MINISTERUL EDUCAŢIEI
UNIVERSITATEA PETROL – GAZE DIN PLOIEŞTI
FACULTATEA LITERE ŞI ŞTIINŢE
DEPARTAMENTUL INFORMATICĂ, TEHNOLOGIA INFORMAŢIEI, MATEMATICĂ ŞI FIZICĂ
PROGRAMUL DE STUDII: INFORMATICĂ
FORMA DE ÎNVĂȚĂMÂNT: IF

LUCRARE DE LICENȚĂ

Conducător științific:
Lector dr. inf. Elia Dragomir

Absolvent:
Petre Tiberiu

PLOIEȘTI
2021

MINISTERUL EDUCAŢIEI
UNIVERSITATEA PETROL – GAZE DIN PLOIEŞTI
FACULTATEA LITERE ŞI ŞTIINŢE
DEPARTAMENTUL INFORMATICĂ, TEHNOLOGIA INFORMAŢIEI, MATEMATICĂ ŞI FIZICĂ
PROGRAMUL DE STUDII: INFORMATICĂ
FORMA DE ÎNVĂȚĂMÂNT: IF

Vizat,
Facultatea Litere și Științe

Aprobat,
Director de departament,
Conf. dr. inf. Gabriela Moise

LUCRARE DE LICENŢĂ

Dezvoltarea unei aplicații a inteligenței artificiale în jocuri video

Conducător științific:
Lector dr. inf. Elia Dragomir

Absolvent:
Petre Tiberiu

PLOIEȘTI
2021

UNIVERSITATEA PETROL - GAZE DIN PLOIESTI
FACULTATEA: LITERE ȘI ȘTIINȚE
DOMENIUL: INFORMATICĂ
PROGRAMUL DE STUDII: INFORMATICĂ
FORMA DE ÎNVĂȚĂMÂNT: IF

Anexa 9

Aprobat, Director de departament, Conf. dr. inf. Gabriela Moise	Declar pe propria răspundere că voi elabora personal proiectul de diplomă / lucrarea de licență / disertație și nu voi folosi alte materiale documentare în afara celor prezentate la capitolul „Bibliografie”. Semnătură student(ă):
DATELE INIŢALE PENTRU PROIECTUL DE DIPLOMĂ / LUCRARE LICENŢĂ / LUCRARE DISERTAŢIE	
Proiectul a fost dat studentului/studentei: Petre Tiberiu	
1) Tema proiectului / lucrării: Dezvoltarea unei aplicații a inteligenței artificiale în jocuri video	
2) Data eliberării temei: 5 Octombrie 2020	
3) Tema a fost primită pentru îndeplinire la data: 2 Noiembrie 2020	
4) Termenul pentru predarea proiectului/ lucrării: 13 Septembrie 2021	
5) Elementele inițiale pentru proiect / lucrare:	
- Inteligență artificială	
- Programare orientată pe obiecte	
- Programare paralelă, concurentă și distribuită	
6) Enumerarea problemelor care vor fi dezvoltate:	
Domeniul jocurilor video, inteligența artificială, implementarea arborilor de comportament și utilizarea acestora	
pentru a realiza o aplicație în domeniul jocurilor video.	
7) Enumerarea materialului grafic (acolo unde este cazul):	
35 de imagini, 1 tabel, 4 ecuații și 35 de secvențe de cod	
8) Consultații pentru proiect / lucrare, cu indicarea părților din proiect care necesită consultarea:	
Ori de câte ori a fost nevoie pentru consultări.	
Conducător științific:	Student(ă)
Lector dr. inf. Elia Dragomir	Petre Tiberiu
Semnătura:	Semnătura:

UNIVERSITATEA PETROL - GAZE DIN PLOIESTI Anexa 10
FACULTATEA: LITERE ȘI ȘTIINȚE
DOMENIUL: INFORMATICĂ
PROGRAMUL DE STUDII: INFORMATICĂ
FORMA DE ÎNVĂȚĂMÂNT: IF

APRECIERE		
privind activitatea absolventului: Petre Tiberiu		
în elaborarea proiectului de diplomă / lucrării de licență / disertație cu tema:		
Dezvoltarea unei aplicații a inteligenței artificiale în jocuri video		
Nr. crt.	CRITERIUL DE APRECIERE	CALIFICATIV
1.	Documentare, prelucrarea informațiilor din bibliografie	
2.	Colaborarea ritmică și eficientă cu conducătorul temei proiectului de diploma /lucrării de licență	
3.	Corectitudinea calculelor, programelor, schemelor, desenelor, diagramelor și graficelor	
4.	Cercetare teoretică, experimentală și realizare practică	
5.	Elemente de originalitate (dezvoltări teoretice sau aplicații noi ale unor teorii existente, produse informatice noi sau adaptate, utile în aplicațiile inginerești)	
6.	Capacitate de sinteză și abilități de studiu individual	
CALIFICATIV FINAL		

Calificativele pot fi: *nesatisfăcător/satisfăcător/bine /foarte bine /excelent*.

Comentarii privind calitatea proiectului/lucrării:
_____
_____
_____
_____
_____

Data:

Conducător științific
Lector dr. inf. Elia Dragomir

CUPRINS

TABEL DE FIGURI

LISTĂ DE ECUAȚII

SECVENȚE DE COD

INTRODUCERE

Proiectul de licență cu tema *„Dezvoltarea unei aplicații a inteligenței artificiale în jocuri video”* își propune implementarea unei metode moderne de dezvoltare a agenților inteligenți prezenți în jocuri video. Algoritmul propus folosește behaviour trees implementați într-un mod asincron pentru a facilita luarea de decizii fără ca agentul inteligent să facă risipă de resurse precum memorie și cicluri procesor.

Din experiența personală s-a observat că adesea se foloseau algoritmi precum arborii de decizie, automate cu stări finite sau algoritmi greu de întreținut folosind concepte clasice de programare. Implementarea mea încearcă să îmbunătățească abordările bazate pe un număr de stări finite, aducând una bazată pe sarcini executate asincron de către agenți inteligenți.

Un agent inteligent este unul capabil să preia informații din mediu, fie acesta fizic sau virtual, și pe baza stărilor acestuia, el execută acțiuni care modifică starea mediului. Agenții care învață sunt agenți inteligenți care pot reține anumite atribute din mediu pe o perioadă mai lungă și care îi ajută în luarea unor decizii ce devin acțiuni.

Obiectivul proiectului este de a implementa un API (o interfață pentru dezvoltarea unor aplicații) care permite scrierea de agenți inteligenți ce pot reacționa fluent la mediu fără necesitatea pierderii unor cicluri de calcul ale procesorului. Mai apoi, acest API a fost folosit pentru dezvoltarea unui joc video demonstrativ. Ca și limbaj de programare s-a folosit C# și mediul de dezvoltare Unity.

Din cercetările făcute până acum s-a descoperit că programatorii evit să folosească algoritmi inteligenți ce pot învăța deoarece acest domeniu este destinat divertismentului și amuzamentului publicului țintă. Acești algoritmi care învață pot deveni mult prea inteligenți față de o persoană care joacă și asta ar produce disconfort și dificultate, astfel pierind cheful jucătorilor.

Unele companii IT (Google DeepMind și Facebook OpenAI) au reușit să facă cercetării bazate pe Învățarea Automată ce vor sta la baza agenților inteligenți folosiți împreună cu tehnologia de cloud computing ce conduc la o experiență asemănătoare cu cea a jocului împotriva unui alt agent uman. Implementarea mea poate fi preluată și adaptată astfel încât agenții inteligenți să poată folosi Reinforcement Learning (învățarea prin întărire) pentru a selecta ramurile unui Behaviour Tree.

Curiozitatea care a stat la baza acestei lucrări a fost aceea de a învăța dezvoltarea jocurilor video, lucru care nu s-a studiat în facultate, și de a vedea ce fel de algoritmi de inteligență artificială se folosesc pentru implementarea inamicilor controlați de un calculator. Această curiozitate este adesea întâlnită și în rândul altor jucători, fie ei de orice vârstă.

Interesul unui programator/cercetător este acela de a vedea ascunzișurile unui domeniu ales și de a le descoperi ca apoi să poată dezvolta lucruri noi și concluziona peste acestea. Interesul stârnit cel mai puternic nu este numai cel de a reinventa roata, ci și de a dezvolta noi algoritmi și tehnici ce pot îmbunătății cele deja existente.

Structura lucrării de licență cuprinde următoarele capitole:

Capitolul 1. *Stadiul actual la domeniului*:

Acest capitol prezintă ce sunt jocurile video, o scurtă istorie a acestora, inteligența artificială și automatele cu stări finite, analiza inteligenței artificiale din jocul video Pac-Man, jocurile video ca și artă digitală, genuri de jocuri și o comparație între tehnologiile utilizate.

Capitolul 2. *Inteligența artificială aplicată în lumea jocurilor*:

Tipurile de agenți inteligenți, ce sunt aceștia, concepte de bază despre *behaviour trees* sunt subiectele abordate în acest capitol. Se va vorbi și despre ce sunt arborii de comportament, arbori bazați pe cod, arbori stocastici și bazați pe evenimente, respectiv utilități, de ce arborii de comportament sunt avantajoși în anumite situatii și o scurtă prezentare a arborilor de comportament asincroni.

Capitolul 3. *Dezvoltarea aplicației EvoAgents*:

În acest capitol se va prezenta arhitectura aplicației, tipuri de agenți inteligenți existenți în jocul video dezvoltat, structura arborilor de comportament ai acestora, interacțiuni între agenți, analiza și proiectarea aplicației și implementarea librăriei API. Se va prezenta și teoria structurală a mediatorilor și a *blackboard*, se va vorbi despre cum sunt construiți arborii de comportament și de ce aceștia sunt asincroni. În final se prezintă testarea aplicației și cum au fost depistate erorile de cod.

Capitolul 4. *Ghidul dezvoltatorului*:

Deoarece implementarea mea este un API, este necesară prezentarea modului de utilizare a acesteia. Se vor prezenta moduri de extindere a arborilor, cum se pot utiliza clasele *blackboard* și *mediator* pentru a izola comunicarea între agenți și se va prezenta modul de construire a unui agent inteligent.

Capitolul 5. *Concluziile finale*:

In finalul lucrării se vor prezenta obiectivele atinse de către mine ca dezvoltator, aplicații și în alte domenii ale implementării arborilor de comportament și se va încheia cu experiența dobândită de mine în urma dezvoltării acestei lucrări.

CAPITOLUL 1 STADIUL ACTUAL AL DOMENIULUI JOCURILOR VIDEO

În acest capitol sunt prezentate premisele domeniului, ce sunt jocurile video, istoria acestora și genuri de jocuri video. Spre final se va prezenta câte un pic despre inteligența artificială și cum a apărut aceasta.

1.1 CE SUNT JOCURILE VIDEO

O primă definiție a termenul de joc video (en. Video Game) se regăsește în (Wolf, 2008). Autorul prezintă, din două perspective, că un joc video reprezintă un mod recreativ de petrecere a timpului liber, respectiv termenul video însemnând tehnologia ce stă la baza acestora. Adesea se folosesc și termeni precum *jocuri de calculator* și *jocuri electronice* pentru a face referire la jocurile video. Jocurile electronice nu erau considerate ca făcând parte din această categorie deoarece acestea nu foloseau elemente vizuale, ci doar text, ele fiind numite și text based games. Și unele jocuri de masă utilizează aparate electronice care scot sunete, acestea făcându-le tot jocuri electronice. (*„Stop Thief”,* 1979)

Colocvial, termenul de joc video face referire la acele jocuri electronice sau de calculator care folosesc grafică interactivă pentru a reprezenta povestea acestora. Multe dintre acestea sunt jocuri *„single-player”* (cu un singur jucător) și el are de a face cu oponenți controlați de calculator (*inteligență artificială*)

Experiența de gaming (activitatea de a juca un joc video) este asigurată de interacțiunea dinamică cu acesta. Acest lucru stă de fapt la baza jocurilor video și le definește. „*Jocuri precum cele de cărți sau de masă implementate digital nu pot fi considerate jocuri video în sensul clasic deoarece primele jocuri video au fost de acțiune, și nu foloseau cărți de joc.”* (Wolf, 2008)

“Keith Feinstein (...) a sugerat că actul de juca un joc video are un element emoțional, similar cu cea de sârguință împotriva unui coechipier de joc, de îndemânare și abilități comparabile” (Wolf, 2008)

Totuși, acele programe software: *Solitaire*, *Minesweeper* și *Pinball Space Cadet*, jocuri populare in anii 2000 datorită Windows XP sunt, și vor rămâne, considerate jocuri video în definiția amplă găsită în cultura populară. Cu toate acestea, termenul de joc video rămâne unul vast în societatea contemporană decât sensul tehnic inițial.

1.1.1 TEHNOLOGII DE JOCURI VIDEO

Primele jocuri video nu foloseau inteligență artificială, acestea erau rudimentare și se numeau jocuri tip *arcade* deoarece rulau pe aparate simple denumite *Arcade Machines*. Acestea au stat la baza dezvoltării industriei de jocuri video, folosind ecrane *CRT* (*cu tub catodic*) de rezoluție 480p. Unele aveau grafică *vectorială* (elemente grafice generate matematic de vectori), altele *rasterizată* (imagini generate cadru cu cadru) și aveau un *microcontroler*, de aici și impresia că erau jocuri de calculator. Termenul de joc de calculator, adesea folosit pentru a acoperi o arie mai mare de jocuri (incluzând cele bazate pe text), este unul discutabil deoarece majoritatea jocurilor video depind de microprocesoare.

Adaptări ale acestor jocuri au apărut pe diferite sisteme „on house” (console) ce erau atât portabile, deci cu rezoluții mai mci decât un televizor, cât și conectabile. Nintendo Virtual Boy, consolă VR, avea afișaje monocrome de rezoluție 384 x 224 pixeli, aceasta putea producea stări de rău de mișcare. Iar console portabile precum Sega Game Gear și Nintendo Game Boy au apărut tot în aceeași perioadă și foloseau ecrane LCD reflective de calitate proasta (anii ’70 +).

Pentru a combate dificultatea rezoluției mici, s-a recurs la utilizarea unui careu de pixeli pentru a reprezenta imagini rasterizate care împreună creează efectul grafic. Acest lucru a permis dezvoltarea tehnologiilor și apariției jocurilor de consolă, și o dată cu computerul personal și a jocurilor video pe calculator.

1.2 ISTORIA PRIMELOR JOCURI VIDEO

Alan Turing este primul care a conceput ideea unui joc pentru calculator, acesta propunând un joc al imitării, destinat mașinilor Turing. În acest joc este vorba de un dialog deschis cu o mașină, dacă jucătorul nu reușea să facă diferența dintre un calculator și o persoană atunci se spune că mașina a trecut acest test, ea fiind astfel considerată un agent inteligent.

Acest concept stă la baza inteligenței artificiale și cu ajutorul lui Joseph Weizenbaum, un profesor la MIT, s-a reușit inventarea unui joc intitulat Eliza (1966) ce a apărut după 13 ani de la ideea lui Turing.

1.2.1 JOCURILE CALCULATOARELOR MAINFRAME

În 1952 au apărut primele jocuri de simulare militare scrise de Bob Chapman și alți cercetători de la Rand Air Defence Lab. Aceste jocuri rulau pe niște calculatoare mari intitulate mainframe. În același an au fost scrise și jocuri mici precum *tic-tac-toe*. Tot pe atunci, un cercetător la IBM a demonstrat funcționalitatea jocului său de dame, ce folosea inteligență artificială pentru a muta piesele de joc, pe calculatorul IBM 701.

Peste 10 ani aceste jocuri au evoluat, a început să apară acele calculatoare de tip terminal care se conectau prin rețea la mainframes. Aceasta a permis inventarea de jocuri educative pentru copii precum „The talking typewriter” care îi învăța să scrie folosind tastatura unei mașini de scris. Pentru afișarea textului erau utilizate ecrane CRT (cu tub catodic).

Pentru că aceste tipuri de calculatoare erau scumpe și greu de întreținut, ele fiind foarte mari și aveau nevoie de spațiu aerisit, jocurile în această perioadă au fost scrise numai la universități de prestigiu și alte instituții cu scopul de a distra angajații și chiar muncă de cercetare pentru inventarea acestora.

1.2.2 JOCURILE ARCADE ŞI DEZVOLTAREA INDUSTRIEI

În jurul anilor ’60 existau jocuri arcade, erau scumpe, dar erau rudimentare, se găseau prin unele săli de jocuri și nu foloseau elemente video. Această perioadă era populată de jocurile pinball, acele jocuri mecanice ce foloseau niște butoane și manete pentru controlul unei bile care trebuia să lovească cât mai multe obstacole fără a o pierde.

Companii precum Sega, Williams și Midway creau astfel de aparate de pinball înainte de a se lansa în industria jocurilor video.

Începutul anilor ’70 a însemnat apariția renumitului joc video Atari, PONG (1972) cu câștiguri de până la $40 pe zi și un număr de 150.000 de unități. Acest joc era bazat pe conceptul jocului de tenis de masă cunoscut popular ca ping-pong. Jocuri PONG au fost făcute în masă iar producătorii au profitat de acest succes și au început să inventeze clone cu diferite tematici ale acestui joc. Majoritatea jocurilor arcade erau inspirate din conceptul de jocuri sportive, și simulatoare de zbor, iar în 1974 piața de mașini arcade era plină de astfel de jocuri.

A doua jumătate a acestui deceniu a constat în dezvoltarea tehnologiei video pentru astfel de aparate iar in 1975 a apărut primul joc video arcade modern, Indy 800, așa cum se poate vedea în *fig. 1***Eroare! Fără sursă de referință.**, care era un simulator de curse pentru 8 persoane ce era controlat folosind volane și pedale.

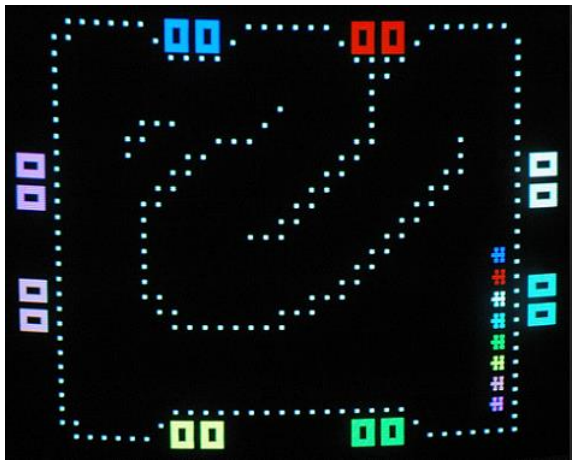


fig. 1 Indy 800, terenul de joc afișat pe un ecran aflat în mijlocul cabinei (Indy 800, 2021)

În 1976 au apărut jocuri arcade cu inamici ce trebuiau atacați și omorâți sau bazate pe filme de acțiune existente pe vremea aceea.

Conceptul de PONG a continuat iar Atari a produs jocul video Breakout (1976) cu un număr de 3.500 de cabinete comercializate și investiții de $7000. Acesta avea o paletă și o minge iar jucătorul trebuia să se folosească de acestea pentru a distruge cât mai multe cărămizi de diferite culori.

Spre sfârșitul anilor ’70, apariția primei console, Atari 2600, a permis industriei arcade să se răspândească chiar până în casele oamenilor.

1.2.3 Analiza inteligenței artificiale din jocul Pac-Man

Majoritatea jocurilor arcade erau de acțiune, cu mașinuțe sau aerovehicule. Jocuri precum *„Space Invaders”, „PONG”* sau *„Breakout”* erau cele mai populare găsite în sălile de jocuri. Cu această ocazie, compania japoneză Namco (Atari), s-a gândit să scrie un joc pentru toate vârstele. Acest joc avea să se numească Pac-Man (*fig. 2*) și a fost dezvoltat de Toru Iwatani la vârsta de 22 de ani în anul 1977.

Pac-Man a devenit un joc foarte popular, cu un număr de 100.000 de unități vândute și câștiguri de un miliard de dolari. Acesta a avut multe apariții în perioada 1980-1999. Inițial a pornit ca un joc arcade, de cabinet, dar cu popularizarea consolelor video Atari, acest joc a pătruns și în casele oamenilor.

