-uri s-au întâmplat în lume prin această istorie. De exemplu, când jucătorul ar fi atacat, actualizarea fiind atacat ar fi salvată în istorie. View-ul și-ar actualiza starea și ar decide ce animații să ruleze după ce toate event-urile au avut loc.

Deci, mi-am imaginat această în așa mod: avem o mulțime de mașini de stări separate în view-ul jucătorului, toate responsabile pentru detectarea event-urilor diferite. De exemplu, există o mașină de stări pentru fandare. “Fandare” înseamnă o atacă imediat urmărită de o mișcare. Mai este o mașină de stări pentru mișcare care constă din doar actualizarea de mișcare.

Deci, după ce s-a procesat mai un tur în model și istoria s-a umplut, view-ul ar primi istoria și ar încerca să ruleze toate mașinile de stări pe această istorie. Deci, după ce jucătorul și a atacat, și a mișcat în același tur, view-ul ar primi istoria cu 2 actualizări: atacul și mișcarea. Fiecare din aceste mașini de stări ar fi încercate. Fandarea este atac și mișcare, deci această mașină de stări ar fi satisfăcută. Mașina de stări care reprezintă mișcare tot ar reuși, deoarece actualizarea de mișcare este prezentă. Din aceste două view-ul ar selecta pentru animare una mai complexă, adică fondarea.

Chiar am avut un termen mai intuitiv pentru această: sitele. Fiecare mașină de stări este o sită care devine blocată când încercați să ciuruiți istoria prin ea. După ce am trecut istoria prin toate sitele, cea mai complexă blocată sită este selectată și animațiile pentru acea sită sunt redate.

### 2.3.2 Care este problemă dar?

Problema vine când aveți nevoie să transmiteți datele împreună cu actualizările.

Ați putea transmite orice date cu ele, însă atunci nu avem o modalitate clară de a vedea ce event a avut loc fără de a face un cast al datelor într-un tip cunoscut. Aceasta aduce la if-else urâte pentru a determina tipul corect.

foreach (object update in history)  
{  
 if (update is AttackingUpdate attackingUpdate)  
 {  
 // facem ceva cu datele din `attackingUpdate`  
 // ...  
 }  
 else if (update is MovingEvent movingEvent)  
 {  
 // ați înțeles ideea  
 }  
}

Din cauza că gaură prin care încercați să împingeți aceste actualizări de la model la view-ul este atât de îngustă, aveți nevoie să convertați actualizările într-un tip analog lui object, astfel pierzând tipul concret al actualizării în proces. Acest fenomen poartă numele “ștergerea tipului.”

Stop, oare nu putem utiliza polimorfizmul în loc de if-else pentru a apela funcțiile concrete care trebuie să proceseze datele?

Având în vedere faptul că modelul nu știe nimic despre logica view-ului Dumneavoastră asociată cu actualizările, însă știe ce datele vor fi în update-urile, nu, aceasta nu este posibil. Nu puteți da modelului un tip cu acele date pe care el l-ar instanția (ca obiectul să conțină funcțiile polimorfice Dumneavoastră). Adică, *ați putea*, însă nu este tare convenabil si sunt modalități mai simple de a atinge același rezultat.

Ați putea menține un dicționar cu tipurile actualizărilor mapate la handler-uri, ca mai jos. Însă acest cod este rău și nu este plăcut de menținut.

void HandleAttack(object update)  
{  
 // În primul rând, castăm în tipul dorit  
 var attackingUpdate = (AttackingUpdate) update;  
 // Facem ceva cu datele ...  
}  
  
void HandleMove(object update)  
{  
 // În primul rând, castăm în tipul dorit  
 var movingUpdate = (MovingUpdate) update;  
 // Facem ceva cu datele ...  
}  
  
// Type este type info unei clase date.  
// Action<T> este void function care ia T ca argument.  
Dictionary<Type, Action<object>> typeErasedHandlers  
{  
 { typeof(AttackingUpdate), HandleAttack },  
 { typeof(MovingUpdate), HandleMove }  
};  
  
void SieveThroughHistory(History history)  
{  
 foreach (object update in history)  
 {  
 typeErasedHandlers[typeof(update)](update);  
 }  
}

### 2.3.3 Soluția

Din fericire, există o modalitate mai bună de a rezolva problema.

Ideea este să permitem mai multe puncte de contact între model-ul și view-ul. Astfel, putem sări peste faza istoriei cu totul. Modelul nu ar trebui să împingă nici o actualizare în istorie. El doar ar declanșa un event corespunzător cu toate datele cu care la moment lucrează, păstrate într-un context. Ca un exemplu simplificat (codul nu este real):

class AttackingContext  
{  
 Player player;  
 Enemy attackedEnemy;  
 IntVector2 direction;  
}  
  
class Player  
{  
 // ...  
  
 EventQueue<AttackingContext> attackEventQueue;  
  
 void Attack(IntVector2 direction)  
 {  
 var enemy = Grid.GetEnemyAt(this.position + direction);  
   
 if (enemy != null)  
 {  
 var context = new AttackingContext(  
 player : this,  
 attackedEnemy : enemy,  
 direction : direction);  
   
 attackEventQueue.Dispatch(context);  
  
 enemy.TakeDamage(this.damage);  
 }  
 }  
}  
  
class PlayerView  
{  
 // ...  
  
 void AttackHandler(AttackingContext context)  
 {  
 // Facem ceva cu:  
 // context.player  
 // context.attackedEnemy   
 // context.direction   
 }   
  
 void Setup(Player playerInstance)  
 {  
 playerInstance.attackEventQueue.AddHandler(AttackHandler);  
 }  
}

Această idee poate vă pare evidentă după ce am ilustrat-o, însă pentru mine ea nu era evidentă până recent. Trebuia să sufăr prin toate probelemele istoriei explicate mai sus pentru a ajunge la această revelație.

Deci, am reușit șă separăm view-ul de la modelul, în același timp având posibilitate de a transmite datele de la model la view fără ștergerea tipurilor și chiar să evităm ca model să cunoască despre existența view-ului, datorită unui astfel de design.

### 2.3.4 Este oare totul?

Mai sunt niște probleme cu așa design. Una din ele este legată de inconviența unelor lucruri care apar din cauza că design-ul este așa. Le-am adresat prin generarea codului. O altă problemă este legată de ordonarea handler-urilor. Am rezolvat-o prin introducerea priorităților.

Vom discuta ambele mai târziu.

## 2.4 ECS (Entitate-Component-Sistem)

S-au spus multe lucruri despre ECS. Însă, eu sunt convins că nu puteți să le înțelegeți integral dacă nu redescoperiți această idee singuri. Când vedeți o problemă reală și încercați s-o soluționați prin diferite metode, incluzând ECS, iată atunci apare înțelerea profundă.

### 2.4.1 Introducere

ECS permite să privim spațiul programului printr-o perspectivă diferită.

ECS zice că există o lume și orice obiect în acea lume este o *entitate*. Toate entitățile încep ca un obiect vid doar cu un identificator. Ele sunt ca un schelet la care adăugăm componente pentru a le da un comportament sau o proprietate specifică. Componentele de obicei doar conțin datele.

Toate mecanicile din joc sunt pur și simplu interacțiunile dintre diferite obiecte din lume. Acestea sunt conceptualizate ca *sisteme*. Ele operează pe componentente individuale ale entităților, astfel asigurându-le comportament specific.

Ideea după ECS-ul este “entitățile flexibile și dinamice.”

### 2.4.2 De ce nu POO?

Dacă vreodată ați încercat să reprezentați tipurile diferite ale entităților într-un mediu dinamic, știți că aceasta nu va lucra.

1. Nu puteți utiliza conceptul de moștenire și ierarhiile cum-se-cade.
2. Tipurile statice sunt prea rigide.

Deci, imaginați-vă pentru un moment că aveți clasa Player. Jucătorul poate face multe lucruri, printre care mișcarea, atacarea și săparea.

Acum, considerați o clasă diferită, Enemy. Puteți observa că inamicul tot poate să se miște și să atace, dar are o schemă de control diferită: jucătorul este controlat de către input-ul utilizatorului, pe când inamicul de o inteligență artificială.

Deci, probabil veți fi tentați să refactorizați lucrurile comune, anume atacarea și mișcarea, într-o clasă de bază. Clasa Player atunci ar moșteni acea clasă de bază adăugând abilitatea de a săpa, pe lângă schemei proprii de input, iar clasa Enemy ar adăuga iteligența sa artificială.

Acum apare un tip nou de inamic: el poate să atace, să se miște și să sape, având un IA. Unde să-l punem în ierarhia noastră? Ar trebuie el să moștenească Enemy? Dar atunci ar trebui să adăugăm săparea, care este deja implementată în clasa Player. Să scoatem săparea într-o clasă de bază DiggingMovingAttackingBase? Nu, deoarece atunci nu putem moșteni IA din clasa Enemy. Poate Enemy trebuie să moștenească DiggingMovingAttackingBase? Din nou nu, deoarece el nu poate săpa după design.

Deci, chiar cu așa exemplu simplist, ideea tipică lui POO a moștenire se destramă. Acum imaginați-vă același scenariu dar amplificat în sute de ori: sunt sute de proprietăți și comportamente pe care orice entitate poate să le posede. Aceasta ar fi imposibil de modelat printr-o ierarhie.

O altă idee este faptul că entitățile, dacă sunt modelate ca instanțe de tipuri statice, nu pot să-și schimbe comportamentul în timpul rulării. Într-un joc real, jucătorul poate să înceapă fără abilitatea de a săpa, însă, când primește lopata, ar învăța această abilitate nouă. Însă, nu puteți modifica clasa Player să poată sape. S-o modificați de la început nu are sens, deoarece jucătorul a obținut abilitatea *eventual*.

Încă un exemplu: aveți un monstru de două faze, zicem, un fluture furios care inițial începe ca o omidă inocentă, deci nici nu poate ataca, însă care poate să se transforme într-un fluture, obținând abilitatea de a zbura și de a ataca jucătorul.

Cu POO există doar un mod de a modela această transformare. Ați avea două clase, una pentru starea de omidă, alta pentru starea de fluture. Ca să transformați omida în fluture trebuie să distrugeți instanța omidei si să creați un fluture nou.

Cu componentele dinamice aveți două posibilități. Puteți ori să facem cum am descris mai sus, ori să transformați omida în fluture dându-i comportamentele de Flying și de Attacking. În acest sens, a două abordare este mai flexibilă.

### 2.4.3 Compresie

O altă idee este să dați fiecărei entități întreaga gama tuturor proprietăților și abilităților posibile însă să nu le dați voie să utilizeze majoritatea lor. Astfel, ar fi ușor să aprindeți unele abilități mai târziu: puteți pur șă simplu să setați sau să ștergeți acel flag care indică dacă entitatea poate aplica acea abilitate.

Avem două probleme cu așa abordare:

1. Cât de multe componente și proprietăți aveți în joc, atât de umflate entitățile Dumneavoastră devin, atât de mult spațiu ele ocupă. Nu doar una din ele, ci toate de o dată.
2. Așa sistem nu poate fi expandat de moduri, ceea ce-i inacceptabil în cazul meu. Unul dintre obiectivele proiectului meu este de a permite modarea.

Deci, păstrarea componentelor în constrast cu lumea unde toate entitățile au toate proprietățile posibile, natural aduce la entități *rarefiate*, în alte cuvinte, la ideea *compresiei*.

### 2.4.4 ECS-ul meu

La moment, perspectiva mea la ECS este ceva specială. - Noțiunea *sistemei* este destul de vagă în codul meu. - Există distincția dintre *componentele cu date* (sau simplu *componente*) și *comportamentele*. - În codul meu, comportamentele sunt acele componente care definesc *event-uri* (utilizez *chain-uri*, mai mult ulterior). Deci, comportamentele în codul meu este o fuziune dintre componente și sisteme. Cum am descris anterior, event-urile sunt esențiale pentru a lega view-ul și model-ul. - Există conceptul unui tip de entitate. La moment, tipul este de fapt un șablon după care entitățile sunt construite la instanțiere. Tipurile la moment sunt modelate printr-o entitate (un *subiect*) care este copiată la instanțiere pentru a crea o instanță nouă de acel tip. Instanța atunci devine independentă de subiect și poate să se schimbe în timpul rulării în orice mod, fără a-l afecta pe subiect. Așadar, tipurile pot fi augmentate cu componente în timpul construcției, la fel ca entitățile în timpul rulării.

În alte sisteme ECS, după cum am aflat numai recent, un tip de entitate se numește un **arhetip**.

Sunt nelniștit referitor la performanța ECS-ului meu. Chestia este, ECS-ul meu este, cum se spune, “fake.” Un ECS optimizat de obicei focusează pe stocarea diferitelor componente într-un loc central în memorie pentru a putea itera printre ele în sisteme (iterarea secvențială este mult mai rapidă). Administrarea memoriei manuală îmbunătățește performanța, salvând lucru pentru GC (colector de gunoi).

Eu fac aceasta în modul “nu-mi pasă,” adică, aloc toate componentele în memorie dinamică cu operatorul new. Aceasta este, de fapt, modul cel așteptat și cel ușor de a face acest lucru în C#, însă nu este performant deloc.