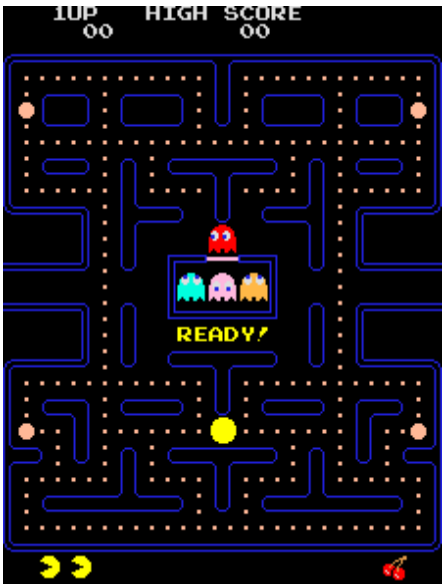


fig. 2 Jocul video Pac-Man (1977) pentru consola Atari 2600 (Pac-Man, 2021)

În articolul său științific, (Mateas, 2003) face o analiză a inteligenței artificiale implementate în jocul Pac-Man. Aici jucătorul este o minge galbenă cu gură mișcătoare care se plimbă printr-un labirint plin cu puncte ce trebuie mâncate pentru a le acumula. Un nivel este completat atunci când toate au fost strânse și jucătorul mai are viață.

În acest joc, inteligența artificială, sub forma unor patru fantome, încearcă să prindă jucătorul – dacă acesta a fost atacat, el pierde o viață. Patru puncte mai mari, „energy dots", îi permit lui Pac-Man să scape de fantome, acestea devenind speriate și putând fi mâncate, astfel fiind trimise la start.

Multe analize anterioare efectuate asupra acestui joc au arătat că pot exista algoritmi de navigare spațială (ca de exemplu, A* pentru găsirea drumului cel mai scurt până la Pac-Man) sau curbe de dificultate care îngreunează șansele jucătorului de a câștiga dar și elemente de power-up care îl fac pe jucător invincibil pentru perioade scurte de timp. Studiul realizat de către autor prezintă analiza comportamentului fantomelor și cum acesta alterează experiența de joc.

Într-un interviu, Toru Iwatani, autorul jocului, explică de ce a fost necesară introducerea inteligenței artificiale (a fantomelor).

„Bine, aici (în Pac-Man) nu există așa de mult divertisment într-un joc despre mâncat, așa că am decis să realizăm inamici care să introducă un pic de entuziasm și suspans. Jucătorul are să lupte inamicii ca să obțină mâncarea. Fiecare inamic are personalitate proprie." (Lammers, 1989)

Fiecare fantomă își are propria ei personalitate, așa cum spune Iwatani în interviul său, deoarece dacă acestea erau doar capabile să urmărească jucătorul atunci s-ar fi format o linie de fantome în urma lui Pac-Man și asta ar fi făcut jocul să nu mai fie interactiv. Asta înseamnă că motivul de a implementa patru fantome ar fi fost unul lipsă deoarece toate se comportă la fel și, astfel, numai cea care se află în fața tuturor ar conta. Singura diferență ar consta în viteza acestora de a urmării jucătorul iar dacă sunt prea rapide decât Pac-Man atunci întotdeauna ar fi putut să îl prindă. Altfel, dacă sunt lente, nu ar fi putut să îl prindă niciodată, acesta fiind mereu cu un pas înaintea lor. În realitate, comportamentul acestora este unul de echipă, ele încercând să îl prindă într-o capcană.

Datorită faptului că acțiunea se petrece într-un labirint, acestea au o multitudine de căi de acces pentru a îl prinde pe Pac-Man. Dacă toate ar face același lucru ar devenii imposibil de greu de a ferii fantomele, acestea așteptând mereu pe la colțuri.

Iwatani descrie în detaliu, în interviul susținut de acesta, cum funcționează IA în Pac-Man (Lammers, 1989):

INTERVIEWER: „Care a fost cea mai dificilă parte în realizarea jocului?"

IWATANI: „Algoritmul celor patru fantome care sunt inamicii fioroși din Pac-Man – realizarea corectă a coordonării mișcărilor acestora. A fost complicat deoarece mișcările monștrilor sunt un pic complexe. Aceștia sunt inima jocului. Am dorit ca fiecare inamic fantomă să aibă o personalitate specifică și propriile mișcări particulare, așa încât acestea nu doar să îl urmărească pe Pac-Man într-o singură coadă, ceea ce ar fi fost obositor și sec (pentru jucător). Una dintre acestea, fantoma roșie, intitulată Blinky, îl urmărea direct pe Pac-Man. Cea de a doua fantomă se poziționează într-un loc cu câteva puncte în fața gurii lui Pac-Man. Aceasta este poziția ei. Dacă Pac-Man este în centru atunci Fantoma A și Fantoma B sunt echidistante față de el, dar fiecare se mișcă independent, aproape făcându-l sandviș. Celelalte fantome se mișcă mai mult aleator. În acest fel ele ajungând aproape de Pac-Man într-un mod natural. Când un jucător este constant atacat în felul acesta, el devine descurajat. Așa că noi am dezvoltat un atac învăluit – întâi atacă apoi desparte; cu cât trece timpul cu atât fantomele se regrupează și atacă din nou. Pe parcurs, vârfurile și văile curbei de învăluire devin din ce în ce mai puțin pronunțate astfel încât fantomele atacă mult mai frecvent."

Din acest răspuns se poate afla că comportamentul variază între fantome (diferiții agenți inteligenți) dar și în timp (diferitele moduri de comportament). Comportamentul diferit al fantomelor deschide oportunități de interacțiuni între agenți variate, precum cel de învăluire a jucătorului atunci când este petrecut prea mult timp în același nivel. În acest interviu, Iwatani nu descrie amănunțit comportamentul fantomelor, deci nu se poate spune concret care este algoritmul IA utilizat.

După descrierea făcută, este posibil să fi fost folosită o implementare a unor automate cu stări finite, structuri precum grafurile orientate care folosesc un set de stări și tranziții între acestea astfel încât fiecare stare să descrie o acțiune ce poate fi efectuată la un anumit moment de către o fantomă (agent inteligent). Se pot observa următoarele atribute ale agenților:

- Fiecare fantomă are propria ei personalitate: *Shadow (Blinky)*, fantoma roșie, are un comportament de urmărire constantă a lui Pac-Man folosind un algoritm de urmărire foarte simplu (posibil Depth-First Search Algorithm, căutarea în adâncime). *Speedy* este fantoma roz, este cea mai rapidă dar se plimbă într-o manieră impredictibilă. *Blushful* este fantoma albastră, inițial se află într-o stare de timiditate și fuge tot timpul de Pac-Man, dar dacă Pac-Man se apropie de ea foarte mult aceasta prinde curaj și începe să îl urmărească. (Acele momente în care jucătorul este urmărit de doua fantome o singură dată.) *Pokey*, precum Speedy, merge impredictibil dar mult mai încet decât celelalte fantome.
- Fiecare dintre fantome folosește o strategie diferită pentru a îl ataca pe Pac-Man. Una îl urmărește, alta încearcă să îl prindă prin față folosind rute alternative prin labirint, alta îl pândește prin mijloc iar cea de a patra obișnuiește să se plimbe prin jur.

Fiind atât de dificil în a face distincție între comportamente, și de a decide ce face fiecare, se poate spune că acest model de IA este unul reușit, chiar dacă acesta este bazat pe niște algoritmi simpli. Pentru o înțelegere detaliată a IA, chiar dacă este prezentată în limbaj natural, ar însemna o descriere detaliată a stărilor în care agenții se află și felului în care acestea se schimbă pe parcursul jocului. Dificultatea prin care Iwatani a trebuit să treacă pentru realizarea jocului deschide noi oportunități de implementare a unor agenți mult mai inteligenți decât cei din Pac-Man.

1.2.4 JOCURILE ANILOR ’80, NINTENDO ȘI REPOPULARIZAREA INDUSTRIEI

O dată cu apariția jocului Pac-Man, anii ’80 a însemnat apariția jocurilor video arcade ca de exemplu: *Missile Command, Defender, Battlezone, Donkey Kong* și *Mario*. Donkey Kong (*fig. 3*) este jocul care a condus la apariția personajului Mario, un instalator care trebuie să salveze prințesa Peach din mâinile gorilei „Donkey Kong” și mai târziu, din cele ale lui Bowser.

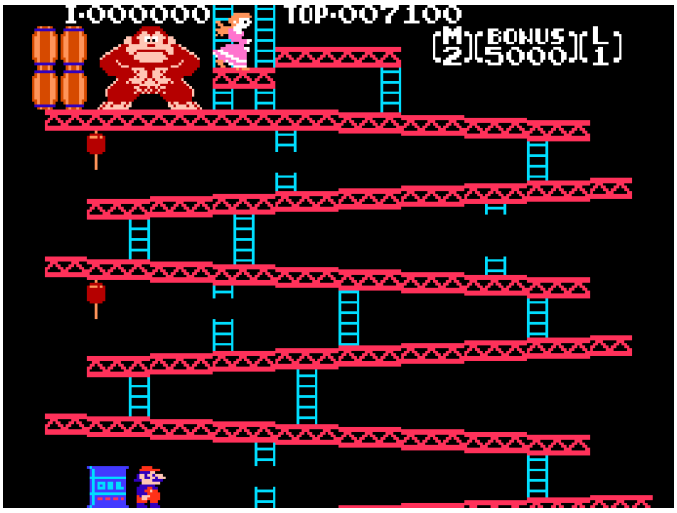


fig. 3 Donkey Kong - jocul arcade ce stătea la apariția a ceea ce va fi Mario (Donkey Kong, 2021)

Și compania japoneză Sega producea jocuri video recunoscute și astăzi în cultura populară precum *Sonic The Hedgehog*, *Frogger* și *Space Fury*. În aceeași perioadă a avut loc și un mare declin în industrie. Multe dintre cabinete, disponibile în sălile de jocuri, nu reușeau să își scoată banii după urma investiției. Lumea obișnuia să se plictisească de același gen de jocuri recomercializate.

Au avut loc multe încercări de repromovare a jocurilor video, unele dintre acestea au adus la apariția consolelor ce utilizau tehnologia discurilor laser, dar era o tehnologie scumpă pentru începutul anilor ’80 iar jocurile nu se cumpărau precum consolele.

O altă încercare a fost dezvoltarea unui joc video ce folosea *ray casting* pentru simularea unui mediu 3D, dar era mult prea devreme inventată tehnologia. Monitoarele acelor vremuri nu permitea rezoluții mari și deci imaginile erau prost afișate. Jocuri precum *Doom* și *Wolfenstein 3D* care foloseau această tehnică au apărut abia la începutul anilor 1990.

A doua jumătate a anilor ’80 a fost una de repopularizare, compania Nintendo a fost cea care a venit cu ideea dezvoltării consolei de jocuri ce folosea ca mediu de stocare acele „game cartridges” (casete cu microcip pentru stocarea jocurilor). Până atunci, consola Atari 2600 (1977) folosea casete cu bandă magnetică (*fig. 4*) ca mediu de stocare, în loc de jocuri programate în microcip.

fig. 4 Casetă cu bandă magnetică pentru consolele Atari

Consola de jocuri dezvoltată de Nintendo se numea *Nintendo Famicom* (în Japonia) respectiv *Nintendo Entertainment System* (NES – în USA). Aceasta a fost lansată în 1985 cu un număr de 61.9 milioane de unități vândute.

NES avea o colecție largă de jocuri video ce puteau fi jucate acasă la televizor. Jocuri precum *Mario World* și *Super Mario World* (fig. 5), *Duck Hunt* (fig. 6) și altele devenise foarte populare. Nintendo a fost și prima companie ce a produs console portabile precum *Game Boy*, *Game Boy Color* și *Game Boy Advance* ce permiteau rularea de jocuri precum *Tetris* sau *Pokémon* oriunde ești.

fig. 5 Jocul video Super Mario World pentru Super NES (Super Mario World, 2021)

In 1993, o dată cu popularizarea World Wide Web, jocurile video au devenit accesibile unui public mult mai larg, jocuri precum *DOOM* și *DOOM II* ce rulau pe calculatoare personale cu Windows 95 începeau să se regăsească în orice familie care aveau un computer personal (PC). *Jocurile multiplayer*, ce puteau fi jucate prin internet cu oricine, de oriunde, încep să se răspândească peste tot în lume.

fig. 6 Jocul video Duck Hunt, folosea un pistol cu infraroșu pentru a ținti și împușca rațele

(Time Magazine - Duck Hunt, 2021)

Tot în această perioadă au apărut și consolele *Sony PlayStation* și *Nintendo 64* care aduceau un nou concept pentru jocurile video de acasă. Acestea aveau grafică 3D, erau jocuri pentru computere personale sau *PlayStation* și de obicei comercializate pe CD-ROM (Consola N64 încă se mai utiliza de casete cu cip, aceasta fiind ultima consolă de acest gen).

Aparatele arcade, fiind de dimensiuni foarte mari, nu au reușit să lupte împotriva jocurilor pentru PC. Spre deosebire de computerele personale, sălile de joc aveau un element special în sufletele jucătorilor, acela de socializare într-un loc public alături de prieteni.

1.3 JOCURILE VIDEO CA ŞI ARTĂ DIGITALĂ

Articolul din jurnalul Contemporary Aesthetics, *„Are video games art?”*, scris de (Smuts, 2005), ne vorbește despre jocurile video din perspectiva unui artist. Acesta le considera artă deoarece elementul grafic este unul realizat de artiști. Acesta spune că oferă chiar și motivele pentru care el le consideră artă din punct de vedere istoric, estetic și reprezentațional.

1.3.1 ARTA DIGITALĂ ŞI DE CE JOCURILE VIDEO SUNT ARTĂ

Jack Kroll (Smuts, 2005) spune că jocurile pot fi distractive din multe puncte de vedere dar acestea nu pot transmite o complexitate așa cum poate fi găsită în operele de artă. Un articol de la MIT, publicat în jurnalul „*Technology Review*” intitulat „*Formă de artă pentru era digitală*” critică mult spusele lui Kroll afirmând că acesta a subestimat foarte mult potențialul jocurilor video.

Eu sunt de părere că un joc video poate fi văzut ca pe un mijloc de a prezenta arta. Sunt jocuri precum *Minecraft* (2011) care au ca scop stimularea acestei creativități atât a dezvoltatorilor cât și a jucătorilor acestuia.

Autorul ne prezintă trei jocuri video, apărute în perioada 2000-2003 care au potențial de a putea fi considerate opere de artă. Acesta ne vorbește pe larg despre jocurile *Max Payne* (Remedy Entertainment, 2001), *Halo* (Bungie, 2001) și *Tom Clancy’s Splinter Cell* (Ubisoft, 2002). Pe acestea le consideră “state of the art” deoarece sunt jocuri sofisticate pentru tehnologia de atunci și au un potențial estetic foarte promițător.

Aceste trei jocuri erau trenduri puternice în acea perioadă și foloseau o tehnologie sofisticată. Narativele, grafica fotorealistă și lumile trei-dimensionale cu grafica detaliată și bogată le fac opere de artă în adevăratul sens al cuvântului, ce transmit emoții specifice similar diferitelor genuri și forme artistice.

La începuturi, jocurile video erau banale, asta le considera mediocre din punct de vedere artistic. Cu timpul tehnologia a evoluat, grafica a devenit din ce în ce mai realistă și au început să transmită emoții specifice artei. *„Deci, acest fapt le pot considera opere de artă? Categoric, pot fi văzute opere de artă comparabilă cu filmele și piesele de teatru care transmit o poveste."* (Smuts, 2005)

1.3.2 CLASIFICAREA GENURILOR JOCURILOR VIDEO

În cartea *„The video game explosion"* (Wolf, 2008) există un capitol care încearcă să explice ce fel de genuri de jocuri video există din punct de vedere interactiv. Fiecare sarcină și obiectiv pe care jucătorul trebuie să o atingă și felul în care acesta controlează jocul conduce la formarea unui gen de joc. O parte dintre genuri: *demonstrativ*, *educațional*, *puzzle*, *simulare* și *utilitar* nu sunt chiar jocuri video, ci doar aplicații software care sunt adesea incluse în această categorie. (ex: *Mario Teaches Typing – „joc video" educațional* pentru NES care te învață să folosești tastatura)

Fiecare gen de joc, prin expansiunea acestuia, a condus la formarea de sub genuri, ca de exemplu: *First Person* și *Third Person Shooter*, sub genuri ale genului *Shooter*. Multe dintre ele chiar se pot suprapune, astfel existând genuri precum *action-adventure*, *maze-escaping* (sau *escape room*), etc.

1. **Genul de jocuri abstract**

Genul abstract reprezintă totalitatea jocurilor cu grafică non reprezentațională și implică obiective neorganizate într-un fir narativ. Adesea implică construirea, vizitarea sau umplerea fiecărei părți de pe ecran. Jocurile pentru mobil, realizate cu scopul de petrecere a timpului liber pot fi și ele considerate abstracte deoarece obiectivul acestora nu este unul bazat pe poveste.

Exemple: Tetris (Puzzle), Pipe Dream (Collecting), Where is my water? (Collecting), Breakout (Shooting targets), Pac-Man (Escape, Maze, Collecting).

2. **Jocurile video de adaptare / Interactive Movie**

Sunt acele jocuri video adaptate după jocurile clasice precum sportive, de masă, de cărți sau jocuri adaptate după povești preluate din romane, nuvele sau benzi desenate. (ex: *nuvele vizuale*, jocuri produse de *Telltale Games*, etc) Acestea, din punct de vedere al dezvoltării, implică întrebări de genul „*cum trebuie modificată munca originală astfel încât jocul să fie unul interactiv?*" (Wolf, 2008) Unele jocuri pentru computer sunt chiar și adaptări ale jocurilor arcade.

Exemple: Casino, Solitaire, Blackjack, Poker (după jocurile de cărți); The Simpsons (video game adaptat după serial); Spiderman, X-MEN (după benzi desenate); Star Wars (după film); The Walking Dead (după roman); Hangman, Tic-Tac-Toe (după jocurile pen and paper); seria FIFA (după sporturi); PONG, Virtual Pool (după jocurile de masă); Life is Strange, Minecraft Story Mode (interactive movie)

3. **Genul de jocuri de tip Action / Adventure (Aventură)**

Jocuri aflate într-o lume virtuală alcătuită din încăperi sau spații a cărui obiectiv este unul complex față de simpla supraviețuire prin scăpare, prindere sau tragere după inamici. Obiectivele adesea conduc la provocări tip puzzle în care găsești chei ca să descui uși către alte zone unde găsești obiecte utile pe parcursul jocului. Adesea sunt bazate pe *Science-Fiction*, *Fantezie* sau *Spionaj* și invocă diferite ere sau locuri precum *Anglia pe vremea Regelui Arthur* (ex: *Lara Croft, Tomb Raider*). A nu se confunda cu jocuri precum *Donkey Kong* (în care ești limitat de nivel prin

mișcările de sari și mergi) sau jocuri adaptate precum *Star Trek* (joc adaptat din film), respectiv *The Walking Dead* (nuvelă vizuală, bazat pe roman).

Exemple: Atari 2600 Adventure, Crash Bandicoot, Myst, Haunted House, Superman, seria Tomb Raider.

Jocul dezvoltat de mine în acest proiect, *Rabbit Catcher*, este un joc *action / adventure*, cu un pic de *artificial life* și *captură* deoarece explorarea unei lumi generate aleator, în care sunt prezente animale, iepuri și lupi, reprezintă scopul tău ca jucător. Prevenirea iepurilor să nu fie atacați de către lupi este elementul cheie care îl face să fie un joc de tip *captură*.

4. Jocurile video Artificial Life (Viață artificială) / Management Simulator

Dacă trebuie să ai grijă de niște creaturi artificiale și de a le prevenii să nu moară atunci acesta este un joc de tip *artificial life*. Adesea obiectivul jocului este menținerea sănătății și bucuriei acestora la un nivel ridicat.

Jocurile care implică alocare de resurse sau bazate pe management (ex: *Cookie Clicker, Adventure Capitalist*) sunt jocuri de tip *Management Simulation*, adesea acestea sunt bazate pe strategie.

Exemple: The Sims, Pou, My Talking Tom / Angela (și altele de genul pentru telefon), Tamagotchi (artificial life); Sid Mayer's Civilization, Monopoly, SimCity. (Management)

5. Jocuri video de masă (Board Games) / Pen and Paper

Jocuri adaptate după jocuri de masă sau jocuri similare care nu existau înainte fac parte din acest gen. Acesta cuprinde și jocuri precum *Șah, Dame* sau *Table* sau jocuri adaptate după *Scrabble* sau *Monopoly*. A nu se confunda cu jocurile de masă ce implică acțiune (*biliard* sau *tenis de masă*) și nici cu cele care implică hârtie și pix (*pen and paper*) pentru a putea fi jucate (*Hangman, Tic-Tac-Toe*).

Exemple: Table, Nave de război, Conquest of the World, Triviador, Monopoly, Scrabble, Video Checkers / Chess.

6. Genul captură (Capturing)

Jocuri a căror obiectiv este acela de a captura obiecte sau personaje care fug de jucător. Asta poate implica și oprirea acestora din mișcare sau blocarea unei căi de acces. Aceste jocuri, spre deosebirea celor ce implică colectare sau prindere, obiectele sau personajele în cauză se află în mișcare.