Să faceți un ECS într-un mod corect este o sarcină extrem de complicată pentru C#. Structuri și tablouri de structuri este singurul mod în care datele pot fi păstrate direct în memorie dar nu împrăștiate undeva în memorie dinamică. C# nu dispune de instrumentele necesare pentru a gestiona memoria manual, care există de exemplu în C++, deoarece nu se așteptă de la programator să facă așa ceva în C#. Am considerat să migrez proiectul în C++, însă C++ tot are problemele sale, de exemplu că modding-ul ar fi mai complicat de implementat, că serializarea este proastă, deci am hotărât să progrezez cu ECS-ul “fake” al meu, în loc de aceasta. În viitorul apropiat este posibil că voi migra proiectul pe **D** care este atrăgător pentru mine în special din cauza facilităților sale de metaprogramare.

# 3 Subiectele tehnice

În această secțiune, prezint unele elemente din joc. Mai specific, explic motivarea lor și cum le-am implementat, cu exemple concrete din codul sursă.

## 3.1 Grila

După cum s-a stabilit anterior, lumea este reprezintată printr-o grilă de două dimensiuni cu entitățile. Întrucât interogările de a afla dacă o entitate se află într-o celulă specifică, dacă există un bloc pe o celulă specifică sunt atât de răspândite, am beneficia dacă am păstra entitățile (mai explicit, *transform-urile* lor) în coordonatele curente, într-un tablou de două dimensiuni. Aceasta este de fapt cum am decis să modelez grila ([uitați-vă la costructor](https://github.com/AntonC9018/hopper.cs/blob/5b3156f38a03867272357085813409e9076cfc6d/Core/World/Grid/Grid.cs#L30)).

### 3.1.1 Celulile

Se presupune că fiecare celulă are mai multe nivele, unde entitățile de la diferite nivele au proprietăți ceva diferite. De exemplu, în general, capcana cu țepușe care dăunează jucătorul când acela o calcă, nu poate fi atacată de către jucători sau inamici, dar poate fi exploadată de către bombe. Aceasta este capcana este locată pe nivelul trap în celula în care se află, pe când jucătorul sau inamicii pot viza doar nivelul real cu atacurile normale. În orice caz, am modelat astfel ideea.

Înainte de refacere, am avut câte un slot pentru fiecare nivel al celulei, dar aceasta nu era bine, minimum deoarece majoritatea nivelurilor erau vide aproape mereu. A se vede, de exemplu, [clasa celulei din codul precedent, în lua](https://github.com/AntonC9018/Dungeon-Hopper/blob/master/world/cell.lua#L19).

Acest design a avut un minus: poate fi doar o entitate la fiecare nivel în fiecare moment al timpului. Aceasta face unele lucruri, de exemplu entități care trec prin alți entități, dificil sau imposibil de implementat sau de considerat.

Am relizat că pot păstra entitățile într-o listă, și itera prin această listă, pentru a lua o entitate din nivelul care mă interesează. Căutarea lineară ar fi de fapt acceptabilă în acel scenariu, deoarece celulele de obicei nu am mai mult decât 1-2 entități. Cazurile unde ele conțină mai multe entități sunt rare și pot fi neglijate.

Încă un beneficiu de așa abordare este faptul că o entitate poate să-și schimbe nivelul în care ea se află în timpul rulării, fără a actualiza unde ea este păstrată în celulă.

Implementarea curentă a celulii implică moștenirea de la List<Transform>. [A se vedea implementarea curentă.](https://github.com/AntonC9018/hopper.cs/blob/5b3156f38a03867272357085813409e9076cfc6d/Core/World/Grid/Cell.cs#L8) Eu personal nu văd nimic rău în așa abordare, cu toate că [unii spun multe lucruri răi despre aceasta.](https://stackoverflow.com/questions/21692193/why-not-inherit-from-listt) Oamenii deseori ridic tema *compoziție peste moștenire*, cum compoziția este mai flexibilă, însă consider că aici un avem un caz de așa ceva. Aici mai mult este vorba despre evitarea boilerplate-ului prin faptul că nu scrieți implementarea pentru interfața IList<Transform>, prin înaintarea tuturor metodelor la un membru privat de tip List<Transform>. În același timp această abordare poate aduce la erori deoarece utilizatorului îi permitem mai multe lucruri de decât el trebuie să fie conștient. De exemplu, dacă castăm celulă în listă, ceea ce putem face în acest caz, putem folosi metoda Add() furnizată de către clasa listei, decât de clasa celulei, care mai conțină unele asserturi pentru debug.

Mie personal nu-mi pasă ce zic oameni la această temă. Eu voi face ceea ce îmi pare admisibil mie. Implementarea interfeței prin scrierea manuală a metodelor de înaintare la un list-membru nu mă face să mă simt bine. În același timp, încă nu mi-am dezvoltat stilul meu propriu de programare în C#, și personal nu sunt de acord cu unele sfaturi ce oameni dau. Poate când progresez profesional, voi înțelege mai bine.

Încă, un test rapid a indicat că dacă de fapt ne moștenim de la List codul merge cu aproape 1.5 ori mai rapid.

Să fie notat, țiglele statice nu sunt considerate ca entități și de aceea nu sunt păstrate în grilă. Același lucru se aplică la efectele de particule, care nu au influența asupra mecanicilor din joc. Modelul este responsabil doar pentru chestiile care sunt legate de logica jocului.

### 3.1.2 Componentele responsabile pentru poziție și mișcare

Evident, abilitatea de a ocupa o poziție în lume și de a putea să-și schimbe poziția în timpul rulării este esențială pentru joc.

Aceste abilități sunt modelate după următoarele componente specializate: - Transform, dând o *poziție în lume*, - Displaceable, dând abilitatea de *a-și schimba poziția în lume*, - Moving, dând abilitatea de *a se mișca volunar*, - Pushable, dând abilitatea de *a fi mișcat involuntar*.

#### 3.1.2.1 Transform

Entitățile care pot fi poziționate în lume trebuie să aibă componenta [Transform](https://github.com/AntonC9018/hopper.cs/blob/5b3156f38a03867272357085813409e9076cfc6d/Core/World/Grid/TransformComponent.cs#L16). Conține informația despre poziția curentă în lume, orientarea curentă (în ce direcție se uită) și ce nivel entitatea ocupă. Fiecare transform mai conține o referință la entitate, pentru a putea accesa entitatea când interogăm grila.

Curent, există conceptul de a fi directat care va fi examinată pe urmă. Este modelată printr-un tag (o componentă fără date), însă recent am adăugat un flag în Transform pentru această. Astfel am putea introduce mai multe flaguri.

Transform mai conține metode ajutătoare pentru interacțiunea cu grila. Această componentă este cuplată cu grila. Aceste metode sunt definite ca metode instanțe pentru transform simplu pentru comoditate. Ele ar putea fi definite ca metode de extindere, sau ca metode pe Grid, deoarece majoritatea lor are analoguri pe Grid.

Transform curent lucrează cu grila globală, adică, presupune că există *doar o lume în același timp*. Aceasta am făcut în primul rând pentru comoditate, deoarece anterior toate transform-urile au avut o referință la lumea în care ele se află. Însă, am schimbat acest lucru în mare parte deoarece aproape toate funcțiile într-un mod referă la grilă și mi-a fost anevoios să transmit manual această referință la toate funcțiile.

Când voi adaug posibilitatea pentru mai multe lumi de a exista deodată, cumva voi schimba aceast lucru. Însă, sper că patch-ul nu va fi unul dificil, având în vedere faptul că codul logicii este de un singur thread, ceea ce înseamnă că am putea să schimb lumea globală când se schimbă lumea curent procesată.

Dacă v-ați uita la cod mai aproape, ați putea să observați că unele câmpuri sunt decorate cu atribute. Aceasta este legat de generatorul de cod. În scurt, atributul Inject este utilizat pentru a genera un constructor și un constructor de copiere pentru această componentă, care ar solicita o valoare pentru acel câmp ca parametru.

Probabil ați observat și apelările la metodele Grid.TriggerLeave() și Grid.TriggerEnter(). Cum acestea funcționează va fi explicat mai târziu.

#### 3.1.2.2 Displaceable

Schimbarea poziției proprii într-o direcție dată, fie voluntar sau nevoluntar, este conceptualizat ca *deplasare*. Teleportarea la o poziție nouă nu este considerată ca o deplasare.

Acest comportament permite deplasarea dacă celula unde entitatea se mișcă nu este blocată. Informația ce nivel să fie considerat ca nivelul de blocare este stocat ca un câmp injectat în acest comportament deci poate fi schimbată în timpul rulării pentru o entitate particulară, dacă necesită.

Displaceable este un *comportament* care poate fi adăugat la o entitate pentru a-i da posibilitate de a se deplasa. [Codul sursă](https://github.com/AntonC9018/hopper.cs/blob/25612ec4438f39f8b590c3a7426c5f0b6a8dea78/Core/Components/Basic/Displaceable.cs). În această clasă puteți vede aproximativ modul în care comportamentele sunt implementate. [Aici](https://github.com/AntonC9018/hopper.cs/blob/25612ec4438f39f8b590c3a7426c5f0b6a8dea78/Core/Components/Basic/Displaceable.cs#L59) definim niște chain-uri (ca atare event-uri). Generator de cod reacționează la ele, inițializându-le într-un constructor generat automat, și le copiază în constructorul de copiere generat automat.

Dacă codul extern dorește să adauge handler-uri la aceste chain-uri pe o instanță de entitate (sau pe un tip de entitate, deoarece fabricele sunt modelate print-o instanță de entitate-subiect), ar utiliza atributul Export. Aplicând acest atribut ar autogenera codul pentru alocarea unui număr de prioritate unic pentru acel handler și, opțional, ar genera un învelitor care ar putea fi utilizat pentru a ușura procesul de obținere a chain-ului necesar de pe entitate și de conectare a acelui handler.

[Aici](https://github.com/AntonC9018/hopper.cs/blob/0bcc623cb17d56f765b402860cd0e62e31885ad2/Core/Retouchers/Reorient.cs#L12) avem un exemplu de așa atribute utilizate pentru a expune o funcție handler la generator de cod.

Diferența dintre chain-urile în comportamentul Displaceable este că ele sunt executate în momente diferite ale procesului de deplasare: - Check se face înainte de deplasare, verificând dacă deplasarea trebuie să fie încercată cu totul. Dacă am chain-ul de verificare până la capăt fără stopare, acțiunea de deplasare se consideră reuțită, cu toate că entitatea ar putea să nu se miște în proces. Acesta este by design. - BeforeRemove reprezintă al doilea verificare pentru a verifica dacă deplasarea trebuie să fie aplicată. Diferența dintre aceste două chain-uri este că, dacă BeforeRemove ar eșua, adică, un handler ar opri propagarea contextului la restul handler-urilor, mișcarea s-ar fi considerată reușită, cu toate că nu ar fi executată în așa caz. Diferența există în mare parte pentru sistemul de acționare, explicată mai pe urmă. Însă ideea cheie este că chain-urile Check sunt utilizate pentru a verifica *dacă o acțiune trebuie să fie încercată*. Dacă deplasarea eșuează la Check, de exemplu *atacarea va fi încercată*, însă dacă eșuează mai pe urmă la BeforeRemove, acțiunea de mișcare va reuși și *nici o acțiune după mișcare nu ar fi încercată*. - BeforeReset este traversată imediat după ce entitatea a fost eliminat din grilă, dar înainte de a fi *resetată* în grilă. Cum ați putea observa, este traversată direct, fără verificare propagării, deci toate handler-urile se execută garantat după ce \_BeforeResetChain.Pass(ctx); este apelat. - After este traversată după ce entitatea a fost resetată în grilă.

Atât de multe chain-uri sunt necesare pentru a putea să schimbe comportamentul de deplasare într-un mod foarte specific. De exemplu, funcția particulară după [acest link](https://github.com/AntonC9018/hopper.cs/blob/0bcc623cb17d56f765b402860cd0e62e31885ad2/Core/Retouchers/Reorient.cs#L12) face așa că când entitatea se deplasează, orientarea lui ar fi schimbată în acea direcție. Însă v-ați putea imagina ideile foarte complicate implementate datorită acestor chain-uri. De exemplu, [*alunecarea* utilizează chain-ul After](https://github.com/AntonC9018/hopper.cs/blob/0bcc623cb17d56f765b402860cd0e62e31885ad2/TestContent/Modifiers/Sliding/SlidingEntityModifier.cs#L58) pentru a opri alunecarea când entitatea se ciocnește cu peretele sau plece de pe o suprafață alunecoasă.

Eu numesc această idee de aplicarea unor detalii mici la algoritmul de deplasare, *retouching*, ca adăugarea unor detalii sau efecte în Photoshop.

#### 3.1.2.3 Moving

Moving este comportamentul responsabil de deplasarea voluntară.

Comportamentul Moving este un *comportament direcționat activat*, ceea ce implică că el are o funcție Activate() care ia o direcție și returnează un boolean, indicând dacă activarea a reușit. Așa comportamente pot fi activate de către *sistemul de acționare*. Moving utilizează Displaceable pentru a executa deplasarea.