Exemple: Keystone Keepers, Take money and run, Texas Chainsaw Massacre

7. Jocuri video de cărți / Gambling

Sunt jocuri video adaptate după jocurile cu cărți sau jocuri ce sunt precum cele care au la bază cărți de joc (precum jocurile tip *Solitaire*). Cum majoritatea folosesc cărți clasice de joc, altele au implementat propriile sale tipuri de cărți specializate (*1000 Miles*). Jocurile de tip *Trivia*, ce folosesc întrebări și răspunsuri înscrise pe cărți rămân considerate un gen propriu. Dacă jocul este unul în care trebuie să pariezi și să mizezi pentru a câștiga bani virtuali, atunci jocul este unul de tip *Gambling*

Exemple: 1000 Miles, Blackjack, Casino, Solitaire sau Poker

8. Genurile jocurilor de tip Catching și Colectare

Aceste genuri cuprind totalitatea jocurilor video care au ca scop prinderea de personaje controlate de calculator sau colectarea de obiecte aflate în mediul virtual. Genul *captură* presupune prinderea prin atac sau blocare, iar *catching* reprezintă prinderea prin capcană.

Exemple: Amidar, Mousetrap (Catching), Pac-Man (Colectare).

9. Genul Combat / Fighting

Sunt jocuri în care doi sau mai mulți jucătorii se luptă unul împotriva celuilalt (1v1 / PvP) sau un jucător împotriva unor agenți controlați de calculator. Aceștia se atacă corp la corp sau cu proiectile până ce unul este eliminat fie de pe hartă, fie prin terminarea vieții acestuia.

Exemple: Mortal Kombat, Brawlhalla, Super Smash Bros. Ultimate

10. Jocuri de tipul demonstrativ (Demos)

Jocuri video scurte descărcate de pe internet sau achiziționate din magazine ce au ca scop demonstrarea idei principale și de a forma o primă impresie jucătorului înainte ca acesta să își cheltuie banii. Adesea sunt disponibile gratuit, sau la un preț redus, pentru a putea fi achiziționate.

11. Jocurile video de tipul Dodging

Într-un joc de acest tip, jucătorul trebuie să evite proiectile sau alte obiecte ce se îndreaptă asupra sa. Cu cât mai multe obstacole evitate, cu atât scorul este unul mai mare. A nu se confunda cu genul *Shooter* sau *Combat*, genuri în care poți trage și trebuie să eviți să fi împușcat de inamici.

Exemple: Frogger, Space Invaders, Dodge 'Em, Freeway.

12. Genul Driving / Racing / Flying

Jocuri video bazate în principal pe îndemânarea jucătorului de a conduce / manevra vehicule. O subcategorie este aceea de tip *Racing*, în care scopul principal este acela de a câștiga o cursă. Dacă vehiculul manevrat este unul precum un avion, atunci jocul este unul de tip *Flying Simulator*.

Exemple: Indy 800, Red Planet, Street Racer, Asphalt, Super Mario Kart (Racing); Microsoft Flying Simulator (Flying), War Thunder (Flying and Driving), World of Tanks, Warships (Driving)

13. Jocurile video educaționale

Jocuri ale căror obiective sunt acelea de a învăța jucătorul anumite lecții de matematică, logică sau limbi străine. Modul în care acestea te învață este unul interactiv și distractiv, spre deosebire de lecțiile clasice care nu sunt pe placul copiilor.

14. Genul jocurilor escape

Obiectivul principal este acela de a rezolva un „*puzzle*” pentru a putea scăpa dintr-un loc sau încăpere. Jocurile în care trebuie luptat împotriva unor inamici nu sunt considerate jocuri *escape*, de fapt acestea sunt de tip *Combat* sau *Shooting*.

Exemple: Pac-Man (labirint), Escape Room (încăpere), Maze Craze (labirint).

15. Jocurile de tipul Role Playing Games (RPG)

Sunt jocuri video în care ești un personaj, adesea războinic, care trebuie să se dezvolte rezolvând misiuni primite de joc pentru a-ți crește statusurile precum *abilități*, *putere* și *dexteritate*. Aceste jocuri sunt de două feluri *Single-Player RPG*, în care faci misiuni singur, sau *MMO RPG (Massive Multiplayer Online RPG)*, în care joci în rețea împreună cu alți jucători.

Exemple: World of Warcraft, ACE Online, Metin 2, AION, Black Desert Online

16. Genul Shooter

Genul *shooter* reprezintă totalitatea jocurilor în care trebuie să tragi cu arma în inamici. Supraviețuirea este esențială deoarece numărul de vieți este unul limitat, iar dacă mori, șansele de a mai putea continua sunt foarte minimale. Este un gen relativ tânăr și popular deoarece a apărut după 1990 și sunt cele mai jucate în rândul tinerilor.

Exemple: Team Fortress, Counter-Strike, Unreal Tournament, Half Life, Doom, Wolfenstein 3D (înainte de 2000); Valorant, Counter-Strike Global Offensive, Overwatch (după 2000)

Alte genuri: *Obstacle Course (Freeway, Frogger, Pitfall!); Pinball (Pinball Space Cadet); Platform (Donkey Kong, Mario, Super Mario Bros., Yoshi's Island); Programming Games (AI Fleet Commander, AI Wars, Game Builder Garage, CORE); Puzzle; Quiz / Trivia (Triviador, $25,000 Pyramid, Wizz Quiz); Rhythm and Dance (Just Dance, Beatmania, GuitarFreak, Pop 'n Music); Simulation; Sports (FIFA, American Football, Miniature Golf, Summer Games, Sonic and Mario at The Olympics); Table-top (PONG, Electronic Table Soccer!); Target; Text Adventure; Training*

1.4 DEZVOLTAREA INTELIGENȚEI ARTIFICIALE ŞI JOCURILE VIDEO

Jocurile video au condus la dezvoltarea unei sub ramuri ale inteligenței artificiale, și anume, identificarea strategiilor de a scrie personaje controlate de calculator. Majoritatea jocurilor video necesită utilizarea unor agenți inteligenți pentru a permite dezvoltarea unei interacțiuni cu jucătorul. În acest sub capitol se va prezenta o comparație între cele trei strategii de implementare ale agenților inteligenți: *automate cu stări finite, arbori de decizie, respectiv arbori de comportament* (behaviour trees)

1.4.1 AUTOMATELE CU STĂRI FINITE

În articolul său, (Luond, 2019) prezintă o metodă de implementare a *automatelor cu stări finite* pentru dezvoltarea unor jocuri bazate pe mutări (în general de tip *Strategy*, *RPG*, *Combat*). În aceste jocuri este necesară o astfel de implementare deoarece trebuie cunoscută în permanență starea în care se află jocul pentru a ști ce acțiuni trebuie efectuate de agentul inteligent.

Există mai multe tipuri de implementări ale unui astfel de automat. În cartea sa, *„Artificial Intelligence for Games",* autorii (Millington & Funge, 2006) spun că există trei metode de bază pentru a îi implementa.

1. **Hard-coded finite-state machines (implementare statică, procedurală)**

Primele jocuri video ce foloseau agenți inteligenți utilizau o implementare statică a automatelor. Această metodă era simplă de implementat deoarece puteai scrie cod C sau ASM direct fără alte bătăi de cap, dar sunt complexe și greu de menținut și adesea se rezumă la mii de linii și cod încurcat deoarece tranzițiile sunt făcute folosind salturi prin *goto* (în C) sau instrucțiuni tip JMP (în Assembly).

Următorul algoritm (*code 1*) este un exemplu de implementare a unui automat cu stări finite folosind limbajul de programare C# și Unity. În acest exemplu, preluat din articolul lui (Luond, 2019), se prezintă algoritmul prin care, la fiecare *tick*, se determină starea în care se află un joc video, de tip combat, bazat pe mutări. Se începe cu starea *OutOfCombat* și treptat, pe parcursul unui meci, au loc mutări, astfel, trecerea între stări depinde de acțiunea jucătorilor (fie ei agenți umani sau artificiali).

```
public class HardCodedFSM : MonoBehaviour {
	public State currentState;
	void Update() {
		switch(currentState) {
			case OutOfCombat: DoSomething(); break;
			case PlayerChoice: WaitForPlayerAction(); break;
			case EnemyChoice: DoSomethingElse(); break;
			case EndOfCombat: ShowGameOver(); break;
		}
	}
	public enum State { OutOfCombat, PlayerChoice, EnemyChoice, EndOfCombat }
}
```

code 1 Exemplu de automat implementat static (Luond, 2019)

2. **Class-based finite-state machines. (implementarea dinamică, POO)**

Cu timpul, o dată cu dezvoltarea paradigmei orientate obiect, s-a observat că automatele pot fi mult mai ușor de menținut dacă fiecare stare este reprezentată printr-o clasă. Limbajul C++ a fost unul care a promovat și revoluționat acest mod de gândire, astfel popularizând utilizarea acestui tip de FSM.

Fiecare stare este o clasă care conține propria ei logică iar fiecare dintre ele conțin referințe către alte obiecte de tip stare, astfel realizându-se tranzițiile între stări (cunoscând starea actuală și regula de tranziție).

fig. 7 demonstrează un automat cu stări finite pentru un agent inteligent a cărui mutări sunt de tipul *„una la un moment dat”.* Acesta pornește din starea *Idle* și poate să se ducă numai în sus sau în jos pe hartă. Dacă acesta ajunge sus, atunci nu poate decât să coboare, altfel, dacă se află jos, trebuie să urce pentru a putea înainta. Fiecare decizie este dependentă de cealaltă iar implementarea acestui automat se poate realiza foarte ușor folosind programarea orientată pe obiecte deoarece tranzițiile între stări se poate realiza prin metode și funcții ale unui obiect dintr-o clasă.

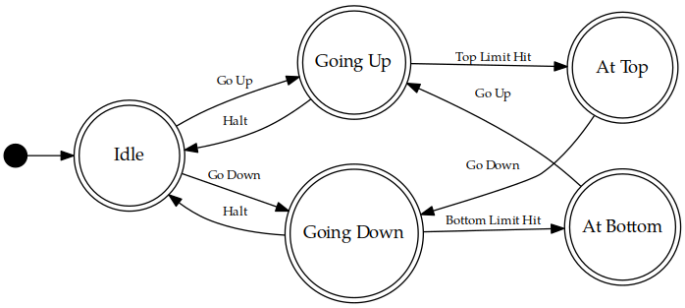


fig. 7 FSM AI implementat dinamic, POO

(Stack Exchange - Event-driven finite state machine, 2021)

3. **Hierarchical finite-state machines. (Automate implementate ierarhic)**

Sunt un tip modern de automate cu stări finite ce vin ca o completare pentru cele bazate pe clase de obiecte. Ideea acestora este de a reduce complexitatea prin împărțirea unor stări majore în mai multe sub stări și tranziții între acestea. Altfel spus, fiecare stare poate fi implementată folosind un sub tip de automat.

În *fig. 8* este prezentată imaginea unui *automat cu stări finit de tipul ierarhic*. Acest agent inteligent este un robot aspirator, el are sarcina de a aduna gunoiul din camere și să renunțe la el. Acesta pornește în starea *A*, sub stare a nodului *CleanUp.* Pe parcursul execuției, acesta caută gunoi în cameră, dacă îl vede atunci se duce după el, îl aspiră iar apoi pleacă către locul de depozitare. La finalul acestor treceri revine în starea de căutare. Dacă pe parcurs nu găsește gunoi se duce către stația de încărcare. Atunci când este reîncărcat, el revine în starea inițială și își continuă sarcinile.

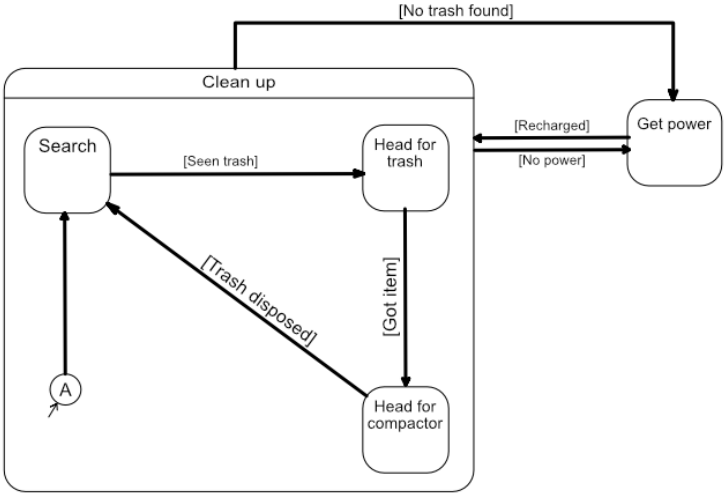


fig. 8 Hierarchical Finite-State Machine (Luond, 2019)

1.4.2 ARBORII DE DECIZIE

În articolul din 2007, *„Map-Adaptive AI for Video Games”* (Blom, 2007) se vorbește despre o metodă ce folosește arborii de decizie pentru reprezentarea lumilor dintr-un joc de strategie (*Real Time Strategy, RTS*). Agenții inteligenți se folosesc de această implementare pentru a putea lua decizii cât mai bune, de exemplu, locul în care trebuie creată o bază sau când trebuie trimise trupele pentru a patrula, etc. Se încearcă recunoașterea tipurilor de hărți astfel încât să se poată alege o strategie bună și să poată învinge oponenții. Au mai fost implementați și alți algoritmi, ca de exemplu *Alpha Star*, pentru învingerea jocului de strategie *StarCraft 2*, realizat de *Blizzard Entertainment*.

Un arbore de decizie (*fig. 9*), este o structură sub formă de arbore (graf orientat aciclic) în care fiecare ramură reprezintă un *atribut* ce stă la baza alegerii unei decizii, iar fiecare frunză este *clasificarea* asociată (sau *decizia*). Este un model de Machine Learning predictiv deoarece pentru fiecare intrare observată (ex. o hartă dintr-un RTS) se prezice un rezultat (*outcome*) asociat. Pentru că se folosesc asupra spațiilor de căutare discrete, aceste structuri se mai numesc și *arbori de clasificare* (în cazul jocurilor, se clasifică acțiuni pe baza stărilor mediului). *Frunzele* sunt *clasificări* iar *ramurile* sunt *conjuncții de atribute* care conduc la aceste *clasificări*. Acești arbori sunt printre tipurile de structuri folosite în video game AI (alături de automate și arborii de comportament)

Într-un joc de strategie, fiecare atribut poate însemna o informație asociată unui *„chunk”* (*porțiune de teren*) ca de exemplu *numărul de resurse*, *densitatea* acestora (cu cât este mai mare, cu atât este mai valoroasă) respectiv *numărul de obstacole* precum *munți*, *văi* și *râuri* (cu cât sunt mai multe, cu atât este mai puțin valoroasă). Iar fiecare *clasificare* poate însemna nivelul de „*valoare*” (care aduce valoare asupra strategiei) ca de exemplu: *nevaloros*, *puțin valoros*, *mai valoros*, *foarte valoros*. Sau *acțiuni* pe care un agent le poate face, ca de exemplu: *evită locul*, *încearcă să investești*, *investește în acel loc*. O *investiție* într-un teren înseamnă *construirea* unor structuri care permit adunarea de resurse care pot conduce la câștigarea jocului.

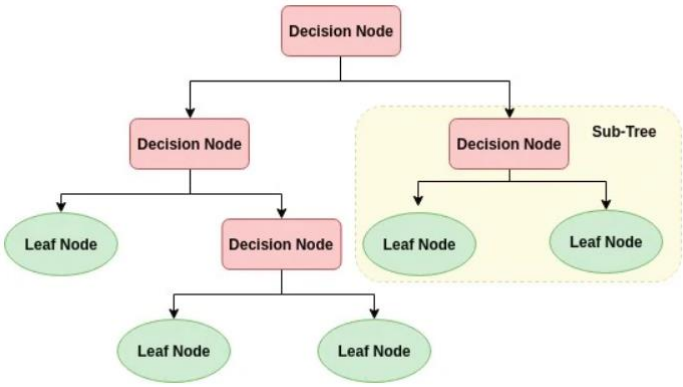


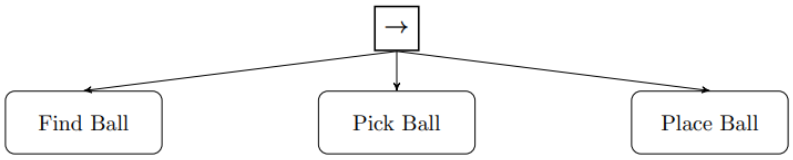
fig. 9 Structura unui arbore de decizie

1.4.3 BEHAVIOUR TREES

Cartea *„Behaviour Trees in Robotics and AI”* (Michele & Petter, 2018) este o introducere cuprinzătoare a *arborilor de comportament*, structură nouă generalizată, de a implementa IA pentru *jocuri video* și *robotică* bazată pe *planificarea sarcinilor* (*actions* / *tasks*) unui agent inteligent.

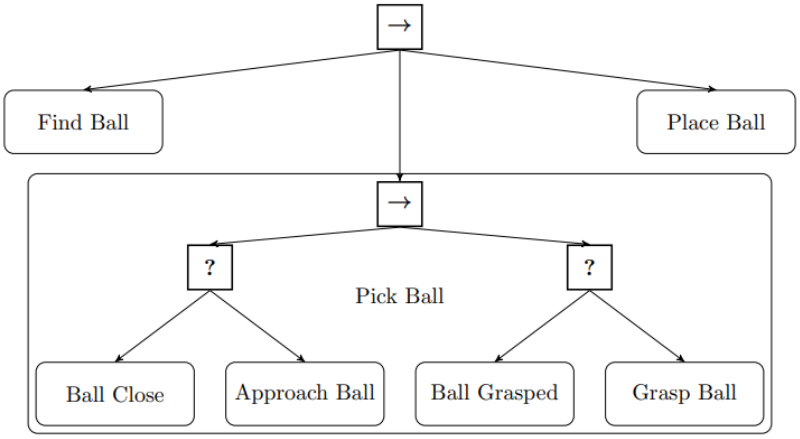
Un *arbore de comportament* (Behaviour Tree BT) este o structură de a comuta între *sarcinile* unui agent inteligent autonom, precum un robot sau o entitate virtuală dintr-un joc video. Altfel spus, un BT este o metodă eficientă de a crea *sisteme complexe* ce sunt *modulare* și *reactive*. Se poate observa structura unui astfel de arbore în fig. 10(a), respectiv fig. 10(b)

În imaginea din fig. 10(a), un braț robotic inteligent poate avea un arbore de comportament care să îl ajute să cunoască un plan ce îi permite realizarea *scopului*, și anume, acela de a muta o minge dintr-un loc în altul. Secvența de acțiuni este următoarea: *găsește mingea*, *ridică mingea* și *pune mingea* într-un alt loc.



(a) *Un BT care execută o sarcină (task) ce presupune găsirea, ridicarea și punerea într-un alt loc a unei mingii.*

Imaginea din fig. 10(b) reprezintă arborele de comportament extins al aceluiași agent inteligent. Acesta, pentru a putea apuca mingea, trebuie să verifice dacă se află în apropiere apoi să poată să o apuce. Împreună, *prinderea* și *apucarea* mingii constituie acțiunea de a o *ridica*.



(b) *Același arbore, doar că acțiunea „Pick Ball" a fost împărțită într-un sub arbore.*

fig. 10 Arbore de comportament ce modelează sarcina de mutare a unei mingi (Michele & Petter, 2018)

(i) O SCURTĂ ISTORIE A ARBORILOR DE COMPORTAMENT (BT)

Datorită problemelor de a menține actualizate implementările bazate pe FSM, respectiv DT (Decision Tree) s-a hotărât dezvoltarea unui algoritm de mijloc care să permită acest lucru. Acest algoritm, în urma dezvoltării automatelor ierarhice cu stări finite (Hierarchical FSM), a fost cel al BTs. În jocuri, structurile de control al NPCs (Non-Playable Characters) erau adesea formulate ca o colecție de stări și tranziții (FSM) ce erau *sisteme concurente* dar non-modulare. Arborii BT au devenit o alternativă la FSM ce suportă *sistemele modulare*. Un sistem AI concurent este un sistem în care trecerea de la o stare la alta se face liniar, iar sistemele modulare permite trecerea între stări într-un mod bazat pe module și sub module.

La Universitatea Carnegie Mellon, arborii de comportament au fost utilizați foarte mult pentru a implementa comportamente robotice. *„Avantajul principal este acela în care comportamente individuale sunt utilizabile în contextele altor comportamente de nivel superior, fără necesitatea specificării relațiilor dintre acestea."* (Michele & Petter, 2018)

Utilizarea lor au permis non-experților să poată dezvolta algoritmi pentru roboți ce execută sarcini precum mutarea de obiecte deoarece aceștia sunt „*modulari, o adaptare a unor sarcini robotice*" și au permis „*utilizatorilor de rând să poată crea vizual programe de același nivel de complexitate precum cele scrise tradițional*". (Michele & Petter, 2018)

CAPITOLUL 2 INTELIGENȚA ARTIFICIALĂ ÎN LUMEA JOCURILOR

Agenții inteligenți sunt utilizați în toate domeniile, de la *jocuri video*, *robotică*, și *muncă de cercetare* până în *medicină*, *chimie*, și alte ramuri ale științelor și a *imagisticii digitale*. În acest capitol se va prezenta ce sunt *agenții inteligenți*, *tipuri de agenți* și *concepte* și *tehnologii* utilizate în domeniul jocurilor video.

2.1 CE SUNT AGENȚII INTELIGENȚI ȘI TIPURI DE AGENȚI

Un *agent* este o *entitate computațională* care este capabil să *acționeze* în numele altor *entități* într-o manieră *autonomă*, este *reactiv* la mediul în care se află și este capabil să *învețe*, *coopereze* și să se *miște*. Un *agent inteligent* este un *agent* care îndeplinește simultan următoarele atribute: *autonomie*, *cooperare*, *învățare*, *reactivitate*, *comunicare*, *inferență* (operează în conformitate cu o specificație folosindu-se de cunoștințe dobândite anterior), *continuitate* (persistența în timp a identității și stării sale), *adaptabilitate* (poate să se dezvolte pentru rezolvarea situațiilor inedite).

Ca *tipuri de agenți inteligenți*, există patru feluri fundamentale: *agenți reflecși simpli*, *agenți reflecși bazați pe un model*, *agenți bazați pe scop*, *agenți bazați pe utilitate*. (Nicoară, 2020)

Agenți reflecși simpli:

- Sunt agenți inteligenți ce răspund direct la obiectele percepute.
- Deciziile alese de aceștia sunt formulate pe baza regulilor de tip condiție-acțiune
- Sunt recomandați în cazul în care deciziile corecte pot fi luate numai pe baza percepției curente asupra mediului (adică, mediul este total observabil, agentul cunoaște în totalitate mediul)
- *Ex: predictori liniari (de clasificare sau regresie)*

Agenții reflecși bazați pe un model:

- Spre deosebire de cei simplii, acest tip de agenți folosesc o structură de date care memorează informații ce nu sunt evidente în obiectele percepute. (Un fel de model al mediului în care se află aceștia)

Agenții bazați pe scop:

- Acționează pentru a atinge niște scopuri predefinite (ex: *alergare*, *vânare*, *evadare* pentru agenții din jocuri video)
- Se pretează în cazurile de *căutare* și *planificare*
- *Ex: agenții construiți folosind Behaviour Trees (algoritm folosit pentru planificarea scopului)*

Agenți bazați pe utilitate:

- Sunt agenți ce încearcă să-și maximizeze propria „*bunăstare*” așteptată. (Fiecare agent primește o recompensă pentru fiecare decizie aleasă)
- Dacă există multiple căi de a ajunge la scop, iar fiecare cale are un cost, aceștia vor ști care este cea mai *bună* / *rapidă* / *sigură* cale.
- *Ex: agenți inteligenți implementați folosind învățare prin întărire (reinforcement learning). Acest tip de agenți sunt viitorul agenților inteligenți folosiți în jocuri video. ex: Alpha Star (StarCraft 2), Alpha Go (jocul clasic de Go)*

2.2 *Behaviour Trees – Concepte şi tehnologii utilizate*

Un *agent* ce implementează un *Behaviour Tree* (*clasic*) este un *agent colaborativ* deoarece este *autonom* și *cooperativ* prin faptul că aceștia se folosesc de arbore pentru a planifica acțiuni viitoare dar nu pot *învăța* informații noi deoarece nu au această capacitate (nu dețin memorie). În jocurile video moderne (precum *Call of Duty* sau *Battlefield*) aceștia dețin capabilitatea de *învățare* deoarece folosesc o structură de date tip „*blackboard*" (un tabel de dispersie a căror chei sunt *atributele* („*string*") și *informația* sunt obiectele observate sau alte date despre acestea). În acest „*blackboard*", fiecare *atribut* este utilizat pentru testarea condițiilor ce permit luarea deciziilor.

Adăugând acest element, alături de *arborele de comportament*, aceștia devin *agenți inteligenți de planificare (bazați pe scop)*.

Arborii de comportament dezvoltați de mine, alături de cele de mai sus, folosesc decizii formulate sub formă de *sarcini compuse ierarhic* (fiecare sarcină au multiple sub sarcini) care sunt executate *asincron* (sarcina părinte nu este considerată completă decât abia atunci când fiul este terminat).

2.2.1 Code-driven Behaviour Trees

La bază, un BT este construit dintr-un set mic de simple *componente* care împreună alcătuiesc structuri foarte *performante* pentru implementarea *agenților inteligenți* din punct de vedere aplicativ dar și teoretic.

Formal, un BT este un *graf orientat aciclic (arbore)* a căror noduri sunt *noduri de control* iar frunzele sunt *noduri de execuție*. Terminologia utilizată este cea clasică, întâlnită în teoria grafurilor, și anume noduri sau *părinți*, respectiv sub noduri sau *fii*. *Rădăcina* arborelui este nodul *fără părinți*, iar *frunzele* sunt nodurile *fără fii*. Fiecare *nod de control* are cel puțin un *fiu*. Iar grafic, arborele de comportament este reprezentat precum un arbore obișnuit. *Nodurile părinte* sunt reprezentate deasupra, sau la stânga (în reprezentarea orizontală) iar *nodurile fiu* sunt imediat sub cele părinte, sau la dreapta (în reprezentarea orizontală).

Orice *arbore de comportament* își începe execuția de la nodul rădăcină ce generează *semnale* care permit execuția nodurilor fiu. Aceste *semnale* se numesc „*ticks*" și au o anumită frecvență dată (de obicei, cea egală cu tick rate-ul jocului, adică de câte ori jocul execută o funcție într-o secundă). Fiecare nod este executat numai atunci când acesta primește *semnal* de la părinte. În primă fază, copilul începe în starea de *rulare*, informând părintele de acest lucru. Dacă execuția a ajuns la un scop (s-a executat cu succes), se întoarce înapoi la părinte cu rezultatul *Succes*, altfel nu s-a putut ajunge la un scop, părintele primește rezultatul *Eșec*.

Acest principiu de execuție al unui BT este unul *condus de cod (code-driven)*, adică, în alți termeni, execuția este bazată pe o implementare funcțională a acestora. Fiecare nod din arbore este o *funcție* care poate să întoarcă unul din cele trei rezultate: *Rulează, Succes, Eșec*.

În formularea clasică, există trei categorii de noduri de control (*Secvență, Selecție / Abandonare, Paralel și Decorator*) și două categorii de noduri de execuție (*Acțiune și Condiție*).

1. *Secvența (fig. 11)*: acest tip de nod execută algoritmul (*code 2*) care corespunde comutării fiecărui tick către copii său unul după altul până când un nod întoarce *Eșec* sau *Rulează* sau până ce toate nodurile întorc *Succes*. Acest nod este reprezentat printr-o săgeată „→"

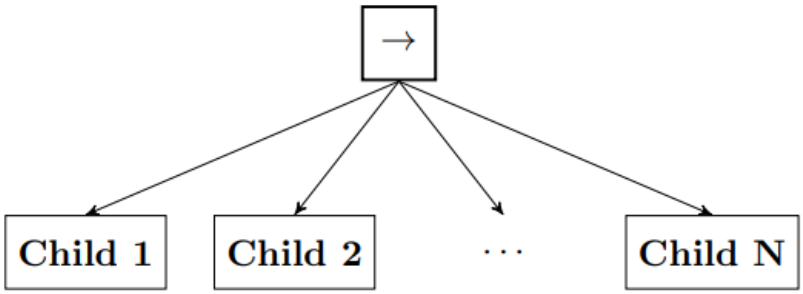


fig. 11 Reprezentare grafica a unei secvențe (Michele & Petter, 2018)

```
BehaviourStatus Sequence(SequenceNode node) {
       foreach(Node child in node.Children) {
              var status = node.Tick();
              if(status == BehaviourStatus.Running)
                     return BehaviourStatus.Running;
              else if(status == BehaviourStatus.Failure)
                     return BehaviourStatus.Failure;
       }
       return BehaviourStatus.Success;
}
```

code 2: Algoritmul unui nod secvență cu o listă de copii. (Michele & Petter, 2018)

2. *Selector (fig. 12):* algoritmul (*code 3*) diferă față de secvență prin faptul că nodul executat informează nodul părinte să se oprească dacă cel puțin un fiu a întors *Succes* sau *Rulează*, altfel dacă toate întorc *Eșec*, și selectorul întoarce *Eșec*. Acest nod este reprezentat prin semnul întrebării „?”

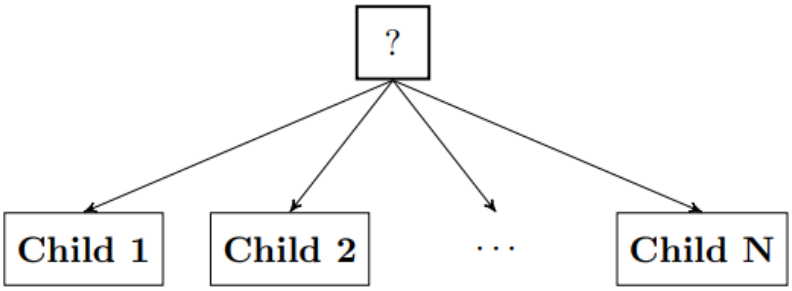


fig. 12 Reprezentare grafică a unui selector (Michele & Petter, 2018)

```
BehaviourStatus Selector(FallbackNode node) {
    foreach(Node child in node.Children) {
         var status = node.Tick();
         if(status == BehaviourStatus.Running)
             return BehaviourStatus.Running;
         else if(status == BehaviourStatus.Success)
             return BehaviourStatus.Success;
    }
    return BehaviourStatus.Failure;
}
```

code 3: Algoritmul de execuție al unui nod fallback (Michele & Petter, 2018)

3. *Paralel (fig. 13):* Acest nod este unul mai special deoarece execută simultan toți fii iar dacă M dintre aceștia au întors *Succes*, întoarce *Succes*, dacă N – M + 1 întorc *Eșec*, întoarce *Eșec*, și altfel întoarce *Rulează*. Următorul algoritm demonstrează modul de rulare. (*code 4*) Acest nod este reprezentat printr-o săgeată dublă „⇒”

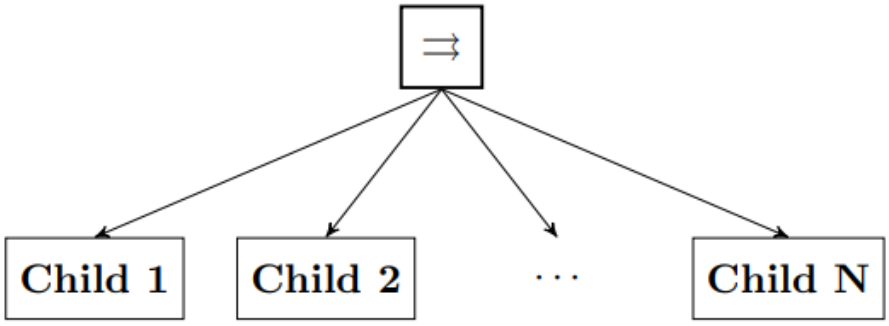
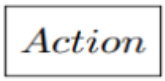


fig. 13 Reprezentare grafică a unui nod paralel (Michele & Petter, 2018)

```
BehaviourStatus Parallel(ParallelNode node, int M, int N) {
       var childStatus = new BehaviourStatus[N];
       for(int i=0; i<N; i++)
              childStatus[i] = node.Children[i].Tick();
       if(childStatus.Sum((status) => status == BehaviourStatus.Success) >= M)
return BehaviourStatus.Success;
       else if(childStatus.Sum((status) => status == BehaviourStatus.Failure) > N – M)
return BehaviourStatus.Failure;
       else
return BehaviourStatus.
}
```

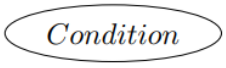
code 4: Algoritmul de execuție al unui nod paralel (Michele & Petter, 2018)

Nodul acțiune, atunci când primește un impuls, acesta execută acțiunea asociată, și în funcție de stare, dacă rulează mult timp întoarce *Rulează*, dacă s-a terminat se întoarce *Succes* și dacă nu s-a putut executa, se întoarce *Eșec*.



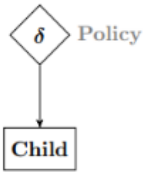
(a) Nod acțiune. Eticheta descrie ce tip de acțiune se execută

În cazul nodului condiție, acesta știe numai să întoarcă Succes dacă condiția este îndeplinită, sau Eșec în cazul în care nu este îndeplinită condiția.



(b) Nod condiție. Eticheta descrie ce condiție este verificată.

Un nod decorator este un nod de control cu un singur fiu ce manipulează statusul întors de fiu în conformitate cu o regulă definită de utilizator și de asemenea execută copilul conform cu aceeași regulă. De exemplu, un „*invertor*” execută fiul și întoarce statusul opus (*Succes* / *Eșec*); un nod „*maxim N încercări*” execută copilul de maxim N ori dacă acesta a întors de fiecare dată *Eșec*, altfel întoarce *Succes* și dacă nu s-a putut atunci se întoarce mereu *Eșec*; un nod „*maxim T secunde*” executa copilul un timp de maxim T secunde înainte ca acesta să se termine, dacă nu s-a terminat acesta întoarce *Eșec*. (Michele & Petter, 2018)



(c) Nod decorator. Eticheta descrie proprietatea după care este alterat statusul nodului copil.

fig. 14 Reprezentarea grafică a nodurilor Acțiune (a), Condiție (b), și Decorator (c) (Michele & Petter, 2018)

Tip nod	Simbol	Succes	Eșec	Rulează
Selector	?	Întoarce *Succes* dacă un fiu a întors *Succes*.	Dacă toți fii eșuează, atunci întoarce *Eșec*.	Dacă un nod a întors *Rulează*
Secvență	→	Dacă toți fii au întors *Succes*.	Dacă un fiu a întors *Eșec*.	Dacă un nod fiu *Rulează*.
Paralel	⇒	Dacă majoritatea fiilor au întors *Succes*.	Dacă majoritatea fiilor au întors *Eșec*.	Altfel
Acțiune	Text	Atunci când se termină de executat.	Dacă este imposibil să se termine de executat.	În timpul execuției
Condiție	Text	Dacă condiția este adevărată	Dacă condiția este una falsă	Niciodată
Decorator	◇	Definit de utilizator	Definit de utilizator	Definit de utilizator

Tabel 1 Tipurile de noduri dintr-un BT (Michele & Petter, 2018)

Exemplu de arbore de comportament pentru jocul Pac-Man

În jocul Pac-Man, implementarea clasică a fantomelor este cea bazată pe automate cu stări finite. Exemplul prezentat aici este unul ce utilizează arborii de comportament pentru modelarea AI-ului folosit de către agentul Pac-Man deoarece este unul simplu, și este prezentat de către autorii cărții despre *Behaviour Trees* (Michele & Petter, 2018).

În acest joc, Pac-Man trebuie să umble printr-un labirint care conține fantome, un număr mare de puncte, respectiv „power dots”. Scopul jocului este consumarea tuturor punctelor fără ca Pac-Man să fie mâncat de către fantome. Punctele mai mari îi oferă lui Pac-Man super puteri temporare și îl face capabil să mănânce fantomele pentru a scăpa de acestea. După un timp, puterea se termină iar Pac-Man devine vulnerabil și poate fi mâncat de fantome iar dacă o fantomă este mâncată pe durata efectului, atunci aceasta se întoarce la start, unde este regenerată și redevine periculoasă. Fantomele ce pot fi mâncate de Pac-Man sunt albastre și atunci când efectul s-a terminat acestea pâlpâie pe ecran, semnalând că sunt pe cale să redevină periculoase.

fig. 15 Jocul Pac-Man pentru care se va realiza un BT (Pac-Man, 2021)

O simplă abordare ar fi lăsarea lui Pac-Man să ignore fantomele și să fie atent numai pe mâncatul de puncte. Aceasta este implementată folosind metoda *greedy* într-un singur nod acțiune intitulat „*Mănâncă puncte*”. (*fig. 16*)

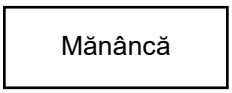


fig. 16 BT pentru cel mai simplu comportament "Mănâncă" (Michele & Petter, 2018)

Pentru a ține cont de fantome este necesară construirea unui arbore complet. Pentru aceasta, se vor utiliza nodurile *selector* și ***secvență*** și o condiție ce verifică dacă fantoma este prin apropiere. Dacă în urma verificării condiției se întoarce *Succes* atunci Pac-Man trebuie să evite fantoma. Prin evitare se înțelege că distanța dintre Pac-Man și fantomă trebuie să fie maximizată. Arborele rezultat (*fig. 17*) va încerca să schimbe între acțiunile de a mânca, respectiv de a fugi de fantome în funcție de rezultatul întors de condiție.

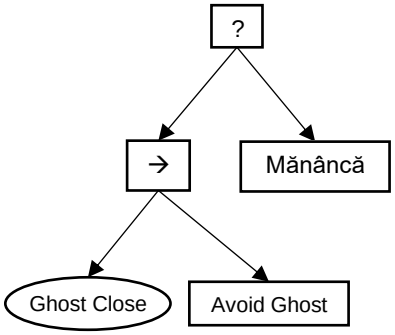


fig. 17 Dacă o fantomă este aproape, arborele va executa acțiunea „Avoid Ghost" (Michele & Petter, 2018)

Ultima extensie făcută asupra arborelui (*fig. 18*) este aceea de a ține cont de „*power dots*”. Deoarece acestea sunt importante, Pac-Man trebuie să fie atent după ele. Atunci când o fantomă este aproape, Pac-Man poate să scape de fantome mâncând un astfel de punct. Dacă Pac-Man face acest lucru fantomele devin vulnerabile și el poate să fugă după ele.

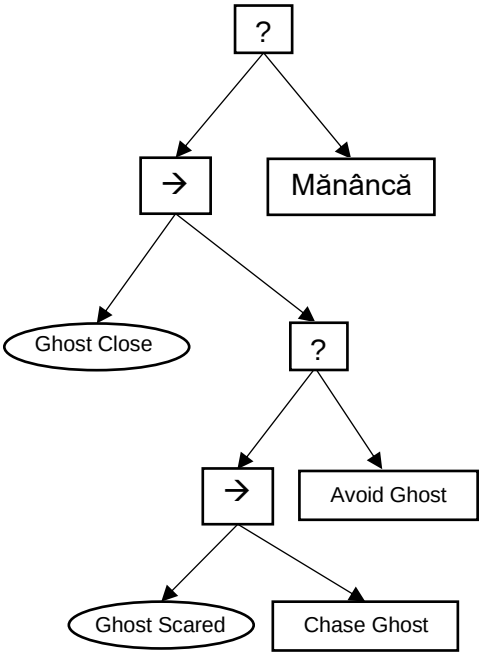


fig. 18 Arborele de comportament complet (Michele & Petter, 2018)

2.2.2 UTILITY AND STOCHASTIC BEHAVIOUR TREES

Arborii de comportament bazați pe utilitate (Utility-Based Behaviour Trees) au apărut din necesitatea de a cunoaște corect când trebuie să aibă loc trecerea de la o acțiune la alta. De exemplu, în comportamentul unui hoț care sparge o casă, dacă acesta este urmărit de poliție, de unde știe dacă este bine să fugă sau să lupte cu poliția? Această decizie este foarte dependentă de circumstanțe: *avem un vehicul cu care să fugim?*, *avem o armă la dispoziție?*, *câți oponenți sunt prezenți?*, etc. (Michele & Petter, 2018)

O abordare ar fi cea a unei *cozi cu priorități*, dacă un fiu al nodului *selector* întoarce utilitatea așteptată de acesta, atunci acesta va reține acțiunea cu prioritate maximă. În momentul în care o altă acțiune a adus la informare că utilitatea este disponibilă, prima acțiune aflată în coadă va fi executată. Fiecare valoare a priorității unei acțiuni este normalizată într-un interval [0, 1] ce reprezintă probabilitatea ca agentul inteligent să execute acea acțiune. O astfel de abordare nu este întotdeauna ușor de implementat. Una dintre puterile fundamentale ale unui BT o reprezintă modularitatea acesteia. Dar cum se poate implementata utilitatea într-un astfel de arbore și acesta să rămână modular?

O strategie ar fi, adăugarea de noduri de tip decorator care calculează simultan utilitatea nodului fiu. Aceste noduri decorator se numesc „*noduri serviciu*" (service nodes) și ele permit calculul periodic al unei informații ce stă la baza executării nodului fiu dar nu modifică rezultatul acestuia. Această strategie a fost implementată de către *Epic Games* în arborii de comportament incluși în *Unreal Engine 4* lansat în anul 1998.