Moving este un exemplu de comportament care utilizează *autoactivarea*. *Autoactivarea* este o modalitate oferită de către generatorul de cod care permite să generăm funcția Activate() și 2 chain-uri Check și Do automat. Scopul lui Check ar fi să verificăm dacă trebuie să traversăm chain-ul Do. Chain-ul Do ar conține handler-urile care fac ceea ce ține de mișcare în acest caz. Aici încă vedem uzul funcției DefaultPreset() care ar seta aceste chain-uri inițial, aplicând handler-urile necesare.

Acest pattern este răspândit între comportament, dar este cel mai util pentru *prototipare*. Strategia unde utilizăm Check și Do lucrează pentru majoritatea comportamentelor la început, însă eventual deseori realizați că aveți nevoie de mai mult control, de exemplu de chain-urile Before sau After. Atunci veți defini toate pe care le aveți nevoie în comportamentele Dumneavoastră particulare, lăsând autoactivarea în urmă.

De exemplu, Displaceable a fost inițial un comportament autoactivat.

[A se vedea codul sursă.](https://github.com/AntonC9018/hopper.cs/blob/0bcc623cb17d56f765b402860cd0e62e31885ad2/Core/Components/Basic/Moving.cs)

#### 3.1.2.4 Pushable

Pushable este asemănător un comportament *autoactivat*, însă nu este *direcționat activat*, deoarece acțiunea asociată cu el nu poate fi executată voluntar.

Codul lui Pushable la moment nu este matur, deci nu pot explica mult aici.

[A se vedea codul sursă.](https://github.com/AntonC9018/hopper.cs/blob/0bcc623cb17d56f765b402860cd0e62e31885ad2/Core/Components/Basic/Pushable.cs)

### 3.1.3 Bloc

Ideea că o entitate nu poate să se miște la o celulă este conceptualizată spunând că acea celulă este *blocată* de o altă entitate. Tipic, această entitate ar fi ori de la nivelul *real*, ori de la nivelul *wall*.

Cum am notat anterior, blocarea mișcării este implementată în Displaceable. Blocuri mai pot afecta *sistemul de selectare țelelor*, explicată mai târziu.

#### 3.1.3.1 Entitățile direcționate

Proprietatea de a fi direcționat semnifică că entitatea ar ocupa doar o parte a celulii în care ea se află. Așa entități direcționate care servesc ca blocuri direcționate sunt numite *bariere*. [Unul din teste](https://github.com/AntonC9018/hopper.cs/blob/25612ec4438f39f8b590c3a7426c5f0b6a8dea78/.Tests/Core_Tests/GridTests.cs#L169-L195) explică ideea blocurilor direcționale cu ASCII mai clar decât eu aș putea prin text. Aceasta introduce mai multă complexitate în procesul de detectare dacă o celulă particulară este blocată.

De obicei, pentru blocuri nedirecționate, pur și simplu trebuie să verificăm doar o celulă pentru a determina dacă celula este blocată sau nu. Dacă celula conține o entitate din nivelul de bloc Dumneavoastră, da, dacă nu conține, nu. Cu blocuri direcționate aceasta este mai subtil.

Dacă entitatea din nivelul de bloc Dumneavoastră este direcționat, trebuie să verificați dacă este la partea corectă a celulei (ce parte ea ocupă este indicată de către orientarea ei). Partea corectă a celulei este partea din care caracterul Dumneavoastră o ar intra. Dacă entitatea potențial bloc ar fi pe orice altă parte a celulei, mișcarea nu ar fi blocată.

Însă nu-i totul! Mai trebuie să verificați dacă celula de pa care caracterul începe mișcare conține un bloc direcțional pe cealaltă parte a celulei de pe care caracterul o ar ieși.

Am implementat toate acestea în [funcția HasBlock()](https://github.com/AntonC9018/hopper.cs/blob/25612ec4438f39f8b590c3a7426c5f0b6a8dea78/Core/World/Grid/Grid.cs#L189) din Grid. Deoarece trebuie să știem din ca parte caracterul ar ieși, luăm ca input direcția pe lângă coordonatelor celulei de interes.

### 3.1.4 Event-urile de intrare și de ieșire

Grila mai definește [4 structuri utile](https://github.com/AntonC9018/hopper.cs/blob/408ae5fb9ec73fa3426648442d122c57f623a6ef/Core/World/Grid/Grid.cs#L16-L19): - TriggerGrids normală de Enter și Leave; - TriggerGrids filtrată de Enter și Leave;

Acestea încă nu și-au demonstrat valoarea utilă în cod, însă le-am utilizat pentru a implementa efectele de *alunecare, bouncing, fixare* și în *proiectile*.

Esențial, aceste grile permit să adăugați funcții handler care vor fi executate când orice entitate intră (sau iese de pe) celula. Diferența dintre grilele cele normale și filtrate este faptul că în grilele normale, handler-ul Dumneavoastră *ar fi eliminat după terminarea turului curent*, ceea ce înseamnă că handler-ul va persista numai până la moment când toate entitățile își termină acțiunile. În constrast, handler-urile adăugate în grila filtrată decid dacă ei trebuie să fie păstrate sau scoase singuri, returnând true sau false. Pentru un exemplu, vedeți [Leave handlerul lui bouncing](https://github.com/AntonC9018/hopper.cs/blob/408ae5fb9ec73fa3426648442d122c57f623a6ef/TestContent/Mechanics/Bouncing/Bouncing.cs#L52).

Remarc că acest API încă nu este complet și am putea să decid să-l schimb în viitor. Pur și simplu am explorat ideea care mi-a părut utilă.

De exemplu, [acest handler](https://github.com/AntonC9018/hopper.cs/blob/408ae5fb9ec73fa3426648442d122c57f623a6ef/TestContent/Mechanics/Bouncing/Bouncing.cs#L52) captează entitatea ca primul argument. Captarea (crearea closure-ilor) ideal am s-o schimb la un lookup pe id al entității pentru a permite entităților să fie “garbage collected” imediat. Însă, să fac acest lucru manual ar fi anevoios. Eu aș putea să utilizez iarăși generatorul de cod pentru acest scop în viitor.

## 3.2 Chain-urile

În secțiunile precedente am apăsat ușor ideea de *chain-uri*. Acestă secțiune dă o descriere mai detaliată despre ce ele sunt.

### 3.2.1 Resposibility chains

*Chain-urile* în codul meu sunt băzate pe ideea unui **lanț de responsibilitate** (responsibility chain).

Un lanț de responsibilitate este o listă de funcții handler care operează cu anumite date. Datele pot fi simple, ca un număr, dar pot fi și mai complexe, în care caz de obicei sunt numite *context*.

Sensul apelării acestor handleri este de a primi un oarecare rezultat sau de a aplica un oarecare efect. După ce unul din handler-uri au reușit să-și aplică efectul sau să-și calcula rezultatul, propagarea se termină, adică nici un handler ce urmează nu ar fi executat.

În cazul chain-urilor *din codul meu*, ideea *“reușirii de a-și aplica efectul”* este mai generală. Oare propagarea trebuie să fie oprită este verificat prin evaluarea proprietății Propagate a contextului, care poate or incapsula un câmp boolean ori să utilizeze o funcție pentru a calcula valoarea în dependența de valorile altor câmpuri din context. Încă, un chain poate fi *trecut fără a verifica propagarea*, adică trecut până la capăt independent de valoare lui Propagate.

A se vedea [testele pentru chain-urile](https://github.com/AntonC9018/hopper.cs/blob/86ca8afdfc40c3de04548f9d66e4738d8b86f9c6/.Tests/Core_Tests/Chain.cs).

### 3.2.2 Prioritatea

Ar putea să fie beneficiar ca handler-urile să aibă o prioritate și să fie sortate după acea prioritate. Aceasta ar face procesul de fixare a diferitor bug-uri legate de ordonarea execuției a handler-urilor mai ușor, dând mai multe flexibilitate chain-urilor.

Întrebarea este, ce structură de date să utilizăm pentru acest scop? Avem nevoie de inserții, ștergeri și căutări rapide, dar în același timp să putem itera prin colecția sortată după prioritate.

O idee ar fi să utilizăm o listă și s-o sortăm înainte de fiecare iterare prin ea. Este o soluție, însă problema este că adăugarea handler-urilor este lentă, și căutarea implică scanarea întregei liste.

O altă idee, care am utilizat-o la un moment, ar fi să utilizăm listele înlănțuite, sortându-le înainte de iterare. Însă: - sortarea listelor înlănțuite nu este frumoasă, - trebuie să avem a doua listă pentru a stoca handler-uri adăugate între iterații, ce ar fi pe urmă adăugate în lista principală.

Curent, utilizez un arbore binar balansat (SortedSet în C#). Ștergerea, inserția și căutarea sunt logaritmuce, și colecția mereu stă sortată. Încă am făcut ca prioritățile să fie unice pentru toate handler-urile în program, ca orice handler să poată fi identificat prin prioritatea sa.

#### 3.2.2.1 Cum prescriem prioritățile?

Prioritatea este prescrisă în funcția de inițializare generată automat, utilizând registrul pentru a genera numere de prioritate.

Am făcut o clasă specială pentru acest lucru, [priority assigner](https://github.com/AntonC9018/hopper.cs/blob/86ca8afdfc40c3de04548f9d66e4738d8b86f9c6/Core/Registry/PriorityAssigner.cs), care mapează *rangurile de prioritate* la *numere de prioritate*. [Rangurile de prioritate](https://github.com/AntonC9018/hopper.cs/blob/86ca8afdfc40c3de04548f9d66e4738d8b86f9c6/Shared/PriorityRank.cs) sunt următoarele: lowest, low, medium, high și highest și sunt definite într-un enum. Posibil am adăuga mai multe ranguri în viitor, însă pentru moment este suficient.

Când marcați un handler pentru export, puteți specifica un rang de prioritate. Handler-ul va primi o prioritate unică pentru acel rang la inițializare.

### 3.2.3 Tipuri de chain-uri

Introducerea priorităților a făcut toate chain-urile cele de prioritate, ceea ce în unele cazuri este o complicare excesivă. Deci, am adăugat LinearChain care este un chain fără priorități, elementele individuale din care nu trebuie să fie șterse, deoarece este modelat printr-o listă. Mai am definit SelfFilteringChain care utilizează un bufer dublu pentru a se filtra în timpul traversării, inserând elemenetele care trebuie să le pastrăm în al doilea bufer, pe urma schimbând buferele cu locuri. Am văzut deja un exemplu de utilizare în TriggerGrids.

A se vedea [implementarea lui DoubleList](https://github.com/AntonC9018/hopper.cs/blob/86ca8afdfc40c3de04548f9d66e4738d8b86f9c6/Utils/DoubleList.cs). A se vedea [implementarea differitelor tipuri de chain-uri](https://github.com/AntonC9018/hopper.cs/tree/86ca8afdfc40c3de04548f9d66e4738d8b86f9c6/Utils/Chains).

## 3.3 Entități și Componente

Entitățile sunt obiectele care **afectează logica jocului**. Exemple: *jucător*, *inamic*, *obiect din mediu*, *capcană*, *țigla de podea specială*.

Lucrurile care nu afectează logica jocului, ca *particule* sau *țiglele de podea simple*, *nu* sunt considerate ca entități. Acestea *nu sunt dirijate de către modelul*.

### 3.3.1 Structura entităților

Entitatea este doar un obiect cu un id și un dicționar de componente. Deci, ar fi echitabil să le numim simplu *containere pentru componente*.

Adițional, entitățile mele încă își stochează *id-ul tipului*, pentru a simplifica interacțiunea cu view-ul. Acest id al tipului este utilizat pentru sistemul itemilor (însă probabil voi schimba aceasta).

[A se vedea codul sursă](https://github.com/AntonC9018/hopper.cs/blob/86ca8afdfc40c3de04548f9d66e4738d8b86f9c6/Core/Entity/Entity.cs).

Cum puteți observa, clasa Entity este sealed, semnuficând că ea nu poate fi moștenită. Cum am discutat deja în [prezentarea generală a sistemului](#Xd546d8ab744d8137dabb35861d6cbc6c1bb0cc1), unicul mecanism utilizat pentru a atinge diversitatea proprietăților și a comportamentelor entităților este *folosirea componentelor*. Datorită acestuia, comportamentul sau proprietățile entităților pot fi augmentate prin aplicarea noilor componente sau prin eliminarea celor existente.

### 3.3.2 Ce componentele necesită pentru a funcționa?

Fiecare componentă are 2 tipuri de câmpuri:

1. *Câmpurile injectate*. Valorile acestora sunt transmite prin constructor, setate și deseori nu sunt schimbate niciodată. Copierea formei inițiale a unei componente ar însemna copierea tuturor valorilor injectate. De exemplu, nivelul entității este un câmp injectat pe componenta Transform.
2. *Alte câmăuri*. Acestea sunt necesare pentru a cuprinde *starea runtime* a componentei. De exemplu, flagurile pe comportamentul Acting indică dacă entitatea deja a făcut o actțiune în acest tur. Poziția și orientarea curentă pe Transform.

Deci, cum am stabilit, pentru a funcționa cum-se-cade, componentele trebuie să fie furnizate cu valorile pentru toate câmpurile injectate când componenta este instanciat.

Comportamentele definesc mai un tip de câmp, anume *chain-urile*. Chain-urile reprezintă mecanismul care permite să atingem *polimorfizmul*.

Al doilea punct important este faptul că comportamentele pot schimba comportamentul general a unei entități într-un mod. Acesta implică adăugarea funcțiilor handler pe chain-urile sale proprii sau pe cele definite de alte comportamente, deja prezente pe acea entitate.

De exemplu, Acting trebuie să-și reseteze flagurile când turul se termină. Pentru a atinge aceasta, adaugă un *handler de resetare* pe chain-ul de ticare a comportamentului Ticking.

Al treilea lucru sunt *retoucher-urile*. Retoucher-urile sunt handler-urile care sunt adăugate separat pe chain-uri specifice a unui comportament specific. Le-am menționat când am dat prezentarea generală a sistemului.

Aceste trei lucruri, anume instanțierea și adăugare comportamentelor, inițializarea handler-urilor comportamentelor și retușare definesc modul în care entitățile sunt fabricate.

### 3.3.3 Tipurile entităților

Prin “tipuri” în acest context nu avem în vedere “subclase,” deoarece, după cum s-a menționat anterior, componentele sunt utilizate în loc de moștenire. Tipurile specifice ale entităților sunt implementate diferit.

#### 3.3.3.1 Procedura în 3 pași

Deci, mai întâi să înțelegem în ce fel pot fi create tipuri. Posibilitățile:

1. Se rulează funcții pe entități care ar adăuga componente, retoucher-uri și le inițializa în modul corect.
2. Se creeză o reprezentare intermediară a tipurilor pe urmă folosite pentru a crea instanțe noi. Acest lucru a eșuat pentru mine, deoarece dacă utilizați așa strategie trebuie practic să creați două copii ale logicii Dumneavoastră.: una pentru entitatea reală și una pentru “fabrică.” Cu aceasta, aveți multe probleme de întreținere.
3. Un compromis între cele două metode: să avem o fabrică, care să augmenteze o entitate și la instanțiere să creeze doar copii ale acelei entități. Adică, vom avea funcții care ar configura fabrica, adăugând componentele adecvate și conectând handler-urile prin funcția Preset() respectivă. Pentru aceasta am optat în implementarea curentă, deoarece strategia este cea mai simplă din aceste trei.

Dacă luăm în considerare lucrurile menționate mai sus necesare pentru ca componentele să fie inițializate corect, ajungem la următoarea procedură de inițializare în 3 pași:

1. Se adaugă toate componentele / comportamentele.
2. Se execută funcțiile de inițializare (posibil vor fi disponibile mai multe funcții de inițializare pentru un anumit comportament, deoarece acestea ar fi utilizate în loc de subclasare pentru a obține pormorfizmul).
3. Se mai adaugă handler-urile (retușare).

Așadar, ideea mea a fost să definesc tipurile de entități ca clase statice cu 3 funcții statice, câte o funcție pentru fiecare pas, care ar augmenta subiectul fabricii pentru a construi inițial entitatea unui tip dat.

Iată [un exemplu de astfel de clasă statică](https://github.com/AntonC9018/hopper.cs/blob/86ca8afdfc40c3de04548f9d66e4738d8b86f9c6/TestContent/EntityTypes/Skeleton.cs). Acest exemplu ilustrează, de asemenea, modul în care se realizează un fel de moștenire a tipurilor: trebuie doar să apelați aceste 3 funcții ale tipului pe care doriți să-l moșteniți la pașii corespunzători, înainte de a le apela pe cele proprii.

Iată un exemplu al sintaxei pentru moștenire.

[EntityType(Abstract = true)]  
public static class BaseType  
{  
 public static void AddComponents(Entity subject) {/\*something\*/}  
 public static void InitComponents(Entity subject) {/\*something\*/}  
 public static void Retouch(Entity subject) {/\*something\*/}  
}  
  
[EntityType(Abstract = false)]  
public static class DerivedType  
{  
 public static EntityFactory Factory;  
  
 public static void AddComponents(Entity subject) {   
 Base.AddComponents(subject);   
 /\*încă ceva\*/   
 }  
 public static void InitComponents(Entity subject) {  
 Base.InitComponents(subject);   
 /\*încă ceva\*/  
 }  
 public static void Retouch(Entity subject) {  
 Base.Retouch(subject);   
 /\*încă ceva\*/  
 }  
}

#### 3.3.3.2 Problemele

Prezint câteva probleme asociate cu această abordare:

1. Implică boilerplate anevoios. Apelarea funcției tipului de bază la *fiecare* din pași, furnizarea corectă a semnăturilor pentru cele trei funcții statice, definirea unui câmp static de tipul EntityFactory sunt acele lucruri anevoioase.
2. Din același motiv, este predispusă la erori. Este ușor să uitați să apelați funcția de inițializare a unuia din comportamente și nu veți primi nici un mesaj clar în timpul dezvoltării ce ar indica această greșeală.

Cu toate acestea, nu îmi fac griji referitor la aceste probleme, deoarece eventual intenționez să *generez acest cod automat* din fișiere JSON cu descrieri de tipuri. Aceste fișiere JSON ideal vor fi modificate printr-un editor grafic sau cu ajutorul Intellisense-ului. Prin Intellisense mă refer de exemplu la sugerări ale valorilor adecvate pentru câmpurile injectate. Aceste valori vor fi obținute prin scanarea codului sursă, căutând tipuri sau câmpuri marcate cu atributele corespunzătoare.

De exemplu, componenta Transform are un câmp injectat de tipul Layers, care evident poate avea doar valori din enum-ul Layers. Deci Intellisense-ul le-ar putea sugera.

Ca un alt exemplu, considerăm componenta SlotComponent. Are nevoie de o valoare injectată de tip Slot. Slot nu este un enum. Deși este un tip cunoscut, sugestiile ar trebui să fie instanțe de acel tip. Spunem că astfel de instanțe sunt stocate ca câmpuri statice și sunt marcate cu atributul [Slot]. În acest fel, analizorul de cod le-ar putea registra și le poate sugera prin Intellisense.

Cu toate acestea, trebuie să facem **cu mult** mai mult lucru, pentru a implementa această funcționalitate, așa că, deocamdată, am decis să simplific și să definesc manual clasele statice. Sigur, așa abordare este predispusă la erori, dar în acest fel aș putea măcar să construiesc un prototip la timp.

### 3.3.4 Copierea componentelor

Prin copierea componentelor mă refer la copierea *chain-urilor* și a *câmpurilor injectate*, adică acelor părți ale componentei care sunt legate de tipul entității, ignorând valorile care s-ar schimba în timpul rulării.

#### 3.3.4.1 Pentru ce să copiem componente?

Acest lucru este esențial pentru inițializarea entităților. După după cum s-a menționat, pentru a inițializa o entitate, fabrica corespunzătoare trebuie să cloneze instanța de entitate “subiect” stocată.

De ce să nu rulați același set de funcții pe o entitate nouă de fiecare dată când o entitate de acest tip urmează să fie instanțiată în loc de a crea o instanță “subiect” întreagă? Cel puțin faptul că acest lucru ar anula posibilitatea de a optimiza copierea chain-urilor pe entitatea nouă să fie *leneșă*, trebuind să fie reconstituită în întregime de fiecare dată când avem nevoie de o nouă instanță, face această idee proastă.

#### 3.3.4.2 Implementarea

Pentru a copia orice obiect, aveți câteva opțiuni, dintre care unele au fost deja menționate:

1. Copierea prin reflecție (un fel de serializare automată);
2. Copierea prin generarea codului (furnizând strategia de copiere implicită pentru majoritatea componentelor);
3. Copierea manuală.

Deoarece toate componentele sunt cunoscute static în prealabil, nu avem nevoie de copiere prin reflexie.

Codul meu folosește pe larg copierea prin generarea codului, oferind constructori de copiere autogenerate și funcții de copiere, care, la rândul lor, folosesc acești constructori. Mai multe detalii despre generarea codului mai târziu.

Copierea manuală este uneori preferate față de generarea codului automată, deoarece, pentru unele componente, strategia de creare a unei copii ar putea fi prea complexă. Deci, generatorul de cod nu va furniza construcorul de copiere, dacă găsește unul specificat de utilizator.

Acum să înțelegem ce ar fi de fapt copiat de fiecare dată:

1. Valorile câmpurilor injectate. Acestea sunt copiate direct (prin referință, dacă este un tip de referință, sau făcând o copie shallow, dacă este un tip de valoare).
2. Chain-urile sunt întotdeauna copiate. Deoarece nu permit nici un closure pentru handler-uri, adică funcțiile nu sunt permise să capteze de exemplu componente, sarcina de copiere a acestora este banală, și anume, se copiează doar structura de date subiacentă. Aceasta poate fi, de asemenea, îmbunătățit în viitor utilizând un set sortat imuabil, astfel încât să fie copiat numai atunci când se schimbă handler-urile obiectului nou.

Faptul că valorile câmpurilor runtime nu trebuie să fie copiate duce la o altă idee interesantă. Am putea să construim un fel de *reprezentare de tip* a componentelor, adică să extragem versiunile prescurtate ale componentelor care ar avea doar câmpurile injectate și chain-urile. Aceasta este de fapt o idee destul de bună, deși singurul beneficiu pe care l-ar obține este că fabricile entității ar ocupa mai puțin spațiu în memorie (componentele nu ar avea câmpurile prezente în timpul rulării). Nu o fac pentru că ar fi greu de implementat și ar lua mult timp, cu beneficii minime.

### 3.3.5 Fabrica de entități

Fabricile de entități, după după cum s-a menționat, conțin o instanță “subiect” și scopul lor este de a crea copii la această instanță la cerere. A se vedea [implementarea](https://github.com/AntonC9018/hopper.cs/blob/86ca8afdfc40c3de04548f9d66e4738d8b86f9c6/Core/Entity/EntityFactory.cs).

Deoarece fabricile pot fi interogate prin ID-ul lor (având un ID de fabrică, este posibil să obțineți fabrica cu acel ID), trebuie să le atribuim aceste ID-uri și să le stocăm în registru. Voi prezenta mai multe teorii despre registru și identificatori mai târziu.

### 3.3.6 Wrapper-uri de entități

O altă idee ipotetică neimplementată interesantă este de a crea wrapper-uri asupra unor tipuri de entități specifice.

În prezent, generatorul de cod creează unele metode de extensie pentru clasa Entitate pentru recuperarea unei anumite componente dintr-o entitate. Deci, având orice entitate, chiar dacă acesta nu are componenta X, puteți executa entity.GetX() sau entity.TryGetX(). Acest lucru este rău atunci când vă așteptați la o entitate de un anumit tip, deoarece toate metodele irelevante, adică metodele care recuperează componente care nu ar exista niciodată pe acea entitate, ar crea dezordine printre metodele relevante. Acest lucru face explorarea API-ului prin Intellisense considerabil mai dificilă și este predispusă la erori.

Deci, considerăm o entitate de tip A care are componenta X. Alte componente, pe care entitatea de tip A nu le-ar avea, sunt Y și Z. Dacă definim un wrapper în următorul fel:

public readonly struct A\_EntityWrapper  
{  
 public readonly Entity entity;  
  
 public A\_EntityWrapper(Entity entity) => this.entity = entity;  
  
 public X GetX() => entity.GetX();  
}  
  
public void HypotheticalUsage()  
{  
 var entity = A\_EntityType.Factory.Instantiate();  
  
 // probabil se face automat de către fabrică, când ea returnează entitatea  
 var wrapped = new A\_EntityWrapper(entity);  
  
 wrapped.GetX(); // lucrează  
 // wrapped.GetY(); // nu lucrează  
 wrapped.entity.GetY(); // lucrează dacă știți că entitatea de fapt are Y   
}

Metodele GetY() și GetZ() nu vor aglomera imaginea, când știm că entitatea este de tipul entității A. Dacă, dintr-un anumit motiv, am decis că dorim să obținem Y de la entitate, putem face simplu wrapped.entity.GetY(); sau chiar să accesăm orice alte metode de extensie care au fost definite pentru entitate.

Deci, ar fi un wrapper *moale*, mai degrabă decât unul dur, ca atunci când moștenești o clasă.

Odată ce avem un mod de a stoca o reprezentare intermediară a tipului de entitate, adică ce componente vor fi prezente pe ea după ce fabrica entității a fost instanțiată, sarcina de a genera aceste clase de wrapper devine relativ trivială.

Totuși, această reprezentare intermediară este, de asemenea, o idee ipotetică la moment. Trebuie prea mult lucru pentru a implementa ideea în timp. Cu toate acestea, o astfel de reprezentare va fi esențială pentru metacompilarea JSON în clasele statice EntityType în C#, precum și pentru această idee. Deci, aceasta este cu siguranță o problemă care merită rezolvată în cele din urmă.

### 3.3.7 Încarcarea tiputilor de entități din JSON în timpul rulării

După după cum s-a menționat, definirea tipurilor de entități în JSON încă nu am implementat-o, dar este o îmbunătățire planificată.