O altă metodă este aceea de a adăuga o ***estimare a utilității*** pentru fiecare acțiune din subarbore, și de a crea o modalitate de a propaga utilitatea de a lungul arborelui, atât nodurilor *selector*, dar și a celor ***secvență***. Modalitatea este una mai dificilă de implementat, deoarece trebuie știut cum se face această propagare. Toate aceste dificultăți au condus la dezvoltarea *arborilor de comportament stocastici*.

Arborii de comportament stocastici sunt o variație a ideii de utilități și este aceea de a ține cont doar de probabilitățile ca un nod să întoarcă succes. Dacă ceva trebuie efectuat în timp util, atunci acțiunea cu cea mai mare probabilitate de succes va fi executată prima. Trebuie ținut cont de faptul că atât costurile cât și timpii de execuție sunt la fel de importanți.

Folosind probabilitatea de a întoarce *succes*, atunci *agregarea* dea lungul ***secvențelor*** respectiv *selectoarelor* este, în teorie, evidentă. Considerând P_i^S ***probabilitățile nodurilor*** unui arbore de a întoarce *succes*, atunci, ***probabilitatea totală*** a sub arborelui este calculată după: (***Ecuație 1***)

$$P^S_{Secventa} = \prod_i P_i^S, \text{ respectiv } P^s_{Selector} = 1 - \prod_i (1 - P_i^s).$$

Ecuație 1 Formula de agregare a probabilităților de a se executa cu succes un nod compus (Michele & Petter, 2018)

Unde variabilele $P^S_{Secvență}$ *și* $P^S_{Selector}$ reprezintă probabilitatea ca un nod ***Secvență***, respectiv *Selector* să întoarcă *Succes*.

Calculul probabilităților P_i^S pentru fiecare nod fiu va folosi ***Ecuație 2*** a ***probabilității empirice***. Pentru fiecare nod executat, se numără de câte ori acesta a întors *Succes* și se împarte la numărul total de încercări, astfel se vor executa nodurile cu probabilitatea cea mai mare înaintea executării celorlalte.

$$P_i^s = P_i(S) = \frac{nr\ succese}{nr\ încercări}.$$

Ecuație 2 Calculul probabilității empirice a unui nod fiu de a întoarce Succes (Michele & Petter, 2018)

Această abordare nu este tocmai foarte bună pentru a optimiza execuția fiilor importanți ai unui *nod compus* (*selector* sau ***secvență***) deoarece dacă există mai mulți fii a căror probabilitate este cea mai mare aceștia pot conduce la o problemă de tip *exploatare* / *explorare* a spațiului de căutare. În acest caz, optimizarea ajunge să rămână blocată într-un maxim local favorizând execuția aceluiași nod fără să se mai exploreze și restul spațiului de căutare.

2.2.3 EVENT-DRIVEN BEHAVIOUR TREES

Autorii articolului din jurnalul „*Expert Systems with Applications"* (Ramiro, Sebastian, & Alejandro, 2019) au dezvoltat o extensie a arborilor de comportament, o metodă bazată pe culegerea de informații din mediu, numite și *evenimente* (***utilități***), care permit interacțiunea dintre mai mulți agenți inteligenți.

În timp, arborii de comportament au evoluat în *arbori **bazați pe utilitate***, *stocastici* și abia apoi au devenit ***bazați pe evenimente***. Necesitatea implementării comportamentelor complexe au impus nevoia de construire a unor arbori din ce în ce mai adânci. Soluția găsită a fost cea de a implementa noduri care răspund la evenimente și renunță la nodurile care rulează. Dacă nodurile ce rulează se execută mai mult timp și are loc un anume *eveniment* (ex: lupul începe să atace iepuri) atunci renunțarea acțiunii curente (iepurele se plimbă) se face mult mai complicat. Prin folosirea de noduri speciale, aceste acțiuni pot fi întrerupte imediat ce un eveniment a avut loc.

Această abordare a permis realizarea de BT mult mai ușor, permițând dezvoltatorilor să evite problemele de performanță a BT clasici.

Noua implementare a acestora este bazată pe *arborii de comportament stocastici*, dar nu varianta bazată pe probabilități de execuție, ci cea bazată pe noduri decorator asociate fiilor nodului. Aceste noduri execută periodic o funcție care calculează necesitatea execuției nodului decorat. Dacă acea funcție a condus la o informație utilă (*utilitate*) atunci nodul fiu va fi executat iar statusul acestuia va fi propagat mai departe. Aceste noduri se numesc *noduri serviciu* deoarece acestea determină *utilitatea* ca pe un *serviciu*. Când un astfel de nod este executat, acesta rulează periodic atâta timp cât nodul fiu se execută.

De exemplu, în cazul în care un iepure se plimbă prin lume, acesta este atent după posibili prădători (lupi). Dacă aceștia sunt prin preajmă atunci el recunoaște acest eveniment și știe că trebuie să fugă de ei. Atenția iepurelui este dată de o ***funcție serviciu*** asociată acțiunii de plimbare, dacă acțiunea este executată iar utilitatea nodului scade (prin faptul că serviciul constată că există prădători) atunci nodul fiu este abandonat, favorizat fiind următorul nod, cel de fugă.

Pentru ca un nod serviciu să poată să fie util este necesară implementarea unei structuri tip *blackboard*. Această structură este un ***dicționar*** (sau ***tabelă de dispersie***) a căror chei sunt *atribute* evaluate de servicii iar valorile sunt *obiecte* întoarse de aceste servicii. Fiecare *cheie atribut* are o singură *valoare* asociată. Acest *blackboard* este o structură privată asociată unui BT. Pe parcursul acestei lucrări, fiecare cheie este un șir de caractere, o accesare a informației este făcută prin *blackboard[key]* iar valoarea întoarsă este una obținută prin apelul unei metode serviciu (ex, iepurele are un serviciu *FindEnemy* care este executată periodic și a cărei valori întoarse este salvată în *blackboard*[„target"])

2.2.4 ASYNCHRONOUS BEHAVIOUR TREES

Sunt o extensie relativ nouă a arborilor de comportament bazați pe evenimente. Această abordare presupune ca fiecare nod să fie modelat folosind o structură de tip ***sarcină***. Această ***sarcină*** este una care conduce la un *obiectiv* și poate fi una ***compusă*** (de exemplu un sub arbore) sau una ***simplă*** (de exemplu o acțiune). Fiecare ***sarcină*** este executată pe baza unui automat cu stări finite în care fiecare stare reprezintă o promisiune efectuată metodei apelant că respectivul cod va fi executat atunci când este nevoie de rezultatul obținut (exemplu: Nodul părinte va știi sigur ce decizie să facă

atunci când sarcina executată a ajuns la un rezultat *Succes* sau *Eșec*). Această abordare deschide posibilitatea de răspuns la evenimente mult mai rapid deoarece renunțarea la o sarcină se va realiza imediat prin abandonarea sarcinii de către nodul părinte (trebuie ținut cont că informarea acestei abandonări se face de către nodul fiu în funcție de condiția de oprire).

Implementarea mea folosește elemente de programare în limbajul C# bazate pe paradigma *async / await* și cea funcțională (*delegates*).

Sintagma *async* / *await* este modalitatea acestui limbaj de a scrie *funcții asincrone*. O *metodă asincronă* este o funcție ce în urma apelului aceasta nu este executată decât abia atunci când sarcina întoarsă de aceasta este pusă în execuție în conformitate cu un automat cu stări finite. La momentul apelări *„await task"* are loc o tranziție de continuare de la o stare la alta și executarea sarcinii respective. Dacă aceasta nu și-a terminat execuția, automatul rămâne în starea curentă așteptând după aceasta să se termine. Un *task (sarcină)* este o promisiune făcută metodei apelant că rezultatul întors de o funcție asincronă este obținut atunci când sarcina va fi executată cu succes și nu au apărut excepții de cod. Acest lucru permite scrierea într-o manieră elegantă a cozilor de execuție.

Un *delegate* este un contract realizat între programator și compilator că orice metodă asociată acesteia va respecta prototipul declarat. Ele sunt o manieră funcțională de a trimite metode altora fără ca acestea să fie apelate decât atunci când cea care a primit un *delegate* a decis acest lucru.

Tot o dată, pentru a putea comunica atât în arbore, dar și între agenți, s-a implementat și modelul de programare POO, intitulat *Mediator Pattern*. Acesta folosește o clasă intermediară, intitulată *mediator* care permite trimiterea de *mesaje* între obiectele abonate (agenții inteligenți). În acest fel este stabilit un *canal de comunicație*.

2.3 AVANTAJELE UTILIZĂRII BEHAVIOUR TREES

Utili nu numai pentru dezvoltarea de *personaje necontrolabile de jucător (NPC)* dar și pentru implementarea de *roboți*, fie ei casnici, sau industriali, *arborii de comportament* au permis dezvoltarea accelerată a inteligenței artificiale pe mult mai multe nișe de dezvoltare.

Avantajele arborilor de comportament sunt numeroase, printre acestea amintim următoarele posibile cazuri de utilizare.

1) *Modularitatea* a permis dezvoltatorilor să poată scrie module specializate pentru anumite sarcini care permit planificarea de scopuri executate de către un agent inteligent. Modulele utilizate de către arborii de comportament pot fi: *nodurile compuse* care definesc variantele de a acționa inteligent astfel încât să se ajungă la același scop, *nodurile decorator* ce permit influențarea luării deciziilor de către agent în funcție de parametrii și condițiile din mediu, și *nodurile acțiune* ce execută o acțiune efectivă care conduce la scopul căutat.

2) *Eficiența în dezvoltare* conduce la scrierea și modificarea rapidă a codului. Dacă un agent inteligent nu face ce trebuie, aplicând metoda automatelor cu stări finite, ar însemna izolarea stării în care se ajunge și determinarea tuturor tranzițiilor prin care se poate trece la o altă stare. Astfel spus, găsirea de probleme în cod este una foarte dificilă de către dezvoltator deoarece nu se rezumă la modificarea stării respective, dar și la determinarea tuturor acțiunilor pe care un agent le poate executa în acel moment. Arborii de comportament, prin natura lor, permit identificarea rapidă a problemei și rezolvarea acesteia. O altă posibilă modificare poate însemna definirea de acțiuni complexe, construite sub formă de sub arbori, rezolvarea problemelor aferente, și reutilizarea acesteia în arbore.

3) *Utilitatea* unui arbore de comportament înseamnă cât de bine se pretează pentru rezolvarea problemei. Un BT este necesar și util în cazurile în care problema ce se încearcă a fi rezolvată presupune atingerea unui scop de către un agent inteligent. În jocurile video, orice NPC își

are un scop predefinit. De exemplu, în jocurile de tip *shooter* un agent inteligent trebuie să bată, fără a pierde viată, jucătorul. Arborele de comportament al unui astfel de agent poate determina acțiunile care trebuie făcute într-o situație astfel încât scopul acestuia să fie atins, acela de a elimina jucătorul.

4) *Execuția rapidă a sarcinii* unui agent este un avantaj foarte mare a arborilor de comportament. Implementarea automatelor cu stări finite, de obicei ineficiente, conduc la pierderea de cicluri de execuție inutile deoarece dacă un agent inteligent poate ajunge într-o stare din care nu se mai poate ieși, acesta produce apariția de erori în cod și la rulări ineficiente pentru a ajunge la scop. Un arbore de comportament, prin intermediul *nodurilor decorator*, poate să renunțe imediat la acțiuni ce durează mult și nu conduc la o utilitate în acel moment. Abordările precum cea ***stocastică*** sau cea ***bazată pe evenimente*** răspund imediat la impulsuri neașteptate din mediu și cunosc exact ce acțiune este mult mai utilă. Implementarea asincronă, dacă este realizată corect, poate și ea să conducă la răspunsuri naturale ale agenților inteligenți.

Deși, arborii de comportament sunt foarte avantajoși în momentul actual, agenții inteligenți bazați pe scop, pot fi cu mult mai buni și mai inteligenți dacă se dezvoltă noi algoritmi care să permită acest lucru atât în jocurile video, cât și pentru roboți. În continuare se vă prezenta dezavantajele arborilor de comportament în comparație cu alte procedee.

1) *Agenții inteligenți, deși inteligenți, nu sunt foarte inteligenți.* Este clar că arborii de comportament permit scrierea de agenți ce sunt considerați ca fiind inteligenți, dar, nu întotdeauna comportamentul acestora este cu adevărat inteligent. De exemplu, folosind BT, un agent inteligent răspunde la un eveniment pe moment, dacă acesta nu este relevant atunci un agent va renunța complet la informație. Pentru a putea salva, pe o durată mai mare de timp, informațiile adunate din mediu, un agent inteligent trebuie să învețe și să fie capabil să refolosească acea informație. Arborii de comportament nu permit salvarea și reutilizarea informațiilor pentru luarea deciziilor. O abordare diferită care ar putea permite acest lucru ar fi utilizarea rețelelor neuronale artificiale. Acestea, împreună cu învățarea automată prin întărire, permit influențarea luării deciziilor pe baza experienței dobândite anterior de către un agent inteligent.

2) *Arborii de comportament, deși încearcă să implementeze agenți inteligenți bazați pe utilitate, nu reușesc în de ajuns să o facă.* Revenind la diferența fundamentală între un agent inteligent bazat pe scop și unul pe utilitate. Agenții inteligenți bazați pe scop sunt folosiți pentru probleme de căutare și atingere de scop. Arborii de comportament sunt cei mai buni în a face acest lucru. De-a lungul timpului s-au încercat dezvoltări de BT bazați pe utilitate dar nu s-a reușit.

Un agent bazat pe utilitate este utilizat în cazurile în care este necesară menținerea unor acțiuni care aduc la o bunăstare agentului inteligent. Aceste acțiuni sunt considerate utile deoarece sunt relevante într-un moment al execuției. Un BT folosește o abordare stocastică pentru a determina utilitatea unei acțiuni. Dacă aceasta aduce rezultatul *succes* de fiecare dată ce este rulată, atunci agentul inteligent are o probabilitate mai mare de a o refolosi. Alți algoritmi de inteligență artificială descoperă utilitatea acțiunilor din experiențe anterioare, dar un BT nu poate realiza acest lucru, el neavând capabilitatea de a reaminti aceste experiențe, având o memorie de moment, imediat ce informația nu mai este utilă, aceasta este ștearsă.

3) *Implementarea acestora este una foarte complexă.* Implementarea lor poate devenii foarte complicată folosind programarea secvențială. Pentru a garanta o funcționalitate completă a arborilor de comportament este necesară rularea în paralel a generării și traversării *tick-urilor,* respectiv executarea acțiunilor. Deși implementată o dată, aceasta poate fi refolosită de mai multe ori, iar implementări există deja pe piață.

CAPITOLUL 3 DEZVOLTAREA APLICAȚIEI EVOAGENTS

Următorul capitol prezintă modul de dezvoltare a aplicației EvoAgents. Această aplicație este o bibliotecă de tip API (interfață de programare a aplicațiilor) ce permite dezvoltarea de arbori de comportament (BT) pentru personaje controlate de calculator. Inteligența artificială este într-o dezvoltare rapidă, așa că necesitatea unei astfel de librării de clase este una foarte importantă pentru societate deoarece permite scrierea ușoară de algoritmi pentru agenți inteligenți.

În continuare se vor prezenta arhitectura aplicației și modul în care aceasta a fost realizată și testată, iar în capitolul următor se va prezenta utilizarea acesteia pentru a realiza agenți inteligenți bazați pe scop ce folosesc BT.

3.1 ARHITECTURA APLICAȚIEI

Aplicația EvoAgents este alcătuită din două module principale, o librărie API care permite scrierea de arbori de comportament în limbajul de programare C# și un joc video demonstrativ (*demo*) care se folosește de acest modul pentru a implementa doi agenți inteligenți, un iepure (*Rabbit*) și un lup (*Wolf*). Acești agenți se află într-o lume generată aleator și jucătorul trebuie să încerce să prindă iepuri astfel încât lupul să nu îi atace.

Această aplicație, conform *fig. 19*, are la bază pachetul „*EvoAgents.Behaviours*”, care folosind pachetul „*UniTask*”, o implementare open source, licențiată MIT, aceasta implementează arbori de comportament asincroni. *UniTask* este o librărie dedicată implementării de funcții asincrone specializate pentru Unity. Clasele oferite de limbajul C#, *Task* și *Task*<*T*>, sunt mult prea dezvoltate pentru a implementa un astfel de cod. Acest pachet folosește o metodă agreată de Unity pentru a scrie cod asincron, single-threaded (pe un singur fir de execuție).

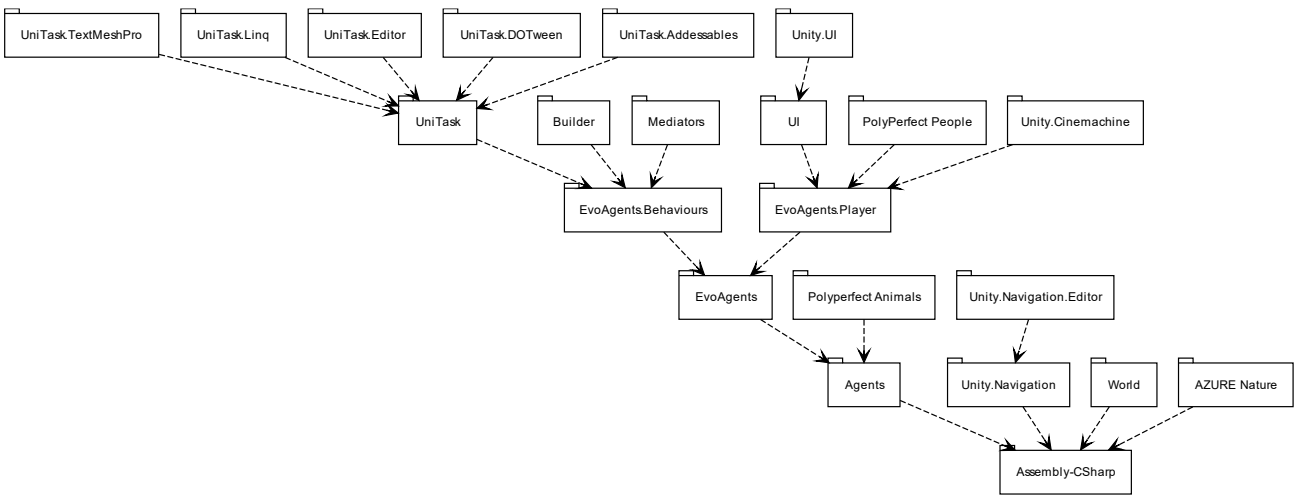


fig. 19 Diagramă de pachete care compun aplicația EvoAgents

Această metodă constă în simularea unei bucle principale de execuție, intitulată „*PlayerLoop*”, fiecare task fiind programat să se execute în această buclă (CySharp - UniTask, 2021). Prin eficiența și limitarea alocărilor de memorie, biblioteca *UniTask* integrează sintagma *async/await* oferită de limbajul C# în Unity.

- Folosește o structură *UniTask*<T>, aceasta fiind alocată în stiva de execuție a funcției asincrone, nu are loc alocări de memorie în *heap*, ceea ce permite o execuție mult mai rapidă a codului
- *Unity* folosește corutine, o formă bazată pe enumerator de a scrie cod asincron, această bibliotecă vine ca o completare astfel încât acestea să pot fi așteptate.

- Fiecare instrucțiune de tip *Yield* (*WaitForEndOfFrame* sau *WaitForSeconds*) devine o instrucțiune de așteptare, aceasta fiind simulată pe un *PlayerLoop* implementat de către bibliotecă
- Rulând peste un *PlayerLoop*, biblioteca nu folosește programare paralelă, ci doar una concurentă, fiecare sarcină fiind executată de îndată ce alta își termină execuția.

Pentru restul pachetelor, de exemplu generarea lumii, m-am folosit de cele de bază oferite de Unity: *Navigation* (folosește algoritmul A* pentru a căuta drumuri, respectiv implementarea agenților autonomi), *Jobs* (scrierea de cod paralel, intitulate *jobs*, pentru generarea rapidă a lumii) și *Cinemachine* pentru tranziția și controlul camerelor video a unei drone respectiv cea a jucătorului.

Pachetele *PolyPerfect* (*Animals* și *People*) conțin elemente grafice 3D pentru animale și pentru personajul principal, iar pachetul *AZURE Nature* este folosit pentru generarea unui mediu în care agenții să interacționeze)

3.2 AGENȚI INTELIGENȚI: TIPURI ȘI INTERACȚIUNI

Aplicația *EvoAgents – Rabbit Catcher* (fig. 20) este alcătuită din doi agenți inteligenți bazați pe utilitate, un iepure (*RabbitAgent*) și un lup (*WolfAgent*). Acești agenți sunt generați într-un număr aleator și folosesc o implementare a arborilor de comportament bazați pe evenimente.