Generatorul de cod ar metacompila aceste fișiere JSON cu descrieri ale tipurilor de entități în clase reale C# statice, care ar produce pe urmă fabrici de entități funcționale atunci când sunt rulate. Dar ar trebui, de asemenea, să fie posibilă încărcarea tipurilor de entități direct din JSON în timpul rulării, transformându-l imediat într-o fabrică de entități funcțională.

Pentru a nu redefini aceeași logică atât în generatorul de fabrică runtime de entități, cât și în metacompilator, poate fi utilizată o logică partajată care ar analiza fișierele JSON și ar genera o reprezentare intermediară a tipurilor de entități. Apoi, această reprezentare intermediară va fi utilizată direct pentru a genera o fabrică în timpul rulării, sau pentru a genera clase statice utilizând generatorul de cod.

Tipurile de entități runtime se pot dovedi utile în prototiparea tipurilor noi chiar în joc.

## 3.4 Acționare și bucla de joc

*Sistemul de acționare* este sistemul al cărui scop este de a permite entităților să interacționeze cu alte entități din lume.

### 3.4.1 Când și cum are loc acționarea

În prezent, atât *sistemul de acționare*, cât și *tick-area entităților* sunt dirijate direct de lume. Numesc subsistemul lumii responsabil pentru acționare și tick-area, “WorldStateManager” (un nume temporar). A se vedea [codul sursă](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Core/World/WorldStateManager.cs).

Acest sistem funcționează ținând evidența la toate comportamentele de acționare Acting și de tick-are Ticking prezente la un oarecare moment în joc. Comportamentele de acționare sunt stocate într-o matrice multidimensională, după *ordinea* lor. Ordinea specifică când ar fi apelat comportamentul de acțiune dat, printre altele. Deci, entitățile cu ordinea inferioară vor fi activate mai întâi și apoi vor fi cele cu ordinea inferioară. În prezent [există 4 ordine](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Core/Acting/Order.cs), ale căror comportamente actuale sunt activate unul după altul.

Lucrurile se fac în acest mod pentru a modela ceea ce trebuie să se întâmple în joc. Și anume, acțiunile inamicilor sunt executate după acțiunile jucătorului, după ce rândul se duce la obiectele din mediu înconjurător.

Comportamentele de tick-are sunt activate unul după altul, fără o ordine clar definită.

Ce se întâmplă exact când este activat comportamentul de acționare al unei entități vom examina mai târziu.

### 3.4.2 Bucla de joc

*Bucla de joc* reprezintă secvența de pași care se întâmplă în timpul unei ture. Ea include activarea tuturor comportamentelor de acțiune în ordine, urmată de activarea tuturor comportamentelor de tick-are. A se vedea [codul](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Core/World/WorldStateManager.cs#L38).

API-ul nu este complet. Noile *chain-uri globale* introduse vor fi utilizate pentru a implementa semnale pentru momentul în care se începe și se termină bucla de joc. *Chain-urile globale* vor fi discutate mai târziu.

### 3.4.3 Acționare

Sistemul de acționare este destul de complex.

Cele două etape distincte ale acestuia sunt:

1. Calcularea următoarei acțiuni, modelată prin strategia CalculateAction (o funcție);
2. Executarea următoarei acțiuni, modelată prin strategia ExecuteAction (de asemenea, doar o funcție).

Acești pași sunt executați separat de WorldStateManager și înainte ca oricare dintre entități să efectueze o acțiune. Deci, acțiunile luate de entități depind exclusiv de starea inițială a lumii la începutul buclei jocului. Rețineți, totuși, că acest lucru afectează doar anumite *tipuri* de acțiuni.

Deci, de exemplu, dacă un inamic decide să atace, nu va putea să se răzgândească după ce jucătorul de exemplu s-a mutat într-un loc nou. Poate schimba doar direcția în care urmează să atace. De asemenea, poate selecta o acțiune compusă, adică încercând mai întâi atacul, pe urmă mutarea, dacă atacul a eșuat.

Ambii pași implică apelarea unei funcții de strategie, deci ceea ce de fapt se va face este decis de acea funcție.

Funcția CalculateAction poate utiliza IA pentru a selecta următoarea acțiune sau poate lua acțiunea din input-ul utilizatorului sau poate să returneze aceeași acțiune de fiecare dată. Se injectează împreună cu instanțierea comportamentul de acționare.

ExecuteAction este puțin mai complicat. Pentru acțiunile prestabilite, unde direcția în care trebuie efectuată acțiunea este cunoscută în prealabil, cum ar fi cu input-ul utilizatorului, acțiunea ar putea fi setată să se execute pur și simplu în acea direcție. Complexitatea vine atunci când considerăm IA a inamicilor.

### 3.4.4 IA inamicilor

Există 3 aspecte de aceasta pe care trebuie să le discutăm:

1. Algoritmul de selecție a acțiunii (Sequence);
2. Algoritmul de execuție a acțiunii (EnemyAlgo);
3. Algoritmul de selecție a direcției (Movs).

#### 3.4.4.1 Sequence

Algoritmul de selecție a acțiunii pentru inamici este reprezentat de un obiect Sequence (secvență). Un obiect de secvență este în esență o listă de pași și un contor care arată pasul curent. Pașii în sine sunt fără stare, definind pur și simplu regulile de tranziție la alți pași din secvență.

De fiecare dată când acțiunea trebuie calculată, secvența returnează acțiunea asociată cu pasul curent. Când acțiunea respectivă a fost executată, pasul curent din secvență este ajustat în dependența de rezultatul acțiunii. A se vedea [codul sursă pentru Sequence](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Core/Acting/Sequence/Sequence.cs).

Iată [un exemplu simplu aș unei secvențe](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/TestContent/EntityTypes/Skeleton.cs#L19-L24). Să trecem peste el.

Deci, această secvență reprezintă IA a unui schelet simplu (zombi). Ideea este ca inamicul să se miște sau să atace jucătorul în fiecare tur. Deci ar ataca în prima rundă, nu face nimic în a doua, ataca, nu face nimic etc.

Specificăm o acțiune pentru primul pas al algoritmului ca compoziție de atac și mișcare scriind action = Compose(Attacking.Action, Moving.Action). Aceasta înseamnă că acțiunea de atac ar fi încercată mai întâi și, dacă nu reușește, mișcarea ar fi încercată.

Pe linia movs = Movs.Basic spunem că direcția ar fi selectată în după algoritmul movs Basic. Aceasta înseamnă atacarea sau deplasarea într-o direcție ortogonală către jucător.

Al doilea pas din secvență reprezintă acțiunea “a nu face nimic.”

Când al doilea pas se termină, secvența trece la primul pas și se repetă.

Iată [un exemplu mai implicat](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/TestContent/EntityTypes/Knipper.cs#L22-L53).

#### 3.4.4.2 Movs

Numesc algoritmul pentru selectarea unei direcții pentru acțiune un algoritm Movs.

În acest [exemplu de bază al unei secvențe](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/TestContent/EntityTypes/Skeleton.cs#L19-L24) a fost utilizat algoritmul Basic. Acest algoritm returnează direcțiile care v-ar apropia de jucător. De exemplu, dacă jucătorul ar fi direct în sus de inamic, doar direcția în sus ar fi returnată de algoritm. Însă, dacă jucătorul ar fi sus și în stânga inamicului, direcțiile în sus și în stânga ar fi returnate. Direcția cea mai aliniată cu orientarea curentă este returnată prima. Deci, pentru exemplul de mai sus, dacă inamicul se uită în sus, atunci ordinea de direcții returnate ar fi în sus și în stânga, însă, dacă se uită de exemplu în jos, ordinea returnată ar fi mai întâi în stânga și pe urmă în sus.

Folosesc matematica pentru a afla ce direcții să returnez. În primul rând, observăm că singurele direcții plauzibile returnate de algoritm coincid cu proiecțiile vectorului diferenței dintre poziția jucătorului și poziția inamicului pe axele x și y. Dacă una dintre proiecții este zero, adică există o singură direcție plauzibilă care ne apropie de jucător (în exemplul de mai sus această direcție era în sus), returnăm cealaltă proiecție non-zero, ceea ce este, de fapt, acea direcţie plauzibilă. În caz contrar, returnăm mai întâi acea proiecție, care este mai aproape aliniată cu orientarea curentă a inamicului, după ce returnăm și cealaltă. Factorul de “aliniere” poate fi definit matematic ca produsul scalar între vectorul de orientare și vectorul proiectat dat. A se vedea funcția Basic din [codul sursă](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Core/Acting/Movs/Basic.cs).

Există mai mulți algoritmi Movs predefiniți. A se vedea [codul sursă](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Core/Acting/Movs/Basic.cs).

Algoritmii de mișcare nu sunt complete din cauza existenței *facțiunilor*. În prezent, presupun că facțiunea vizată este facțiunea jucătorului, adică orice entitate care folosește funcțiile movs se presupune că vizează jucătorul.

De asemenea, deoarece jucătorii nu sunt în prezent în niciun fel memorate în cache, orice invocare a algoritmului movs implică căutarea prin toate entitățile din registru, ceea ce este foarte lent. În cele din urmă, voi începe să folosesc un fel de sistem de cache, pe care nu l-am conceput încă. Acest sistem ar trebui să fie similar cu ideea de *indici* pe tabelele bazei de date.

PS: Am început lucrul asupra acestui sistem. Deja am un prototip funcțional.

De notat: jocul *nu presupune că există un singur jucător*. Acest lucru se face pentru a permite multiplayer în viitor. Faptul că pot exista mai mulți jucători îi face pe inamici să caute cel mai apropiat jucător, în loc de a-l considera doar pe primul din registru.

#### 3.4.4.3 Enemy Algo

Algoritmul de execuție a acțiunilor inamicilor, sau pur și simplu *enemy algo*, a fost conceput pe baza următoarelor cerințe:

1. Acțiunea selectată reușește cel mult o dată;
2. Direcțiile în care trebuie încercată acțiunea sunt definite de algoritmul movs;
3. Dacă există o entitate care blochează una dintre acțiuni, entitatea respectivă ar trebui să-și facă mai întâi acțiunea.

Așa cum s-a explicat în secțiunea despre acțiuni, ele se consideră reușite dacă a reușit verificarea. Așadar, inamicii ar încerca acțiunea până când trece una din verificări și atunci s-ar opri.

De exemplu, dacă acțiunea este atacarea, mai întâi, țintele vor fi selectate și apoi, dacă țintele sunt goale, acțiunea ar eșua. Dacă lista țintelor nu a fost goală, acțiunea de atacare ar reuși.

Dacă o acțiune eșuează, la fel eșuează și direcția asociată acțiunii respective. În acest caz, următoarea acțiune disponibilă ar fi încercată. Dacă o acțiune reușește, algoritmul de execuție a acțiunilor se oprește.

Acum, ce se întâmplă dacă o acțiune eșuează deoarece un alt inamic a împiedicat-o să reușească? Oare acțiunea inamicul în final rămâne neexecutată, chiar dacă ar fi fost posibilă? (A se vedea o [descriere a acestei probleme](https://github.com/Zakru/opencrypt/issues/1#issue-457013204) pe opencript lui Zakru, un proiect similar cu al meu, dar care a fost abandonat).

Soluția mea la această problemă este de a face acea entitate să acționeze, dacă nu a făcut-o deja. Deci, atunci când decidem că de exemplu un alt incamic ne blochează calea, facem acel inamic să acționeze. După ce a acționat, ne încercăm din nou acțiunea.

Problema asociată acestei abordări este că inamicul pe care îl impunem să acționeze ar putea aștepta el însuși un alt inamic să acționeze în același timp. Ca rezultat, astfel am putea revine la prima entitate, creând un ciclu infinit de așteptare. Acest lucru este rezolvat în mod trivial prin setarea unui flag la comportamentul de acțiune, indicând dacă acesta este activat. Deci, înainte de a-l face pe celălalt inamic să se miște, verificăm dacă flagul respectiv nu este setat.

Ar fi beneficiar să mai adaugăm un flag care ar indica dacă entitatea dată și-a terminat procesul de acționare în turul acesta. Dacă fie acest flag, fie flagul care indică faptul că acționarea se face curent sunt setate, nu facem inamicul să se miște.

Cealaltă problemă, una mai mare, este cum să ne dăm seama ce entitate ne blochează acțiunea. Pentru acum, am optat pentru o abordare euristică simplă: entitatea în direcția curent încercată este solicitată să acționeze. În general, însă, această entitate nu ar fi cea care ne-ar împiedica să ne acționăm.

A se vedea [implementarea curentă](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Core/Acting/Algos/Enemy.cs).

### 3.4.5 Predicții

Predicțiile sunt necesare pentru a indica jucătorului acele celule din lume care, dacă jucătorul ar rămâne pe ele, l-ar dăuna. Acest sistem nu este încă cu totul complet în cod.

L-am implementat în așa fel că orice acțiune poate avea o funcție de predicție asociată, care ar returna astfel de poziții. Apoi, când vizualizarea dorește de exemplu să deseneze cruci la acele coordonate, ar itera prin toate entitățile, făcându-le să-și calculeze următoarea acțiune și apoi să utilizeze funcțiile de predicție ale acțiunilor calculate pentru a obține celulele periculoase pentru jucător. A se vedea [codul sursă](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Core/Acting/Predictions/Predictor.cs).

În prezent, API-ul nu este suficient de lustruit, dar funcționează deja cu atacuri și explozii.