Diagrama următoare (*fig. 20*) prezintă structura modulelor implementate în acest proiect. Fiecare modul reprezintă o clasă asociată jocului *Rabbit Catcher*, joc video demonstrativ format dintr-o lume generată aleator și doi agenți inteligenți bazați pe BT.

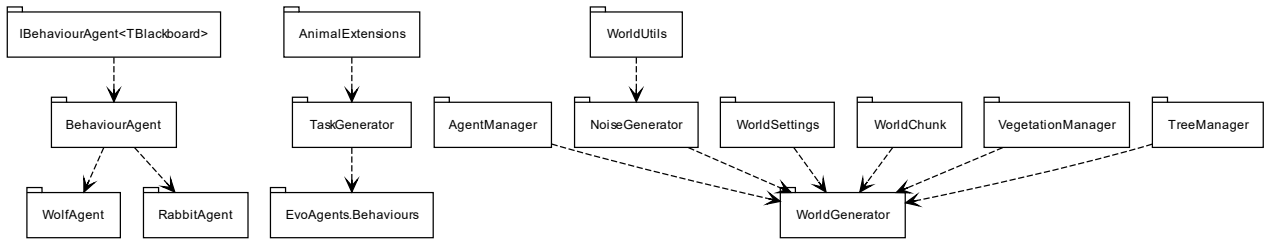


fig. 20 Diagramă de module pentru jocul EvoAgents - Rabbit Catcher

IBehaviourAgent<TBlackboard> este o interfață care folosește un *Blackboard* pentru a implementa un *BehaviourAgent*. Un *Blackboard* este o structură de date care alcătuiește memoria unui agent inteligent. Aceasta este bazată pe un dicționar de obiecte ce reprezintă o informație aflată din mediu. Pentru ca un agent să fie inteligent, acesta trebuie să fie unul autonom, care pe baza unor informații culese din lume să fie capabil să învețe și să ia decizii. Această structură de memorie reprezintă elementul care definește agenții ce implementează arbori de comportament ca fiind agenți inteligenți.

Modulul *AnimalExtensions* conține extensii de comportament pentru agenții inteligenți. Acestea sunt de fapt niște acțiuni pe care numai un animal le poate realiza. Iar *WorldGenerator* permite generarea de lumi în care agenții inteligenți să poată interacționa.

3.3 *ANALIZĂ ŞI PROIECTARE*

În această secțiune se prezintă analiza problemei și abordarea aleasă pentru implementarea soluției. Problema cere dezvoltarea unui sistem care să permită scrierea de arbori de comportament, interacțiuni între agenți și moduri de comunicare între aceștia. Sunt necesari ca agenții inteligenți să poată coopera unii cu ceilalți și ca aceștia să poată reacționa la evenimente din mediul în care aceștia se află.

3.3.1 WOLF BEHAVIOUR TREE VS. RABBIT BEHAVIOUR TREE

În *fig. 21* este reprezentat arborele de comportament al unui agent inteligent, iepure. La începutul existenței agentului, el trebuie să cunoască că are energie suficientă și că nu există inamici. Această inițializare se realizează secvențial setând informațiile utile într-un dicționar apoi este pornit arborele de comportament principal.

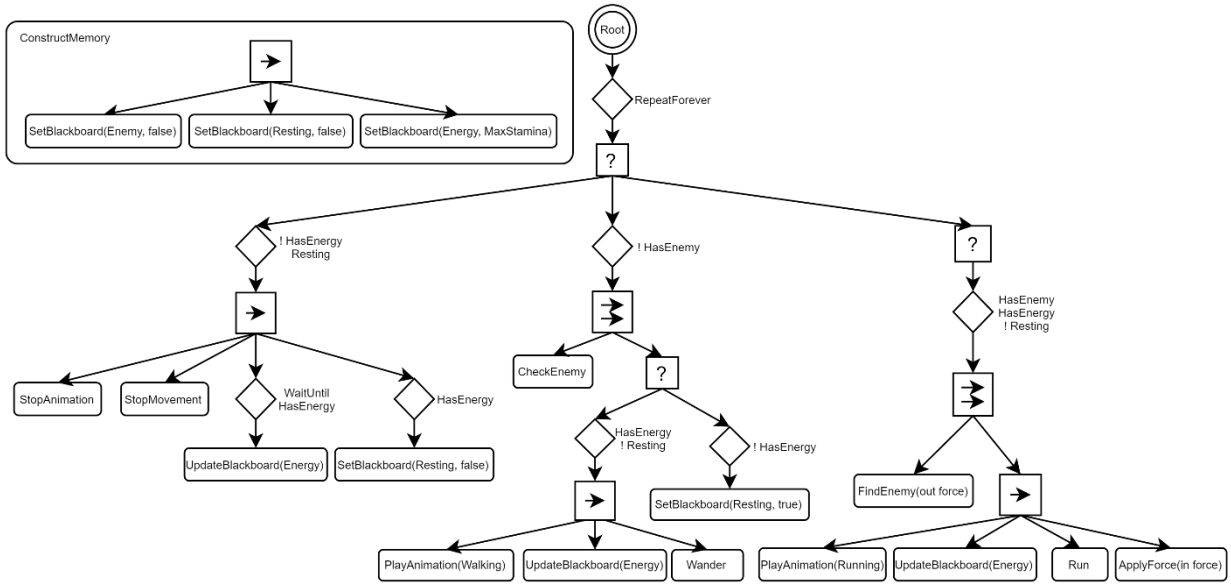


fig. 21 Behaviour Tree pentru iepure. Construct Memory este un arbore dependent care setează variabile pentru arborele principal.

Acest arbore folosește ca și rădăcină un decorator ce repetă la infinit, sau până la oprirea agentului, un nod care selectează una din cele trei acțiuni posibile: ***odihnă***, *plimbare* respectiv ***fugă***. Fiecare acțiune este executată dacă și numai dacă condiția asociată este îndeplinită. Un iepure se odihnește dacă nu are energie; asta înseamnă că pas cu pas, este oprită animația și nu se mai deplasează apoi execuția este blocată până când acesta recapătă energie. În tot acest timp este reactualizată, iar dacă și-a recăpătat-o, atunci nu se mai odihnește. Având cum să se plimbe, dacă nu există inamici prin preajmă, în paralel se verifică dacă există și se plimbă dacă are energie, altfel intră în starea de odihnă. Dacă are energie, este rulată o animație de mers și pierde energie mergând într-o destinație aleasă aleator. În final, dacă în timpul plimbării iepurele a văzut un lup, sau a simțit jucătorul că se află în apropierea acestuia, atunci este anunțat public evenimentul și intră pe ultima ramură. Dacă are energie și știe că există inamici prin apropiere, în paralel se caută inamicii și se calculează o forță utilă ce reprezintă direcția în care agentul trebuie să fugă pentru a evita să fie prins. În secvență, iepurele începe să alerge pierzând energie într-o direcție aleasă la întâmplare și care este influențată de forța de evitare a inamicilor. În final dacă acesta a scăpat și nu mai are energie ciclul se repetă și reintră în starea de odihnă.

Forța de evitare (*Ecuație 3*) este calculată folosind formulele ce definesc un comportament de evitare (*steering behaviour*). Formula este utilizată pentru calcularea forței ce trebuie aplicată agentului, văzut ca un vehicul, pentru ca acesta să poată evita inamicul ce se află în apropierea acestuia. Variabilele p_f și p_{enemy} reprezintă poziția finală respectiv actuală a inamicului iar v_{enemy} reprezintă vectorul viteză care arată în ce direcție se îndreaptă inamicul. Calculul vitezei dorite de către agentul care evită se realizează în funcție de poziția inamicului și agentului normalizată la viteza de deplasare a agentului (variabila s). În final, forța $\overrightarrow{st}$, calculată sub formă de sumă de vectori (diferență între viteze) a fiecărui inamic, reprezintă forța de evitare.

$$p_f = p_{enemy} + v_{enemy};\ v_d = -1 * s * normalized(p_f - p);\ st = \sum_{enemy} (v_d - v)_{enemy}$$

Ecuație 3 Calculul forței de evitare a inamicilor

Arborele de comportament al unui lup (*fig. 22*) este diferit față de iepure prin limitarea comportamentală a acestuia. Decoratorul probabilitate este unul *stocastic* și acesta determină probabilitatea ca un lup să fie unul agresiv. Un agent agresiv este atunci când acesta începe să atace un alt agent inteligent. Dacă acesta a găsit o pradă el determină o forță de hărțuire cu ajutorul căreia acesta are o atracție după pradă. El începe să fugă după ea și să o prindă atacând-o.

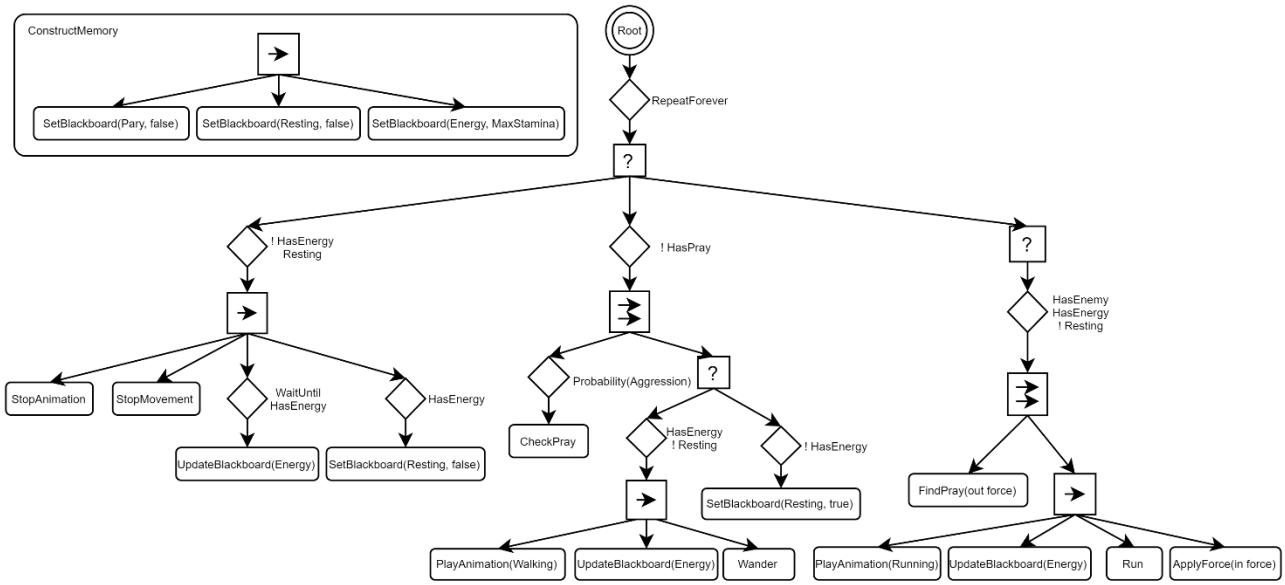


fig. 22 Behaviour Tree pentru lup. Diferența o face nodul decorator „Probability" atașat asupra nodului ce verifică dacă există pradă.

Forța de hărțuire este calculată asemănător cu cea de evitare determinată de iepure. Acest calcul se realizează folosind următoarele formule (*Ecuație 4*):

$$p_f = p_{pray} + v_{pray};\ v_d = +1 * s * normalized(p_f - p);\ st = \sum_{pray} (v_d - v)_{pray}$$

Ecuație 4 Formula forței de hârțuire a unui agent inteligent lup.

3.3.2 INTERACȚIUNEA ÎNTRE AGENȚI

Generați în lume (*fig. 23*), iepurii și lupii se plimbă liniștiți. Aceștia interacționează între ei prin intermediul unui *mediator* și *blackboard*. Fiecare agent iepure verifică dacă există inamici, lupi, iar ceilalți caută pradă, iepure, în lume. Dacă s-au găsit inamici sau pradă, lumea informează agenții de acest lucru ca apoi să își trimită mesaj între ei despre acest fapt. Mesajul este trimis printr-un intermediar iar arborele de comportament, pe baza unui *blackboard,* îi atenționează pe agenți că trebuie să fugă unul de celălalt. În cazul în care un lup a prins un iepure acesta este atacat și moare. Dacă iepurele a murit, este șters din lume iar lupul începe să se plimbe din nou.

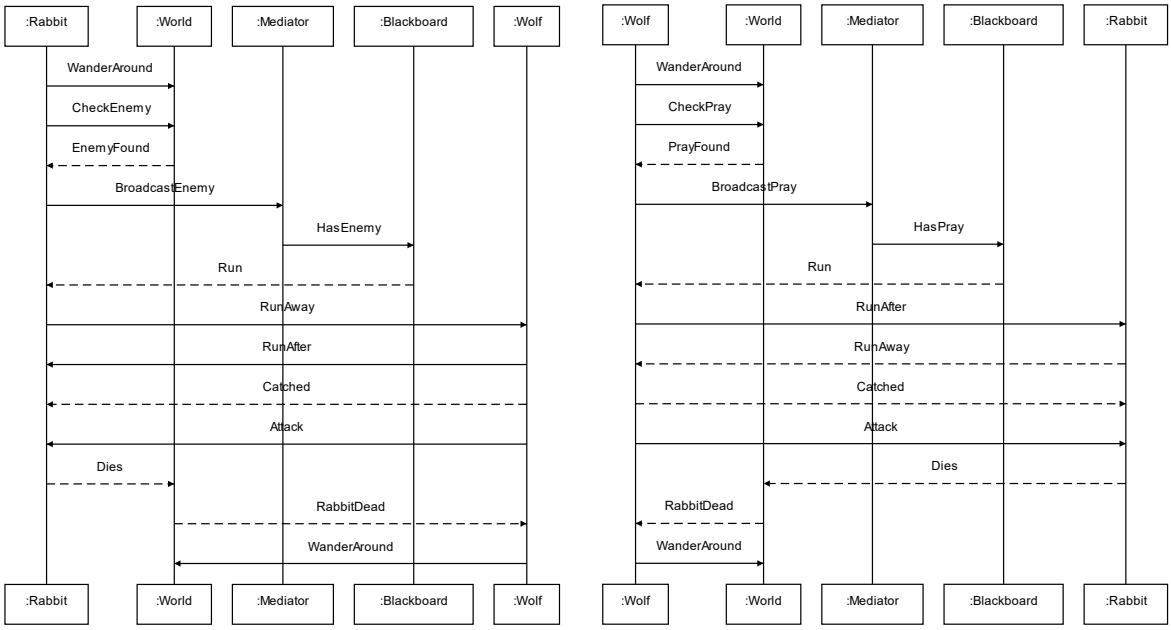


fig. 23 Interacțiunile între iepuri și lupi

Există un caz particular (*fig. 24*) în care iepurele reușește să scape de lup. Dacă scapă atunci revin la stadiul de plimbare până obosesc. Dacă un agent este obosit acesta se oprește și intră în starea de odihnă.

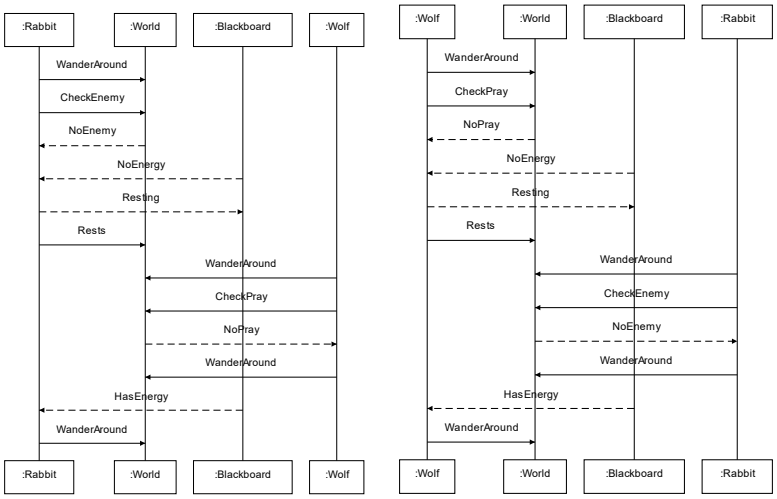


fig. 24 Interacțiunea dintre agenți în cazul în care unul este obosit și vrea să se odihnească

3.3.3 ARHITECTURA PACHETULUI EVOAGENTS BEHAVIOURS

În următoarea diagramă (*fig. 25*) se poate observa împărțirea pe module a pachetului API. Fiecare modul expune dezvoltatorului câte o funcționalitate ce poate fi utilizată pentru a implementa arbori de comportament și agenți inteligenți.

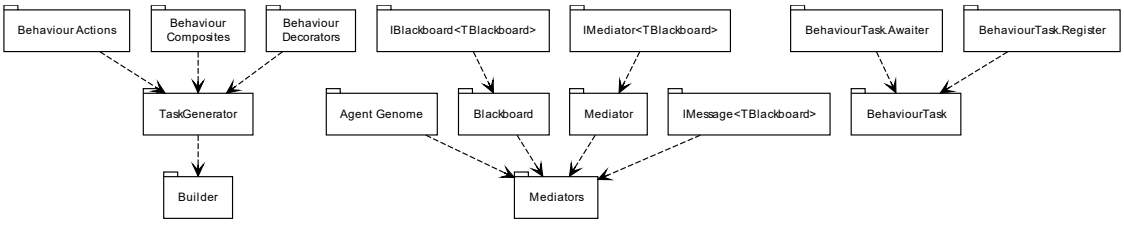


fig. 25 Diagramă de module pentru EvoAgents API

Modulul *Builder* folosește un model de implementare POO care permite generarea de arbori de comportament într-o manieră bazată pe înlănțuirea de apeluri către metode tip extensie ce construiesc noduri ce stă la baza arborelui. Acest procedeu se numește *Factory Pattern* și are la bază abordarea „*Fluent*" adesea întâlnită în *Web Development* pentru configurarea de obiecte folosite pe partea de *backend (server)*. Modulul *TaskGenerator* este o clasă care permite generarea de obiecte de tip *BehaviourTask,* sau BT. Acest modul este compus dintr-o colecție de metode extensie ce implementează noduri de tip acțiune, compuse respectiv decoratoare.

Un agent inteligent folosește un BT, ce reprezintă un *BehaviourTask* compus, obiect ce execută o sarcină (ce reprezintă scopul agentului) într-o manieră asincronă (implementând un *Awaiter* ce reprezintă un tip de obiecte care controlează execuția sarcinii) și este extinsă prin două metode care permit înregistrarea de noi subsarcini fiu ale sarcinii părinte.

Memoria agentului inteligent este alcătuită dintr-o structură *blackboard* care este un dicționar de atribute învățate din mediu și care definesc deciziile ce vor fi luate de către agent. Comunicarea dintre agenți se realizează folosind implementarea unui *Mediator Pattern*. Un *Mediator* reprezintă o clasă intermediară ce transmite mesaje între obiectele abonate la aceasta. Prin acest mod se pot implementa evenimentele ce pot apărea într-un mediu

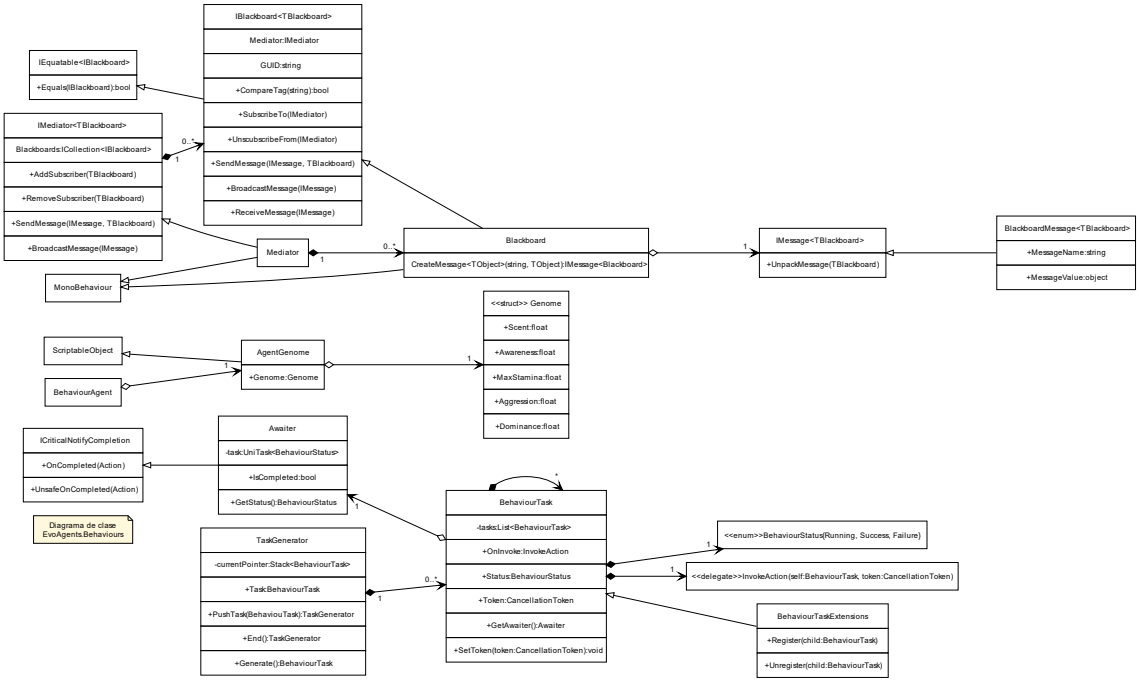


fig. 26 Diagramă de clase pentru biblioteca EvoAgents API

Mediator Pattern, comunicarea între agenți, este implementat pe baza următoarelor clase și interfețe. Interfață *IBlackboard* este o extindere a interfeței C# *IEquatable* deoarece pentru a realiza o distincție între agenții abonați la un mediator este necesară scrierea unei funcții ce verifică dacă două obiecte sunt egale.