### 3.4.6 Acțiuni

Action din cod reprezintă o acțiune concretă sau o succesiune de acțiuni care pot fi executate de o entitate din joc. Am menționat deja că acțiunile pot fi de 2 tipuri principale:

1. *acțiuni direcționate*, care necesită o direcție în care vor fi executate;
2. *acțiuni nedirecționate*, care nu necesită o direcție.

A se vedea [codul sursă](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Core/Acting/Action.cs).

Înainte ca acțiunea să fie executată, îi se asociază o direcție după ce ea este stocată ca CompiledAction. Atunci când este compilată o acțiune direcționată sau o acțiune nedirecționată, devine posibilă executarea acesteia direct, fără a furniza o direcție.

Acțiunile au fost implementate în cod în stil POO: avem o interfață pentru DirectedAction (IAction), o interfață pentru UndirectedAction și anumite clase care implementează aceste interfețe care reprezintă acțiuni: - SimpleAction, care acceptă o funcție care este chemată atunci când acțiunea este executată; - ActivatingAction, care activează componenta specificată IStandartActivateable pe entitate, pentru a executa acțiunea; - CompositeAction, care conține o listă de acțiuni, fiecare dintre acestea fiind încercată, iar execuția se oprește odată ce oricare dintre acestea reușește; - JoinedAction, care conține, de asemenea, o listă de acțiuni, dar execuția nu se oprește, chiar dacă o acțiune reușește. Acest lucru este util pentru definirea acțiunilor în doi pași, cum ar fi DieExplodeAction; - ConditionalAction, unde a doua acțiune stocată se face numai dacă prima reușește.

De asemenea, am definit mai multe funcții ajutătoare pentru instanțierea rapidă a acțiunii dorite fără prea mult boilerplate.

După după cum s-a menționat, acțiunile pot conține, de asemenea, o funcție de predicție. În acest caz, ar trebui să implementeze interfața IDirectedPredictable sau IUndirectedPredictable.

#### 3.4.6.1 Înlocuirea acțiunilor

Este posibil ca o entitate să facă o altă acțiune în locul celei alese. De exemplu, atunci când o entitate alunecă, ea nu ar trebui să poată face acțiuni direcționate. Acest lucru se realizează prin înlocuirea acțiunii selectate de jucător prin mișcarea în direcția de alunecare și efectuarea acțiunii inițiale a jucătorului numai dacă alunecarea eșuează. A se vedea [codul sursă](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/TestContent/Modifiers/Sliding/SlidingEntityModifier.cs#L55).

Poate modul în care am realizat această funcționalitate nu este corect. Poate am trebui s-o înlocuiesc prin ajustarea acțiunii atunci când ea este calculată, deoarece dacă lăsăm așa cum este acum, predicțiile ar devine uneori incorecte. Adică, dacă inamicul alunecă, în același timp încearcând să atace, celula periculoasă pentru jucător ar fi totuși evidențiată pe ecran, cu toate că nu trebuie. De fapt, această soluție nu este greu de implementat în codul curent.

PS: Am început lucrul și asupra acesteia.

## 3.5 Registru

Ideea unui *registru* este esențială pentru *implementarea serializării, multiplayerului online* și a *modurilor*.

În acest moment nu mă preocupă primele două, însă implementarea suportului pentru moduri este una din obiectivele mele inițiale pentru proiect, așa că nu o voi ignora cu totul.

### 3.5.1 Funcția unui registru

Un registru poate fi utilizat pentru a atribui identificatori *conținutului* și pentru a furniza o mapare de la identificatori la conținutul corespunzător. Conținutul poate fi orice lucru care necesită un identificator, cum ar fi tipurile de entități, tipurile de statistici, tipurile de itemi, handler-urile (au nevoie de prioritate, gestionate tot de registru), tipurile de componente etc.

Registrul poate fi, de asemenea, utilizat pentru a atribui *identificatori de runtime*, care sunt utilizați pentru a identifica *instanțe*, cum ar fi entitățile. O mapare este, de asemenea, furnizată de la un identificator cunoscut al instanței de entitate cu acel identificator.

### 3.5.2 Cazuri de utilizare

#### 3.5.2.1 Serializare

Vom considera sarcina de *serializăre*. *Serializarea* înseamnă salvarea stării curente a jocului într-un fișier și *deserializarea* înseamnă restabilirea stării respective înapoi în joc. Imaginați-vă că jucătorul a progresat cu jumătate de nivel și apoi a decis să oprească jocul. Când ar porni jocul data viitoare, nu ar mai putea continua de unde s-a oprit ultima dată, dacă starea jocului nu a fost cumva salvată.

Acestea sunt diferitele abordări privind serializarea:

1. Se salvează un bloc de memorie, cu toate referințele și pointeri. Desigur, toți pointerii vor trebui să fie relative la, de exemplu începutul blocului de memorie, deoarece dacă sunt absolute, cel mai probabil nu vor indica memoria corectă după deserializare. Această abordare este una destul de elegantă, însă problema este că ea are nevoie de un control ne nivel scăzut asupra memoriei. Deoarece memoria din C# este gestionată de către runtime, nu avem acest control, deci nu cred că această strategie este aplicabilă în C#.
2. Se serializează datele în, de exemplu, JSON prin utilizarea reflecției, după ce ele se deserializează din JSON prin instanțierea claselor potrivite și setarea corectă a tuturor proprietăților prin reflexie. Există biblioteci care ar ajuta cu acest lucru, cum ar fi Newtonsoft.Json.

O problemă obișnuită asociată acestei abordări este faptul că o entitate ar putea stoca o referință la o altă entitate (să o numim țintă). Datorită acestui fapt, ținta ar fi deserializată de două ori (ca și cum ar fi două entități diferite). Când starea jocului ar fi fost deserializată, entitatea ar avea acum o referință la o copie a țintei, în loc de o referință la ținta reală. Acest lucru este nedorit, deoarece aceasta ar însemna că statul jocului nu a fost capturat corect.

De exemplu, spuneți că inamicul a stocat o referință la jucător într-un câmp. Starea jocului a fost serializată și apoi deserializată. Acum sunt doi jucători, dintre care unul este jucătorul propriu-zis iar altul este stocat în acel câmp al inamicului.

Această problemă este adesea rezolvată de aceste biblioteci, ținând evidența tuturor obiectelor la care se face referire și atribuind identificatori celor repetați. Când se întâlnește o referință la o entitate de a doua oară, aceasta este înlocuită cu acel identificator în rezultatul serializării. Acest lucru este OK în cazuri simple, dar ce se întâmplă dacă doriți să serializați handler-uri? Funcțiile nu sunt serializabile în C#, deci abordarea automată nu ar funcționa.

Ceea de ce aveți de fapt nevoie este să puteți avea un sistem care să atribuie un identificator (prioritate) fiecărui handler, la inițializare. Apoi, definiți serializarea ca salvarea pur și simplu a acelui identificator. Definiți deserializarea ca mapare a numărului de prioritate citit din, de exemplu, JSON în handler-ul propriu-zis. Cu această abordare, puteți permite chiar și utilizarea closure-urilor (desigur, closure-urile pentru ceva de a face cu conținutul static; closururile pentru, de exemplu, entitățile, sunt oricum interzise).

De fapt, acesta este motivul pentru care closure-urile în handler-uri nu sunt permise. Cum ați serializa un handler definit în timpul rulării? Deoarece în C# nu se permite instanțierea unui closure de un tip anonim corect, ar trebui să definiți tipuri de închidere specifice dacă doriți ca acestea să fie serializate, ceea ce ar fi anevoios (prea multe clase, sintaxa ar fi prea detaliată). Mai mult, amestecarea tipurilor de closure specifice, definite de utilizator ca clase, și cele anonime este complicată. Acesta este, de fapt, un neajuns al C#, deoarece nu este greu de implementat, cel puțin conceptual. Oricum, este așa cum este.

Deci, în codul meu, spun că codul care dorește să utilizeze închideri peste obiecte runtime (cum ar fi entități) ca handler la evenimente (chain-uri), trebuie să le atașeze fie în constructor (fie într-o funcție de inițializare), fie să le facă temporare, deoarece aceste handler-uri nu ar fi serializate.

Același sistem poate fi aplicat la oricare conținut static, cum ar fi tipurile de entități.

Nu am abordat serializarea cum-se-cade, deci aceste idei sunt în mare parte speculative.

#### 3.5.2.2 Multiplayer

Cum ar putea spune serverul ce player este asociat cu ce adresă IP? Cum ar semnaliza serverul către ceilalți clienți ce item a obșinut un anumit jucător sau ce acțiune au selectat etc.?

Fiecare jucător, item sau entitate ar avea un identificator, pe care serverul l-ar folosi în pachete de date, care ar conține de exemplu acțiunea selectată de un alt jucător împreună cu identificatorul lui, sau itemul care a fost obținut etc.

Fără identificatori, serverul nu ar putea transmite clienților informații despre jucători sau orice alte entități, deoarece nu poate transmite referințe la obiecte prin pachete. Singura modalitate de a referi la un obiect menționat în pachet este prin transmiterea identificatorului său și apoi maparea acelui identificator la o referință pe partea clientului.

Deci, registrul este esențial pentru multiplayer.

### 3.5.3 Stocarea și accesare componentelor

Am menționat anterior că componentele unei entități sunt stocate într-un dicționar, însă nu am subliniat după ce cheie sunt stocate.

O idee este să folosiți TypeInfo ca cheie. De fapt, am folosit-o la început. - În primul rând, acest lucru este nefast și nu-mi place această abordare. - Nu ar funcționa pentru un sistem în care aveți mai multe instanțe de același tip, care totuși trebuie să fie cumva identificate și stocate într-un loc central pentru a permite maparea, cum ar fi tipurile de componente. - Dacă permiteți vreodată mai multe componente de același tip să existe simultan pe o entitate, decizia de a folosi TypeInfo ca cheii ar avea efecte nefaste în viitor.

În mod clar, ar trebui să se dezvolte o modalitate de identificare a unei colecții de tipuri ca aceasta.

Astfel, am optat pentru *indici* (în alte sisteme, acestea se mai numesc și *referințe*). În opinia mea, indicii sunt o idee bună. Ei încapsulează atât tipul obiectului asociat identificatorului, cât și identificatorul în sine. A se vedea [codul sursă](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Core/Components/Index.cs).

Acest lucru este util, deoarece permite rezolvarea automată a genericilor, ceea ce ajută la evitarea cast-urilor explicite și, prin urmare, a erorilor din cod.

De exemplu, să considerăm că entitatea are următoarea metodă generică de recuperare a componentelor (presupunem că acestea sunt stocate după tip, ceea ce, după cum s-a stabilit, nu este maxim flexibil):

T GetComponent<T>() where T : IComponent  
{  
 return (T) components[typeof(T)];  
}

Ați utiliza codul asemănător la acesta:

entity.GetComponent<Attacking>();

Utilizând abordarea mea cu indicii, se schimbă la:

T GetComponent<T>(Index<T> index) where T : IComponent  
{  
 return (T) components[index.Id];  
}

Iar utilizarea devine:

class Attacking : IComponent   
{   
 // Acestui indice se atribuie un identificator de către registru  
 static Index<Attacking> Index;   
}  
  
entity.GetComponent(Attacking.Index);  
  
// Țineți minte că genericul aici se rezolvă, deoacere indicele are informație despre tip.  
// Exemplul de fapt invocă următoarea metodă:  
entity.GetComponent<Attacking>(Attacking.Index);

Dacă ați păstra doar identificatorul fără a-l include într-un indice, componenta ar fi recuperată în felul următor:

entity.GetComponent<Attacking>(Attacking.Identifier);  
// Sau, și mai rău, dacă GetComponent() nu este generică  
var attacking = (Attacking) entity.GetComponent(Attacking.Identifier);

Abordarea cu indici este mai bună, deoarece oferă anumite garanții cu privire la tipul stocat de identificator și nu necesită casturi sau completarea explicită a argumentului generic.

De asemenea, cu ajutorul indicilor este posibil să se decidă cum să se recupereze componentele în mod diferit în funcție de tipul de indice din care provine identificatorul lor:

interface IComponent {}  
interface ISpecial {}  
  
class PlainComponent : IComponent { static Index<PlainComponent> Index; }  
class SpecialComponent : IComponent, ISpecial { static SpecialIndex<SpecialComponent> Index; }  
// ...  
// Definim 2 metode supraîncărcate, luând diferite tipuri de indici.  
// Nu puteți supraîncărca o metodă cu aceleași parametri, chiar dacă interfețele sunt diferite.  
// Nu știu de ce ei au impus această restricție peste C#.   
// Unica modalitate de a atinge metodele de stocare diferite este să utilizăm tipurile de indici diferite  
// Din fericire, așa tipuri pot fi generate prin creare subtipurilor lui Index<T> fără membrii noi  
T GetComponent<T>(Index<T> index) where T : IComponent   
{  
 return (T) components[index.Id];  
}  
T GetComponent<T>(SpecialIndex<T> index) where T : IComponent, ISpecial   
{  
 return (T) specialComponents[index.Id];  
}  
// ...  
PlainComponent plain = entity.GetComponent(PlainComponent.Index);  
SpecialComponent special = entity.GetComponent(SpecialComponent.Index); // Evocă metoda supraîncărcată.