Un *IMediator* deține metode care permit comunicare și o colecție de zero sau mai mulți abonați de tip *IBlackborad*. Clasele *Blackboard* și *Mediator* implementează aceste interfețe și extind un *MonoBehaviour* (o clasă specială Unity care permite atașarea de componente la un *GameObject*).

Orice *Blackboard* returnează un mesaj de tip *BlackboardMessage* ce implementează interfața *IMessage*. Acest mesaj este o structură ce conține un nume de mesaj (*atribut*) și o valoare generică de tip *obiect*.

Un agent inteligent are și un set predefinit de atribute unice intitulat *AgentGenome*. Această structură este de tip *ScriptableObject*, tip de date specializat ce permite salvarea și refolosirea de informații de către unul sau mai multe *GameObject*.

Un arbore de comportament este văzut ca pe o sarcină compusă, ***părinte***, ce deține mai multe sub sarcini, *fiu*. Această structură, deținută de un generator, este un obiect intitulat *awaitable* deoarece sarcina aferentă poate fi așteptată de către funcția apelant. Acest lucru este realizabil folosind o structură *Awaiter* care implementează o interfață C# denumită *ICriticalNotifyCompletion*. Atunci când o sarcină este terminată (completă) se poate continua la următoarea și tot așa. În acest mod, arborele de comportament implementează o modalitate de execuție asincronă.

Orice sarcină deține o expresie ***delegată*** ce reprezintă o funcție asincronă de tip *UniTask*. La rândul ei, sarcina trebuie să întoarcă către sarcina apelantă un status (***rulează***, *succes*, ***eșec***) ce reprezintă o enumerație. Pentru a putea asocia mai multe subsarcini fiu, a fost necesar implementarea unor metode extensie ce facilitează acest lucru. Fiecare fiu înregistrat este adăugat într-o structură de date de tip listă. Această listă este utilizată de nodul părinte pentru a cunoaște exact ce fiu trebuie executat.

3.4 IMPLEMENTAREA LIBRĂRIEI API

În acest sub capitol se prezintă modul de implementare a librăriei API pentru dezvoltarea de arbori de comportament. Această bibliotecă dispune de interfețe ce permit implementarea unor canale de comunicare *multi-agent* și posibilitatea de a scrie și extinde structuri de tip BT.

3.4.1 MEDIATOR PATTERN ŞI BLACKBOARD

Este un model descriptiv, de design, utilizat în programarea orientată pe obiecte. Acest model are la bază o clasă mediator care reprezintă un intermediar între obiectele abonate acesteia.

```
public interface IMediator<TBlackboard>
where TBlackboard : IBlackboard<TBlackboard> {
    public IReadOnlyCollection<TBlackboard> Blackboards { get; }
    public void AddSubscriber(TBlackboard blackboard);
    public void RemoveSubscriber(TBlackboard blackboard);
    public void SendMessage(IMessage<TBlackboard> message, TBlackboard to);
    public void BroadcastMessage(IMessage<TBlackboard> message);
}
```

code 5: Interfață de implementare a unui Mediator

Metodele *AddSubscriber* și *RemoveSubscriber* (*code 5*) sunt folosite pentru înregistrarea unui obiect abonat la mediator, astfel, realizându-se un canal de comunicare între agenții inteligenți. Deoarece o interacțiune este definită prin trimiterea de mesaje, implementarea unei astfel de abordări a permis inițierea de evenimente între agenți.

SendMessage este o metodă care trimite un mesaj către obiectul destinatar. Iar *BroadcastMessage*, după cum spune și numele, permite trimiterea aceluiași mesaj către toate obiectele abonate.

```
public interface IBlackboard<TBlackboard>: IEquatable<TBlackboard>
where TBlackboard: IBlackboard<TBlackboard> {
    public string GUID { get; }
    public object this[string key] { get; set; }
    public IMediator<TBlackboard> Mediator { get; }
    public bool CompareTag(string tag);
    public void SubscribeTo(IMediator<TBlackboard> mediator);
    public void UnsubscribeFrom(IMediator<TBlackboard> mediator);
    public void SendMessage(IMessage<TBlackboard> message, TBlackboard to);
    public void BroadcastMessage(IMessage<TBlackboard> message);
    public void ReceiveMessage(IMessage<TBlackboard> message);
}
```

code 6: Interfață de implementare a unui Blackboard

Un *Blackboard* (*code 6*) este un obiect ce se abonează la un mediator. În implementarea unor *Behaviour Trees*, un blackboard reprezintă memoria agentului. Acesta este un dicționar de perechi cheie – valoare care reprezintă informația utilă a agentului. Un blackboard poate fi abonat la un mediator folosind metoda *SubscribeTo* respectiv dezabona folosind *UnsubscribeFrom*. Trimiterea de mesaje și recepționarea acestora se realizează folosind metodele *SendMessage*, *BroadcastMessage* și *ReceiveMessage*. Cea din urmă primește un mesaj ce urmează a fi despachetat și salvat în dicționar.

```
Blackboard b1 = new Blackboard();
Blackboard b2 = new Blackboard();
b1.SendMessage(message, b2);     // Trimite mesaj către b2
b1.BroadcastMessage(message);    // Trimite mesaj către toţi blackboards
```

code 7 Exemplu de comunicare între două obiecte

Indexarea într-un blackboard se realizează folosind operatorul de indexare. (*code 8*) Acesta permite găsirea informației într-un dicționar. Dacă a fost găsită, aceasta va fi returnată. Dacă informația ce urmează a fi adăugată există deja în dicționar va fi modificată dacă nu este nulă, altfel va fi ștearsă iar dacă nu există, această informație va fi nou creată.

```
public object this[string name] {
    get {
        if (_blackboard.ContainsKey(name))
            return _blackboard[name];
        return null;
    }
    set {
        if(name != "") {
            if (_blackboard.ContainsKey(name)) {
                if (value is null)
                    _blackboard.Remove(name);
                else _blackboard[name] = value;
            }
            else _blackboard.Add(name, value);
        }
    }
}
```

code 8 Operatorul de indexare a unui blackboard

Structura prezentată în *code 9* este o implementare a unui pachet de tip mesaj trimis între agenți. Acesta conține un atribut nume și o valoare obiect. Ea este salvată în *blackboard* folosind metoda *UnpackMessage*. Folosind atributul pe post de index în dicționar, metoda asociază acestei chei valoarea din pachet.

```
private readonly struct BlackboardMessage<TBlackboard> : IMessage<TBlackboard>
where TBlackboard: IBlackboard<TBlackboard> {
    public string MessageName { get; }
    public object MessageValue { get; }
    public void UnpackMessage(TBlackboard blackboard) {
        blackboard[MessageName] = MessageValue;
    }
    public BlackboardMessage(string name, object value) {
        MessageName = name;
        MessageValue = value;
    }
}
```

code 9 Implementarea unor mesaje trimise între agenți

Trimiterea de mesaje între agenți se realizează prin intermediul mediatorului. Conținutul este o cheie salvată într-un dicționar iar valoarea asociată este un obiect de tip generic.

Într-o listă, dacă aceasta este nulă atunci se alocă memorie și se salvează primul abonat. Altfel, acesta este adăugat în listă. Un mesaj trimis public este primit de către toți abonații dacă aceștia există în listă. Dezabonarea unui obiect de la mediator înseamnă găsirea acestuia și ștergerea lui. Iar la trimiterea unui mesaj, este necesară verificarea existenței acestuia și primirea mesajului.

```
public class Mediator : MonoBehaviour, IMediator<Blackboard> {
    [SerializeField] private List<Blackboard> subscribers;
    public IReadOnlyCollection<Blackboard> Blackboards => subscribers;
    public Blackboard this[int index] => subscribers[index];

    public void AddSubscriber(Blackboard blackboard) {
        if (subscribers is null) subscribers = new List<Blackboard> { blackboard };
        else subscribers.Add(blackboard);
    }

    public void BroadcastMessage(IMessage<Blackboard> message) {
        if (subscribers is null) return;
        foreach (var subscriber in subscribers)
            subscriber.ReceiveMessage(message);
    }
    public void RemoveSubscriber(Blackboard blackboard) {
        if (subscribers is null) return;
        var index = subscribers.FindIndex((target) => target == blackboard);
        if(index > -1) subscribers.RemoveAt(index);
    }

    public void SendMessage(IMessage<Blackboard> message, Blackboard to) {
        if (subscribers is null || subscribers.Count == 0) return;
        Blackboard found = null;
        foreach (var subscriber in subscribers)
            if (subscriber.GUID == to.GUID) found = subscriber;
        if (found is null) return;
        found.ReceiveMessage(message);
    }
}
```

code 10 Implementarea clasei mediator

3.4.2 FLUENT API ŞI FACTORY PATTERN

Construirea de arbori de comportament se realizează folosind un model de tip *Factory*. O fabrică este clasă specializată în construirea și configurarea unui obiect care, altfel, acesta se putea definii foarte greu. Limbajul de programare C# permite declararea de metode extensie, abordare non-destructivă de a definii noi metode unei clase, care în mod obișnuit nu poate fi accesată de utilizator spre a putea fi modificată.

Declararea acestei clase (*code 11*) este una simplă, aceasta conținând o stivă de obiecte tip *BehaviourTask*. Folosind o înlănțuire de metode tip *Push*, *End* și *Generate*, respectiv metode extensie, declararea de arbori de comportament folosind această arbore respectă sintaxa de construire intitulată Fluent API.

Funcția *PushTask* este o implementare peste metoda *Push* a unei stive. Aceasta primește o sarcină, o inserează în stiva de sarcini și întoarce obiectul generator. Orice metodă intermediară trebuie să întoarcă obiectul care o apelează. În acest mod este facilitată apelarea înlănțuită de metode constructoare (*factory methods*).

Funcția *End*, apelată la finalul unei ramuri, sau a unui decorator, preia părintele din vârful stivei și dacă există cel puțin două astfel de noduri, atunci, nodul scos din stivă este înregistrat nodului imediat următor.

```
using BehaviourStack = Stack<BehaviourTask>;
public struct TaskGenerator
{
    private readonly BehaviourStack currentPointer;
    public BehaviourTask Task => currentPointer.Peek();
    public static TaskGenerator Begin() { return new TaskGenerator(); }
    private TaskGenerator() => currentPointer = new BehaviourStack();

    public TaskGenerator PushTask(BehaviourTask _task) {
        currentPointer.Push(_task);
        return this;
    }
    public TaskGenerator End() {
        if(currentPointer.Count >= 2) {
            var task = currentPointer.Pop();
            currentPointer.Peek().Register(task);
        }
        return this;
    }
    public BehaviourTask Generate() => currentPointer.Pop();
}
```

code 11 Clasa TaskGenerator, clasă tip factory care construiește arborii de comportament

În urma acestui lanț de apeluri, prin folosirea metodei *End* se va ajunge la un singur nod, acesta este rădăcina arborelui și trebuie întors imediat folosind *Generate*. În acest fel, obiectul *TaskGenerator* permite construirea de arbori de comportament.

Arborele descris (*code 12*) implementează, folosind un obiect *TaskGenerator* și abordarea „fluent", acțiunea realizată de un agent de a se plimba în mediul în care acesta se află. Stiva este inițial vidă, cu fiecare apel de metodă, dacă este una ce definește un nod compus, aceasta introduce în stivă câte un nod asupra căruia se realizează operațiunea de construcție. Selectorul, de exemplu, se așteaptă să primească noduri fiu, fiecare nod, dacă este acțiune, acesta este imediat înregistrat, altfel, este pus în stivă. Nodurile tip acțiune se vor adăuga întotdeauna la nodul din vârful stivei iar dacă construcția este finalizată acesta va fi scos din stivă și adăugat nodului următor. La sfârșit se va apela metoda Generate, semnalând finalizarea construcției întregului arbore.

```
public BehaviourTask Wander(float speed, float distance) =>
    TaskGenerator.Begin()
        .Selector()
            .Conditional(() => (float)Blackboard["Energy"]>0&&!(bool)Blackboard["Resting"])
                .Sequencer()
                    .PlayAnimation(Animation, "Walking", AnimationCategory.WalkingAnimation)
                    .UpdateBlackboard(Blackboard, "Energy", () => {
                        float depletionRate = -0.05f * Random.value;
                        return depletionRate* Blackboard.Genome.MaxStamina * Time.deltaTime;
                    })
                    .Wander(Agent, speed, distance)
                .End()
            .End()
            .Conditional(() => (float)Blackboard["Energy"] < 0)
                .SetBlackboard(Blackboard, "Resting", () => true)
            .End()
        .Generate();
```

code 12 Exemplu de construire a unui BT

3.4.3 Asynchronous Behaviour Tree

Un *arbore de comportament* este unul implementat *asincron* atunci când unui nod îi se poate asocia un ***automat cu stări finite*** ce descrie etapele de luare de decizie respectiv executare a nodurilor fiu. De obicei, nodurile tip compus și decorator sunt noduri asincrone, dar și noduri tip acțiune pot avea la rândul lor un astfel de automat (de exemplu, acțiunea mers, dacă nu este necesară abandonarea acesteia este o acțiune asincronă). Un nod fiu poate fi abandonat atunci când sarcina executată nu ajunge la un compromis sau în cazul în care are loc realizarea de evenimente care nu aduc „***bunăstare***" agentului inteligent. Această abandonare trebuie tratată de către nodul copil

deoarece numai acesta are mecanisme de oprire, fiecare automat cu stări finite fiind independent de cel al părintelui.

Implementarea nodurilor compuse

Algoritmul (*code 13*), deși este iterativ, acesta este de fapt o implementare asincronă. Diferența față de implementarea clasică este dată de funcția asociată nodului care este una asincronă (folosește sintaxa *async / await*). Pentru fiecare sarcină fiu, dacă părintele nu este oprit din execuție, acesta setează un *token* de oprire nodului fiu, așteaptă după aceasta să își termine execuția iar apoi decide dacă continuă (acesta a întors *succes*) sau se oprește (acesta întoarce *eșec*).

```
public static TaskGenerator Sequencer(this TaskGenerator builder) {
    return builder.PushTask(BehaviourTask.Run(async (self, token) => {
        foreach (var task in self.Tasks) {
            if (token.IsCancellationRequested)
                return BehaviourStatus.Failure;

            task.SetToken(token);
            var status = await task;
            if (status == BehaviourStatus.Failure)
                return BehaviourStatus.Failure;
        }
        return BehaviourStatus.Success;
    }));
}
```

code 13 Algoritmul de funcționare a unui nod secvență

În *fig. 27* se poate observa structura automatului cu stări finite $\delta(S, \mathrm{T})$. Mulțimea stărilor automatului, $S = \{a, b\}$ unde $a = \langle foreach\ task \rangle$ și $b = \langle run\ task \rangle$ sunt stările în care nodul se poate afla la un moment dat. În starea *a*, automatul parcurge fiecare task și decide viitoarele acțiuni, iar în starea *b* se execută sarcina la care s-a ajuns. Fiecare tranziție din automat permite nodului fiu alegerea deciziei. Dacă există sarcini de executat și acțiunea nu este oprită, se rulează sarcina. Dacă aceasta este terminată și a eșuat atunci se termină execuția, altfel se continuă cu următoarea sarcină. Dacă s-au terminat toate sarcinile de executat, atunci se v-a termina cu succes și se va ieși din funcție.

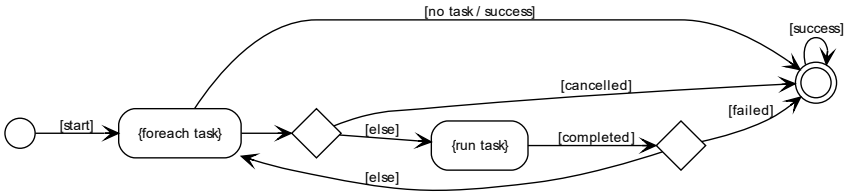


fig. 27 Automatul cu stări finite asociat unui nod secvență

Spre deosebire de nodul secvență, nodul selector (*code 14*) se oprește atunci când cel puțin un nod fiu a fost executat și a întors *succes*, altfel se v-a întoarce *eșec*. Automatul asociat este unul asemănător cu cel al unei secvențe, diferența este dat de tranzițiile care au loc în acesta.

```
public static TaskGenerator Selector(this TaskGenerator builder) {
    return builder.PushTask(BehaviourTask.Run(async (self, token) => {
        foreach (var task in self.Tasks) {
            if (token.IsCancellationRequested)
                return BehaviourStatus.Failure;

            task.SetToken(token);
            var status = await task;
            if (status == BehaviourStatus.Success)
                return BehaviourStatus.Success;
        }
        return BehaviourStatus.Failure;
    }));
}
```

code 14 Implementarea nodului Selector

Automatul din *fig. 28* își începe execuția în starea *a,* stare în care pentru fiecare sarcină i-a decizia dacă trebuie continuată sau nu execuția, iar în starea *b* acesta rulează sarcina fiu la care se află în acel moment. Dacă are sarcini de executat și nu a fost anulat nodul, se v-a executa sarcina până ce aceasta este terminată, dacă a întors rezultatul *succes*, se oprește execuția, altfel, se continuă cu următorul nod. Dacă toate nodurile au fost epuizate, se va ieși din execuție returnând *eșec*.

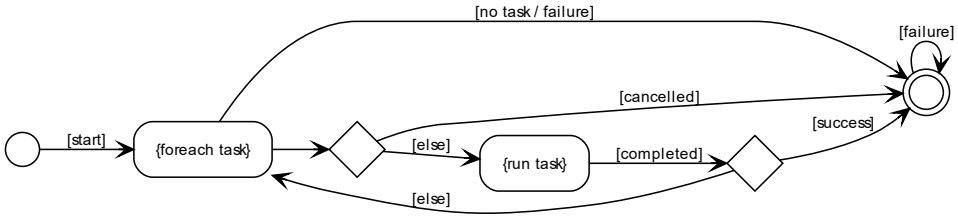


fig. 28 Automatul cu stări finite ale unui nod selector

Nodul paralel preia fiecare sarcină împachetată sub formă de *UniTask* și le execută pe toate simultan. Spre deosebire de nodul paralel clasic, această implementare nu folosește verificarea stării nodurilor deoarece nu contează dacă acestea au întors *succes*, sau *eșec*. Se va întoarce *succes* deoarece se știe clar că toate au fost executate.

```
public static TaskGenerator Parallel(this TaskGenerator builder) {
    return builder.PushTask(BehaviourTask.Run(async (self, token) => {
        var tasks = new UniTask<BehaviourStatus>[self.Tasks.Count];
        for (int i = 0; i < self.Tasks.Count; i++)
            tasks[i] = UniTask.Create(async () => await self.Tasks[i]);
        await UniTask.WhenAll(tasks);
        return BehaviourStatus.Success;
    }));
}
```

code 15 Algoritmul unui nod paralel

Automatul (*fig. 29*) pornește iterativ execuția nodului construind o listă de sarcini. La sfârșitul definirii listei, se trece în starea de execuție a sarcinilor. Când toate acestea s-au terminat de executat, se va ieși cu starea succes.