Implementarea reală este destul de similară, a se vedea [codul sursă relevant](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Core/Entity/Entity.cs#L41-L51).

Există o problemă aparentă cu această abordare: fiecare componentă ar conține boilerplate-ul acelui câmp de index static și fiecare componentă ar trebui să primească un identificator din partea registrului. Am reușit să rezolv această problemă prin generărarea codului.

### 3.5.4 Moduri

Să presupunem că doriți să încărcați o anumită combinație de moduri. Fiecare mod definește un anumit conținut nou (presupunem ordinea intițializării conținutul constantă). Presupunem, de asemenea, că nu există dependențe circulare dintre moduri. Urmând terminologia mea, *conținutul* unui mod este definit ca o colecție de tipuri, care au fiecare un identificator unic în cadrul categoriei lor.

Ideea mea inițială a fost că, odată ce un tip (de exemplu, un tip de entitate) a fost instanțiat, acesta ar primi un identificator global unic, care ar fi salvat în *registrul global*. Acum, dacă presupunem că conținutul unui mod este construit static, adică tipurile sunt salvate ca și câmpuri statice pe unele clase definite de mod, problema devine că acesta nu poate fi reinitializat ulterior. Odată ce un mod a fost încărcat, ID-urile nu pot fi realocate, iar tipurile nu pot fi recreate. Acest lucru înseamnă că descărcarea și reîncărcarea modurilor este inconsecventă.

De exemplu, aveți un mod A care a definește 4 tipuri, care au primit identificatorii de la 0 la 3. Acum încărcați un nou mod, numit B, care la rândul său definește 4 tipuri. Acestea primesc identificatorii de la 4 la 7. Acum doriți să descărcați modul A. Deoarece nu există niciun mecanism de reîncărcare sau de modificare a id-urilor, tipurile din modul B vor avea în continuare id-urile lor anterioare. Acest lucru este rău, deoarece, deși avem un singur mod, mod B, dacă am fi avut doar B de la început, id-urile ar fi fost diferite, deoarece tipurile ar fi avut în schimb identificatorii de la 0 la 3. Aceasta o am numi *inconsecvență*.

Așadar, cel mai simplu mod de a remedia această problemă este ca fiecare mod să definească *lista de conținut* pe care o creează și o funcție *init* (de intițializare), care poate fi rulată din nou pentru a realoca identificatorii și a-i citi în registru.

Să presupunem că avem modurile A și B. Acum, dacă am dori să descărcăm A, am șterge totul din registru și am redefini B, folosind funcția sa de inițializare.

Acest lucru ar elimina, de asemenea, problemele cu serializarea (save-uri și server).

Să presupunem că avem o salvare care a fost jucată cu modurile A și B. Acum, dacă ordinea în care au fost definite A și B ar fi fost inconsecventă, identificatorii ar putea fi încurcate în sensul că după reîncărcarea jocului, conținutul din cele două moduri ar primi alți identificatori decât cele stocate în save-ul și astfel s-ar încărca incorect.

Acum, în cazul serverului, să presupunem că jucătorul activat un număr mai mare de moduri decât serverul sau că identificatorii tipurilor nu corespund cu identificatorilor tipurilor corespunzătoare de pe server. În acest caz, dacă am dori să trimitem informații care se referă la un anumit tip de la server la client, ar trebui să urmărim maparea identificatorilor de la server la client, ceea ce, cred, este complicat. În schimb, atunci când ne conectăm la un server, verificăm dacă modurile corespund, iar dacă nu, descărcăm toate modurile și le reîncărcăm doar pe cele de care are nevoie serverul. Bineînțeles, acest lucru nu ar trebui să afecteze modurile care se referă doar la grafică, de exemplu, cele care schimbă texturile default.

Această abordare ar necesita ca modurile să definească în mod explicit tot conținutul lor și să enumere dependențele lor. Acest lucru este facilitat de generatorul de cod, care creează automat o astfel de funcție init.

### 3.5.5 Identificatori

*Identificatorul* a fost menționat de multe ori, dar nu s-a precizat niciodată cum arată exact.

Un identificator este, în esență, un număr unic. Pentru orice conținut definit de moduri (inclusiv modul Core), am decis să folosesc unul care să conțină un număr întreg pe 32 de biți care să indice numărul modului și un număr întreg pe 32 de biți care să indice numărul secvențial al tipului dat. A se vedea [Indentifier în codul sursă](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Core/Registry/Identifier.cs).

Identificatorul *runtime* este utilizat pentru identificarea *instanțelor* entităților. În codul actual am optat pentru un singur număr întreg pe 32 de biți, dar acesta poate fi modificat. A se vedea [RuntimeIdentifier în codul sursă](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Core/Registry/RuntimeIdentifier.cs).

Registrul atribuie identificatorii în mod secvențial. Modurile, de asemenea, primesc identificatori secvențiali, deși ar fi benefic ca ele să-și specifice prin hardcode numărul lor de mod. Iată [cel mai simplu identificator](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Core/Registry/IdentifierAssigner.cs), fără mapare. Iată un [subregistru pentru orice categorie specifică de tipuri, cu mapare](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Core/Registry/StaticRegistry.cs).

În prezent, codul de registru nu l-am perfecționat. Nu există nicio modalitate de a defini categorii personalizate de tipuri, ceea ce va fi cu siguranță util în viitor.

### 3.5.6 Înregistrarea flagurilor

Înregistrarea flagurilor este neimplementată în momentul curent.

Ideea este ca un tip de indicator să fie extensibil în timpul execuției. De exemplu, există un enum Faction enum, care reprezintă facțiunea unei anumite entități. Aceasta poate fi Player, Enemy sau Environment, care ar avea valorile 001b, 010b și, respectiv, 100b. Acum, imaginați-vă că un mod a decis să adauge mai multe facțiuni, cum ar fi RedTeam sau BlueTeam. Singura modalitate de a face acest lucru pe care o au este de a codifica în hardcode biți de flaguri specifice pentru acestea și de a spera că nici un alt mod nu a decis să folosească aceiași biți pentru flagurile facțiunilor ale *lor*.

Pentru a se asigura că flagurile nu se ciocnesc niciodată, este necesar să facem ca registrul să atribuie și să distribuie noi flaguri. Evident, dacă numărul de flaguri deja înregistrate este mai mare decât dimensiunea numărului întreg, nu ar putea fi adăugate noi flaguri, dar cred că aceasta nu se va întâmpla niciodată. Așadar, 32 sau 64 de flaguri este limita, în dependența de dimensiunea selectată pentru numărul întreg care reprezintă tipul.

## 3.6 Generarea codului

*Generarea codului* este un instrument foarte util pentru: - a elimina boilerplate, făcând astfel baza de cod mai ușor de întreținut; - a face sistemul mai prietenos față de refactorizare, încurajând dezvoltarea treptată; - facilitarea definirii sau asigurarea unei definiții automate a anumitor tipuri sau funcții din cod; - asigurarea integrării cu instrumente externe, cum ar fi un plugin pentru editor.

Cele două instrumente principale pe care le folosesc pentru generarea de cod sunt **T4** pentru generarea de text din șabloane și **Roslyn** pentru analiza de cod.

### 3.6.1 T4 (Text Template Transformation Toolkit)

**Text Template Transformation Toolkit** (denumit de obicei “T4”) este un framework de generare de text bazat pe șabloane, gratuit și open-source. Fișierele sursă T4 sunt denumite, de obicei, prin extensia de fișier “.tt.”

T4 este utilizat de dezvoltatori ca parte a unei aplicații sau a unui cadru de instrumente pentru a automatiza crearea de fișiere text cu o varietate de parametri. Aceste fișiere text pot fi, în cele din urmă, orice format de text, cum ar fi codul (de exemplu, C#), XML, HTML sau XAML.

T4 utilizează un format de șablon personalizat care poate conține cod .NET și șiruri literale în el, acesta fiind analizat de instrumentul de linie de comandă T4 în cod .NET, compilat și executat. Rezultatul codului executat este fișierul text generat de șablon. T4 poate fi, de asemenea, rulat complet în cadrul aplicațiilor .NET prin utilizarea clasei TextTransformation, ceea ce elimină necesitatea ca utilizatorul final să aibă instalat Visual Studio.

[Sursa: wiki](https://www.wikiwand.com/en/Text_Template_Transformation_Toolkit).

#### 3.6.1.1 De ce T4?

Folosesc T4 pentru a genera cod C# pentru proiect, pe baza informațiilor din fișierele JSON sau extrase din codul sursă.

Motivele pentru care folosesc T4: - Se integrează cu alt cod C#, ceea ce a fost o cerință în cazul meu, deoarece o parte din codul dintre generatorul de cod și aplicație urma să fie partajat și nu voiam să îl scriu în 2 limbaje diferite; - T4 a fost prima și singura opțiune pe care am încercat-o. Nu este deloc ideală, dar își face treaba; - Scrierea unui motor de modelare suficient de puternic ar implica prea multă muncă, de aceea mă bucur că am reușit să o evit.

#### 3.6.1.2 Un exemplu simplu

Șabloanele T4 includ logica de control, scrisă în cadrul <# ... #>, împreună cu textul care urmează să fie inserat.

<# if (X == 2)   
 { #>  
This text will be in the output if X is 2.  
<# }  
 else  
 { #>  
Otherwise, this text will be output.  
<# } #>

Rularea șablonului cu X = 3 produce următorul text ca output:

Otherwise, this text will be output.

Mai este posibil să afișăm valoarea unei variabile:

X at generation time was <#= X #>

Rularea șablonului cu X = 4 produce următorul text ca output:

X at generation time was 4

Practic, aceasta este esența lui T4. Desigur, este puțin mai complicat în practică, deoarece acest X trebuie definit și transmis cumva șablonului, dar este doar un detaliu.

### 3.6.2 Roslyn (.NET Compiler Platform)

**Roslyn** oferă un set de API-uri pentru analiza lexicală a codului, analiza semantică și sinteza codului. Acesta poate citi și analiza codul dintr-un întreg proiect, oferind programatorului acces la arborele sintactic și la modelul semantic.

Modelul semantic este deosebit de util. Acesta permite obținerea cu ușurință a informațiilor despre *simboluri*: unde a fost definit un anumit tip în sursă, găsirea namespace-urilor sau tipurilor unde a fost definit, găsirea tuturor referințelor la un acest simbol, inspectarea argumentelor generice, găsirea și evaluarea atributelor etc.

#### 3.6.2.1 De ce Roslyn?

Folosesc Roslyn pentru citirea și analiza codului din proiectul *Core*, precum și din orice proiect al unui mod pentru a obține anumite date din codul sursă, oferind aceste informații lui T4 pentru a genera codul necesar.

* Roslyn este un framework pe C#, ceea ce este o cerință pentru mine, deoarece o parte din cod este partajată, așa cum s-a menționat;
* Scrierea propriului meu framework pentru analiza codului C# este mult mai mult de lucru decât chiar scrierea propriului meu motor de șabloane;
* Nu cred că există opțiuni mai bune pentru analiza codului în .NET.

### 3.6.3 Elemente de bază

Să deslușim modul în care se integrează generarea de cod în proiectul meu.

Primul lucru care trebuie menționat este că proiectul se bazează foarte mult pe funcțiile și tipurile definite în codul generat. Astfel, nu este posibil nici măcar să compilați proiectul fără a rula mai întâi generatorul de cod.

Funcțiile și tipurile sunt generate fie: - dintr-un alt limbaj (din fișiere JSON), care utilizez datorită sintaxei mai ușoare sau pentru a permite comunicarea cu alte instrumente; - prin inspectarea codului sursă și prin decizia de ce trebuie generat prin analiza acestuia.

În prezent, prima strategie este folosită doar pentru *stats*, care sunt atinse destul de puțin în această lucrare, dar în viitor va fi folosită și pentru *tipuri de entități*. Prima strategie este, într-un fel, mai dificilă decât a doua, deoarece sarcina de a înțelege fișiere JSON și de a crea un fel de model semantic este pe umerii mei de programator.

Pașii pentru generarea de cod din fișiere JSON trebuie să fie evidente:

1. Programatorul scrie fișiere JSON valide, conform unor reguli, prin care generatorul de cod (metacompilator) să analizeze datelor din interiorul acestora și să le considere ca sursă pentru generatorul de cod;
2. Citirea și înțelegerea conținutului fișierelor JSON care urmează să fie *metacompilate*;
3. Adunarea informații din fișierele JSON analizate și prezentarea lor la un șablon T4;
4. Generarea codul folosind T4.

A doua strategie este doar puțin diferită:

1. Programatorul fie își adnotează codul sursă C# în conformitate cu anumite reguli, de exemplu, prin utilizarea atributelor, fie doar scrie cod, pe care generatorul de cod l-ar prelua automat, de exemplu, implementarea lui IComponent ar trebui să permită automat generarea de cod;
2. Utilizarea lui Roslyn pentru a citi, procesa și analiza codul sursă scris de programator;
3. Colectarea informațiilor din, de exemplu, modelul semantic și prezentarea lor lui T4;
4. Generarea codului cu ajutorul lui T4.

### 3.6.4 Șabloane

În această secțiune aș dori să împărtășesc câteva sfaturi și descoperiri pe care le-am făcut în timp ce lucram cu șabloanele T4.

Lucrul cu șabloanele este, în mare parte, destul de simplu, iar modul în care acestea funcționează este și el transparent.

Să considerăm un șablon de text, care este un amestec de cod de control și textul care urmează să fie tipărit. Acesta este mai întâi compilat în 2 clase C#: - O clasă de bază, care implementează [*interfața duck* necesară pentru generarea codului](https://docs.microsoft.com/en-us/dotnet/api/microsoft.visualstudio.texttemplating.texttransformation?view=visualstudiosdk-2019). Aceasta implică definirea unei metode virtuale sau abstracte TransformText(), a metodelor Write() și WriteLine() pentru afișarea textului și mai niște lucruri; - Clasa de afișare propriu-zisă, care moștenește din clasa de bază, care transformă codul de control în cod C# real, iar textul care urmează să fie afișat în apeluri la Write() și WriteLine(). Acesta inserează tot acest cod de afișare în metoda suprascrisă TransformText(). Deoarece sunt lipite în scopul clasei de afișare, ele pot accesa orice câmpuri sau proprietăți definite în aceasta. Acest lucru este deosebit de util, deoarece clasa generată este parțială, deci este ușor de adăugat mai multe câmpuri sau proprietăți.

De asemenea, este posibil să definiți propria clasă de bază, care să implementeze interfața duck.

Este posibilă împărțirea bucăților de cod T4 comune mai multor șabloane în fișiere txt și includerea lor în șabloane în timpul compilării. Am folosit această abordare pentru a insera, de exemplu, o remarcă că codul a fost autogenerat în partea de sus a fiecărui fișier generat, remarca însăși fiind preluată dintr-un [fișier txt cu textul ei](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Meta/Templates/Snippets/autogen_notice.txt).

Am avut unele probleme cu indentarea, pe care le-am rezolvat folosind PushIndent(), PopIndent() și o modificare șmecheră a metodei Write(). A se vedea [aceast post pe stackoverflow unde am împărtășit soluția mea](https://stackoverflow.com/questions/67561998/t4-indent-code-included-from-another-file).

Încerc să păstrez logica mea separată de șablonul de text, pentru a păstra șablonul mai simplu. Definesc proprietăți, metode sau câmpuri de ajutor pentru componentele mele logice, referințe pe care le includ ca și câmpuri în clasa specială a clasei de afișare. A se vedea, de exemplu, [AllInitPrinter](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Meta/Templates/Printers/AllInitPrinter.cs) și [proprietățile definite în una dintre clasele de bază din codul logicii](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Meta/Templates/Logic/Components/TypeSymbolWrapperBase.cs#L145-L155).

### 3.6.5 Atribute

Atributele reprezintă facilitatea principală de marcare a unor bucăți de cod ca ele să fie luate în considerare de Roslyn. T4 ar trebui apoi să primească aceste informații sub o anumită formă pentru a genera codul necesar.

Pentru o listă completă de atribute, a se vedea [acest fișier sursă](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Shared/Attributes.cs).

Să ne uităm la câteva exemple.

#### 3.6.5.1 FlagsAttribute

Probabil cel mai ușor de înțeles și cel mai independent atribut este atributul Flags. Acesta este un atribut personalizat pe care l-am definit pentru a marca un enum pentru care să fie generat codul. În special, se consideră că enum-ul marcat definește un set de flaguri.

Iată cum aș marca un enum cu acest atribut:

[Flags]  
enum MyFlags  
{  
 Colored = 1,  
 Tasty = 1 << 1,  
 Warm = 1 << 2  
}

Când codul meu Roslyn analizează codul sursă dat, găsește toate enum-urile marcate cu acest atribut și încapsulează simbolul [într-o clasă specială](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Meta/Templates/Logic/FlagEnumSymbolWrapper.cs), care expune toate informațiile relevante despre simbol șablonului T4.

Șablonul T4 [arată în felul următor](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Meta/Templates/FlagsPrinter.tt).

Rulând acest șablon pe baza informațiilor recuperate din codul nostru, motorul de modelare produce următorul cod:

namespace Hopper.Core  
{   
 public static class MyFlagsFlagsExtensions  
 {  
 /// <summary>  
 /// Checks whether the given flags intersect with the other flags.  
 /// Returns true if either of the other flags are set on the flags.  
 /// To see if flags contain all of some other flags, use <c>HasFlag()</c> instead.   
 /// </summary>  
 public static bool HasEitherFlag(this MyFlags flag1, MyFlags flag2)  
 {  
 return (flag1 & flag2) != 0;  
 }  
  
 // mai multe funcții ...

Oricum, ce rost are aceasta?

Ideea este că este mult mai ușor de utilizat funcția HasEitherFlag(), decât (flag1 & flag2) != 0, însă nu poate fi definită în mod adecvat ca o funcție generică pentru un oarecare enum de flaguri. Prin urmare, ar trebui să definim o astfel de funcție pentru orice enum de flaguri pe care l-ar adăuga în viitor. Pentru a păstra calitatea înaltă a codului, ar fi trebuit să scriem manual același comentariu summary pentru fiecare funcție de acest tip. Cu alte cuvinte, prea mult boilerplate.

Dacă acest cod este generat automat, economisim timpul de a scrie funcții similare, care ar trebui să fie disponibile în mod ideal pentru fiecare enum de flaguri. Dacă dorim ca aceste funcții similare să se schimbe toate deodată pentru a se adapta la noi condiții sau cerințe, trebuie doar să modificăm șablonul T4 și tot codul generat se schimbă automat. Credeți-mă, acest lucru economisește mult timp și nervi.

#### 3.6.5.2 AliasAttribute

Acesta este un alt exemplu ușor de înțeles.

Scopul atributului Alias sau, în general, scopul definirii de alias-uri pentru anumite *metode* ale componentelor este de a exploata un anumit pattern comun în cod și de a-l înlocui cu un cod mai puțin verbos. În special, pattern-ul constă în obținerea unei componente de la o entitate și apelarea imediată a unei metode a acesteia cu anumite argumente, posibil și transmițând însăși entitatea ca argument.

De exemplu, să luăm metoda Activate() a comportamentului Moving. Aceasta primește un argument de tip Entity, care reprezintă actorul care face mișcarea, adică entitatea de la care a fost luată componenta Moving, și o direcție. Cod simplificat (corpul funcției a fost omis pentru simplitate):

class Moving { void Activate(Entity actor, IntVector2 direction) {} }

Deci, pentru a activa comportamentul Moving, adică, pentru a evoca această metodă corect, ați trebuie să scrieți codul de felul următor:

actor.GetMoving().Activate(actor, IntVector2.Right);  
// Pentru a încerca a se mișca, utilizați versiunea TryGet în loc de Get. Prea verbos, nu vă pare?  
if (actor.TryGetMoving(out var moving)) moving.Activate(actor, IntVector2.Right);

Nu ar fi frusos să puteți face ceva în felul următor?

actor.Move(IntVector2.Right);  
// Aceasta încearcă de a recupere componenta, pe urmă chemând Activate().  
actor.TryMove(IntVector2.Right);

Acesta este un pattern foarte comun, iar codul din urmă este foarte util în practică, eliminând mult boilerplate.

De fapt, eu generez metode ca aceste cu generatorul meu de cod. Le numesc *metode de alias*, deoarece acestea oferă un alias pentru versiunea mai lungă.

Pentru a permite generarea automată a unor astfel de metode alias pentru metoda Activate() de mai sus, decorați-o cu atributul Alias, astfel:

class Moving   
{   
 [Alias("Move")]   
 void Activate(Entity actor, IntVector2 direction) {}   
}

Cele două funcții vor fi generate automat, disponibile ca metode de extensie pentru tipul Entity.

### 3.6.6 Utilizarea Roslyn

În codul meu, Roslyn este utilizat în principal pentru a extrage datele necesare din codul sursă, prin detectarea claselor care implementează o anumită interfață, extragerea datelor din atributuri aplicate la tipuri, câmpuri și metode. Nu folosesc funcțiile de sinteză sintactică ale lui Roslyn, deoarece ele sunt prea verbose, chiar și pentru sarcini simple. Șabloanele T4 sunt mult mai ușor de citit și mai simple.

Să analizăm un exemplu, care a fost menționat în secțiunea anterioară: cum sunt extrase informațiile despre metodele *alias* din atributele relevante din codul sursă. Trebuie să vă avertizez că codul pe care urmează să îl vedeți nu este refactorizat și include chiar unele locuri cu cod mort, pe care încă nu l-am curățat.

Numai pentru acele funcții care sunt marcate cu atributul Alias trebuie să fie generat cod.

Scanez modelul semantic pentru clasele care implementează IComponent, pentru a obține clasele care ar putea defini metode alias. Pentru aceasta, folosesc SymbolFinder.FindImplementations(), a se vedea [codul sursă](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Meta/Templates/Logic/Shared/GenerationEnvironment.cs#L133);

Apoi găsesc toate metodele care au atributul Alias. A se vedea [codul sursă](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Meta/Templates/Logic/Components/TypeSymbolWrapperBase.cs#L66-L92). Aici, încerc să obțin atributul dat și, dacă există, îl transform într-un tip de atribut cunoscut (atributele sunt definite într-un proiect partajat, astfel încât generatorul de cod să poată transforma reprezentarea generică a unui atribut lui Roslyn în acest tip cunoscut). Pentru mai multe informații despre transformarea aceasta în tipuri de atribute cunoscute, consultați [această întrebare a mea pe stackoverflow](https://stackoverflow.com/questions/67539903/converting-attributedata-into-a-known-attribute-type-roslyn).

Așadar, am reușit am găsim simbolurile metodelor cu atributul Alias, definite în cadrul claselor care implementează IComponent.

Numele funcției generate trebuie să fie același ca aliasul din atributul Alias.

Deoarece am reușit să transformăm datele de atribut descoperite de Roslyn în tipul real de atribut, putem obține numele dorit prin simpla accesare a câmpului (proprietății) Alias după nume, deci aliasAttribute.Alias ar reprezinta numele alias-ului selectat.

Având în vedere că o metodă alias este întotdeauna definită pe o componentă și faptul că metodele alias sunt întotdeauna definite ca metode de extensie asupra clasei Entity, trebuie să luăm ca prim argument this Entity actor. Metoda marcată poate primi și alte argumente în afară de actor sau poate să nu primească deloc argumentul actor. Acest lucru înseamnă că metodele noastre generate trebuie să ia și aceste argumente suplimentare.

Așadar, ar trebui să analizăm semnătura funcției marcate pentru ca funcția noastră generată să aibă argumentele corecte. Practic, trebuie să copiem argumentele funcției marcate în funcția generată, cu excepția cazului în care argumentul dat este primul și indică actorul (deoarece este deja primul argument al funcției generate).

Acest lucru se face [aici](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Meta/Templates/Logic/Shared/SymbolExtensions.cs#L124-L141). Practic, dacă primul parametru este de tip entitate, returnăm toți parametrii, dacă nu este, returnăm un parametru de tip Entity, concatenat cu parametrii funcției și, în cele din urmă, dacă nu există parametri, parametrii vor include doar entitatea.

Generați codul.

Acum, tot ce a mai rămas de făcut este să generăm codul pentru aceste metode alias. Pentru aceasta, definim un fragment de șablon, [ca cel văzut aici](https://github.com/AntonC9018/hopper.cs/blob/6bed84a0603d0f1f782ab8f243d2df1adb36f286/Meta/Templates/Snippets/ComponentEntityExtension.txt#L49). Șablonul produce un cod puțin mai implicat decât cel arătat în exemplul de mai înainte. Acesta ține cont de tipurile de returnare în versiunea Try, returnând rezultatul apelului la metoda alias printr-un parametru out.

Utilizarea șablonului este la fel de simplă ca apelarea metodei TransformText() și scrierea rezultatului într-un fișier.

# Concluziile

În timpul dezvoltării acestui sistem am înțeles intuitiv multe lucruri legate de dezvoltarea jocurilor și sistemelor complexe în general. Am înțeles sensul decuplării componentelor, abstractizării, event-urilor, generării codului și altelor. Toate aceste tehnici aduc la o bază de cod mai ușor de întreținut, mai scalabilă și mai ușor de raționalizată. Am primit experiența în documentarea sistemei proprii, prin acestă lucrare și prin altă documentare pe care am scris-o pentru proiect.

Cu părere de rău, nu puteam să găsesc surse bune care să descrie dezvoltarea de așa sisteme complexe până recent, însă în continuare voi consulta cârțile și lucrările științifice scrise de profesioniste cu o experiență imensă în programarea jocurilor, pentru a minimiza experimentarea inutilă și a maximiza productivitatea, ca proiectele mele să fie mai gestionabile din start.

Am mai multe idei referitor la sistem ce deja am, de exemplu am format deja un plan vag de cum să adaug reprezentarea intermediară a tipurilor de entități pentru a realiza deserializarea din JSON.

# References

Anton, Curmanschii. n.d. “Dungeon-Hopper Pagina Pe Github.” <https://github.com/AntonC9018/Dungeon-Hopper>.