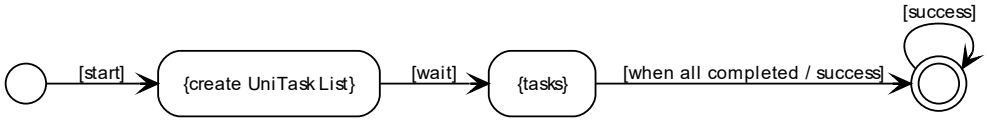


fig. 29 Automatul cu stări finite a unui nod paralel

Implementarea nodurilor decorator

Pentru a permite execuția asincronă a condițiilor, acestea au fost definite ca noduri decoratoare. Un nod decorator condițional (*code 16*) primește o condiție, sub formă de funcție care întoarce *bool*, și o evaluează. Dacă aceasta a întors *false*, se întoarce *eșec*, altfel, se execută nodul fiu.

```
public static TaskGenerator Conditional(this TaskGenerator builder, Func<bool> condition) =>
    builder.PushTask(BehaviourTask.Run(async (self, token) => {
        if (token.IsCancellationRequested) return BehaviourStatus.Failure;
        if (!condition()) return BehaviourStatus.Failure;
        return await self.Tasks[0];
    }));
```

code 16 Algoritmul de execuție a unui decorator condițional

Abordarea asincronă (*fig. 30*) presupune așteptarea execuției nodului fiu numai în cazul în care condiția a fost validată. Se va ieși din rulare numai atunci când nodul fiu a fost executat sau atunci când condiția nu poate fi validată.

Dacă sarcina principală este anulată, execuția se oprește cu *eșec*, altfel, dacă condiția nu a fost validată, acesta întoarce *eșec* iar în final, dacă condiția este una validă, atunci nodul fiu este executat până acesta termină execuția și se iese din funcție cu rezultatul întors de fiu.

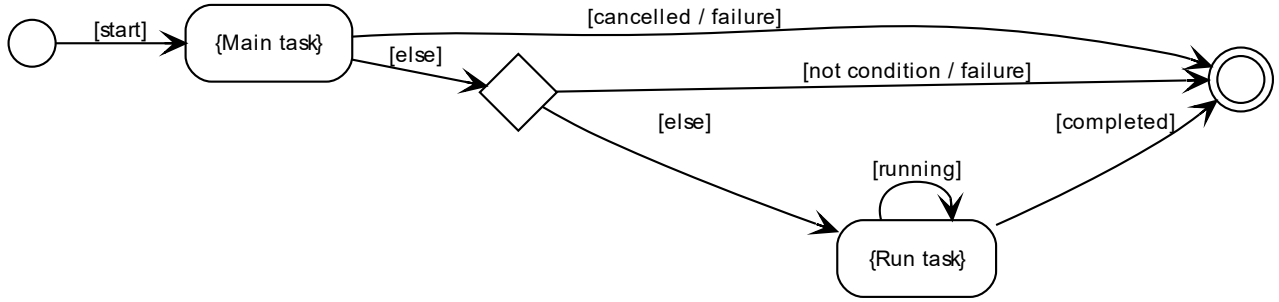


fig. 30 Automatul cu stări finite a unui decorator condițional

În aplicația *EvoAgents – Rabbit Catcher*, pentru a putea repeta acțiunea cadru cu cadru, și pentru a putea executa și alte instrucțiuni în paralel, implementarea folosește *noduri decorator repetitive*. Un *decorator repetitiv* este un nod care execută o sarcină repetat până la o condiție de oprire sau până la oprirea totală a execuției.

Nodul *decorativ repetitiv cu repetiție la infinit* (*code 17**fig. 17*) execută nodul fiu, la interval de un *tick* (un cadru pe secundă) și indiferent de rezultatul nodului fiu acesta își continua execuția până ce a primit semnal de oprire. În acest moment, nodul fiu întoarce rezultatul *eșec*.

```
public static TaskGenerator RepeatForever(this TaskGenerator builder) =>
    builder.PushTask(BehaviourTask.Run(async (self, token) => {
        while (!token.IsCancellationRequested) {
            await self.Tasks[0];
            await UniTask.WaitForEndOfFrame();
        }
        return BehaviourStatus.Failure;
    }));
```

code 17 Algoritmul de execuție al unui nod decorator repetitiv

Automatul (*fig. 31*) își începe execuția în starea principală, dacă aceasta nu a fost terminată, execuția se repetă și rulează o sarcină fiu, dacă aceasta nu și-a terminat execuția, se așteaptă terminația ei, în final, dacă s-a executat se așteaptă finalizarea unui *frame*, apoi se continuă execuția. Aceasta nu se termină decât după ce s-a primit semnal de oprire.

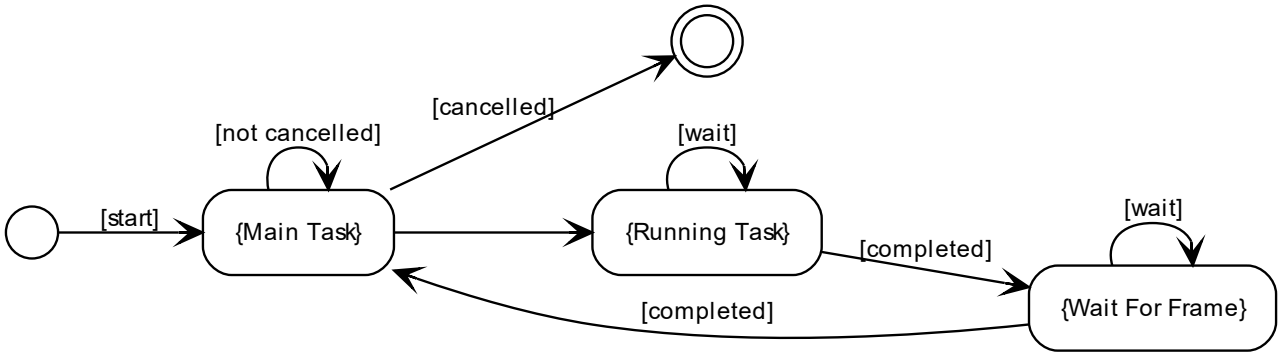


fig. 31 Automatul cu stări finite asociat nodului decorator repetitiv

3.5 TESTAREA APLICAȚIEI

Etapa de testare a constat în scrierea de *teste unitare (Unit Tests)* folosind librăria oferită de către *Unity* pentru a testa funcționalitățile unui joc video. (*fig. 32*) Deoarece proiectul principal implică dezvoltarea unei metode de a implementa arbori de comportament, s-a recurs la scrierea unor teste ce presupune verificarea execuției arborilor. Agenții inteligenți s-au testat folosind metoda obișnuită, empirică, și anume, execuția repetată a jocului și verificarea erorilor de comportament ale acestora.

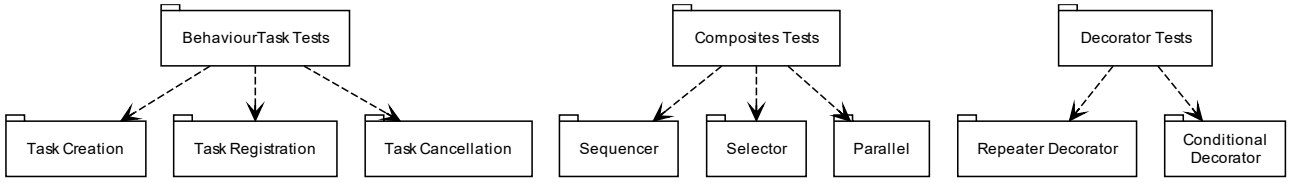


fig. 32 Diagramă arhitectură teste unitare

Dacă BT sunt bine implementați, singurele probleme ce mai pot apărea sunt cele de atenție din partea programatorului.

În prima fază s-a testat dacă o sarcină este executată și întoarce rezultatul așteptat. (*code 18*) Dacă în urma execuției, cumva nu se poate crea o sarcină, sau rezultatul întors de ea nu este cel așteptat, atunci se vor afișa mesaje corespunzătoare de eroare.

```
[Test]
public async void TaskCreationTest() {
    var task = BehaviourTask.Run((self, token) => {
        Debug.LogWarning("Main task running.");
        return UniTask.FromResult(BehaviourStatus.Success);
    });
    Assert.IsNotNull(task, "Task creation failed.");

    var status = await task;
    Assert.AreEqual(BehaviourStatus.Success, status, "Values are not equals.");
}
```

code 18 Testarea creării și execuției unei sarcini

Pentru a putea construi un arbore de comportament, a fost necesar implementarea unei modalități de a înlănțui multiple sarcini. (*code 19*) Testarea acestui lucru a însemnat crearea unei sarcini părinte care execută și verifică rezultatul întors de o sarcină fiu înregistrată la aceasta. O sarcină fiu este înregistrată la un părinte dacă în lista acestuia se află cel puțin o sarcină fiu. Aceasta este rulată asincron iar rezultatul este verificat să fie cel așteptat. Dacă sarcina părinte a putut fi rulată aceasta va întoarce rezultatul *succes*, semnalând că testul a fost unul favorabil.

```
[Test]
public async void TaskRegistrationTest() {
    var task = BehaviourTask.Run(async (self, token) => {
        Debug.LogWarning("Main task running.");
        Assert.GreaterOrEqual(1, self.Tasks.Count, "There is no task in the list");
        var status = await self.Tasks[0];
        Assert.AreEqual(BehaviourStatus.Success, status);
        return BehaviourStatus.Success;
    });
    Assert.IsNotNull(task, "Task creation failed.");

    var status = await task.Register(BehaviourTask.Run((self, token) => {
        Debug.LogWarning("Task has been runned.");
        return UniTask.FromResult(BehaviourStatus.Success);
    }));
    Assert.AreEqual(BehaviourStatus.Success, status, "Values are not equals.");
}
```

code 19 Testul de înregistrare a unor sarcini fiu la una părinte

Nodul decorator repetitiv execută un nod fiu în mod repetat până la oprirea execuției acestuia. În final, testul acestui nod (*code 20*) este de fapt un supra test peste cel al nodurilor compuse și a opririi execuției unei sarcini. S-a creat un *token* de oprire care va semnala acest fapt după un sfert de secundă de la crearea acestuia și punerea în execuție a arborelui. În urma opririi execuției, nodul decorator întoarce rezultatul ***eșec***, indiferent de cel al nodului fiu asociat.

```
[Test]
public async void RepeaterDecoratorTest() {
    var tokenSource = new CancellationTokenSource();
    tokenSource.CancelAfter(TimeSpan.FromSeconds(0.25f));
    var task = TaskGenerator.Begin()
        .RepeatForever()
            .Selector()
                .Action((self, token) => {
                    return UniTask.FromResult(BehaviourStatus.Failure);
                })
                .Action((self, token) => {
                    Debug.Log("This task is running");
                    return UniTask.FromResult(BehaviourStatus.Success);
                })
            .End()
        .Generate();
    task.SetToken(tokenSource.Token);
    var status = await task;
    Assert.AreEqual(BehaviourStatus.Failure, status);
}
```

code 20 Testarea nodului decorator repetitiv cu test de oprire

Nodul decorator condițional rulează o sarcină fiu dacă condiția acestuia este validată, altfel întoarce ***eșec***, acest rezultat permite trecerea condiționată la următorul fiu dacă nodul compus este unul selector. În cazul unui nod secvență, execuția este oprită imediat dacă condiția fiului este una invalidată. (*code 21*)

```
[Test]
public async void ConditionalDecoratorTest() {
    var tokenSource = new CancellationTokenSource();
    tokenSource.CancelAfter(TimeSpan.FromSeconds(0.25f));
    var task = TaskGenerator.Begin()
        .RepeatForever()
            .Selector()
                .Conditional(() => false)
                    .Action((self, token) => {
                        Assert.Fail("This task dosen't run");
                        return UniTask.FromResult(BehaviourStatus.Success);
                    })
                .End()
                .Action((self, token) => {
                    Debug.Log("This task is running");
                    return UniTask.FromResult(BehaviourStatus.Success);
                })
            .End()
        .Generate();
    task.SetToken(tokenSource.Token);
    var status = await task;
    Assert.AreEqual(BehaviourStatus.Failure, status);
}
```

code 21 Testarea nodului condițional

Un arbore, deoarece rulează independent de restul clasei din care face parte, trebuie testată oprirea execuției acestuia. (*code 22*) În acest test se va realiza un *token* de oprire care va semnala acest lucru. El va fi propagat în arbore, de la rădăcină către frunze iar fiecare nod va trebui să verifice dacă sarcina a fost sau nu oprită. Dacă aceasta este oprită se întoarce ***eșec*** către părinte, iar el își va continua execuția până ce ajunge la condiția de oprire.

```
[Test]
public async void TaskCancellationTest() {
    var tokenSource = new CancellationTokenSource();
    var task = BehaviourTask.Run(async (self, token) => {
        Debug.LogWarning("Main task running.");
        self.Tasks[0].SetToken(token);
        var status = await self.Tasks[0];
        Assert.AreEqual(BehaviourStatus.Failure, status);
        return BehaviourStatus.Success;
    }, tokenSource.Token);
    Assert.IsNotNull(task, "Task creation failed.");

    tokenSource.Cancel();
    var status = await task.Register(BehaviourTask.Run((self, token) => {
        if (token.IsCancellationRequested) {
            Debug.Log("Task has been cancelled.");
            return UniTask.FromResult(BehaviourStatus.Failure);
        }
        Debug.LogWarning("Task has been runned.");
        return UniTask.FromResult(BehaviourStatus.Success);
    }));
    Assert.AreEqual(BehaviourStatus.Success, status, "Values are not equals.");
}
```

code 22 Testarea anulării unei sarcini puse în rulare

Testarea nodurilor compuse, *code 23* și *code 24*, a presupus verificarea dacă modul de funcționare este cel corect. Nodul *secvență*, spre exemplu, presupune execuția la rând a acțiunilor dacă acestea sunt *succese*, iar nodul *paralel* dorește rularea concurentă a tuturor acțiunilor, indiferent de statusul acestora.

```
[Test]
public async void SequencerTest() {
    var status = await TaskGenerator.Begin()
        .Sequencer()
            .Action(BehaviourTask.Run((self, token) => {
                Debug.LogWarning("This task runs first and fails");
                return UniTask.FromResult(BehaviourStatus.Failure);
            }))
            .Action(BehaviourTask.Run((self, token) => {
                Debug.LogAssertion("This task fails the test");
                return UniTask.FromResult(BehaviourStatus.Success);
            }))
        .Generate();
    Assert.AreEqual(BehaviourStatus.Failure, status);
}
```

code 23 Testarea nodului secvență utilizând două noduri fiu

```
[Test]
public async void ParallelTest() {
    var status = await TaskGenerator.Begin()
        .Parallel()
            .Action(BehaviourTask.Run((self, token) => {
                Debug.LogWarning("This task runs first in parallel");
                return UniTask.FromResult(BehaviourStatus.Failure);
            }))
            .Action(BehaviourTask.Run((self, token) => {
                Debug.LogWarning("This task runs second in parallel");
                return UniTask.FromResult(BehaviourStatus.Success);
            }))
        .Generate();
    Assert.AreEqual(BehaviourStatus.Success, status);
}
```

code 24 Testarea nodului compus paralel

CAPITOLUL 4 GHIDUL DEZVOLTATORULUI / UTILIZATORULUI

Biblioteca ce a fost implementată în acest proiect este una extensibilă pentru dezvoltatori, aceștia putând să dezvolte noi tipuri de noduri și arbori de comportament. În acest capitol se va prezenta ghidul dezvoltatorului, ce trebuie un programator să facă pentru a putea folosi acest API, și cum se joacă jocul *EvoAgents – Rabbit Catcher*.

4.1 GHIDUL DEZVOLTATORULUI (UTILIZARE API)

Primul mod de implementare constă în definirea de clase derivate din cea principală, *BehaviourTask*. Tot o dată, realizând această derivare se vor putea utiliza variabile asociate clasei derivate deoarece atunci când metoda delegat *OnInvoke* este apelată, obiectul trimis prin self este de proveniență din clasa derivată. În exemplu de mai jos a fost declarat un nod de tip ***secvență*** care își va executa fii numai dacă variabila test este *true,* altfel nu va rula și va întoarce ***eșec***.

code 25 Exemplu de implementare a unei clase derivate demonstrează modul în care se pot realiza clase derivate de tip *BehaviourTask* care să permită funcționalități specifice, implementate de utilizator. Clasa de mai jos propune un nod secvență care cunoaște o stare în care acesta nu poate să execute acțiunile. Această stare este definită prin intermediul variabilei *test*.

```
public class Sequencer : BehaviourTask
{
    private bool test = false;
    private static async UniTask<BehaviourStatus> SequenceTask(BehaviourTask self,
CancellationToken token) {
        var sequence = self as Sequencer;
        foreach (var task in self.Tasks) {
            if (token.IsCancellationRequested) return BehaviourStatus.Failure;
            task.SetToken(token);
            if (sequence.test) {
                var status = await task;
                sequence.test = false;

                if (status == BehaviourStatus.Failure)
                    return BehaviourStatus.Failure;
            }
            return BehaviourStatus.Failure;
        }
        return BehaviourStatus.Success;
    }
    public Sequencer() : base(SequenceTask) { }
}
```

code 25 Exemplu de implementare a unei clase derivate

Acest mod de implementare presupune definirea unei metodă statice care va fi trimisă către constructorul clasei de bază. Cel al clasei derivate se poate ocupa de elemente specifice acesteia făcând abstracție de ceea ce știe clasa de bază. Este important de reținut că pentru a putea accesa, folosind obiectul „*self*”, a variabilelor proprii ale clasei, acesta trebuie convertit la un obiect din clasa derivată. Această conversie este una explicită și folosește elemente de polimorfism.

Pentru a putea construii arbori de comportament folosind nodul definit de utilizator, acesta are obligația de a declara funcții extensie la clasa tip *factory*. Exemplul *code 26* demonstrează implementarea acestei metode extensie. Ea creează un obiect de tip ***secvență*** și o pune pe stiva de creare. Acest fapt permite construirea de noduri în arbore folosind clasa de tip *TaskGenerator*.

```
public static class CustomSeqeunceExtension {
    public static TaskGenerator CustomSequence(this TaskGenerator builder) {
        return builder.PushTask(new Sequencer());
    }
}
```

code 26 Definirea unei metode extensie

Cel de al doilea mod de a implementa noduri definite de utilizator este de a crea metode extensie de la obiectul *TaskGenerator*. Această abordare este una limitată deoarece nu se pot reține proprietăți specifice nodului dar se pot trimite atribute ca și parametrii funcției declarative. Orice atribut static nu va putea fi modificat în timpul execuției deoarece este unul specific declarării (nu sunt transmise prin referință).

Un mod generic, ce permite alterarea valorilor trimise ca parametru o reprezintă definirea unei funcții delegat care să facă acest lucru, nodul din arbore doar apelează funcția și preia informația întoarsă de aceasta. În acest mod, nodul declarat poate fi considerat decorator, dar se poate aplica și acțiunilor. *Declararea unei acțiuni cu eveniment stocastic. (cu funcție parametru)* (*code 27*) se poate în modul următor. Se presupune că funcția parametru determină o probabilitate de execuție. Dacă aceasta este mai mare de 50% atunci acțiunea va fi executată cu *succes*.

```
public static TaskGenerator CustomActionWithFunc(this TaskGenerator builder, Func<float> func){
    return builder.Action((self, token) => {
        var valoare = func();
        if (valoare > 0.5f){
            Debug.Log("Execute action");
            return UniTask.FromResult(BehaviourStatus.Success);
        }
        return UniTask.FromResult(BehaviourStatus.Failure);
    });
}
```

code 27 Declararea unei acțiuni cu eveniment stocastic. (cu funcție parametru)

Pentru a putea adăuga acțiuni, clasa *factory* nu dispune de metodă proprie deoarece este suficientă implementarea extensiei *Action*. Această metodă se folosește de abilitatea de a înregistra sub sarcini la cea aflată în vârful stivei de construcție. (*code 28*)

```
public static TaskGenerator CustomAction(this TaskGenerator builder, float attribute) {
    return builder.Action((self, token) => {
        Debug.Log(attribute);
        return UniTask.FromResult(BehaviourStatus.Success);
    });
}
```

code 28 Definirea unei metode extensie ce implementează o acțiune

În același mod cu implementarea unui nod compus, implementarea unui decorator se poate realiza fie prin derivare din *BehaviourTask* ori a unei metode extinse la *TaskGenerator*.

Dacă se dorește un control mai fin al funcționalității unui nod decorator se poate folosi abordarea cu clasă derivată. Un astfel de nod va putea utiliza funcții sau metode proprii pentru alterarea stării unui nod fiu, executarea ei sau renunțarea la aceasta. Alternativa metodei extinse este presupunerea transmiterii de valori atribut sau metode delegat la declarare către decorator.

Exemplul evidențiat de *code 29* demonstrează o astfel de implementare a unui nod decorator care să execute sarcina numai dacă condiția este adevărată. Clasa primește ca variabilă funcția delegat care face acest lucru.

Pentru a construi arbori de comportament, biblioteca dispune de o modalitate declarativă folosind clasa de tip *factory* care permite înlănțuirea de noduri extensie pentru a definii structura acestuia. Orice construire a unui arbore începe cu *Begin*() și trebuie încheiată cu *Generate*(). Aceste două metode, împreună cu *PushTask*() și *Action*() și toate extensiile de noduri, folosind o stivă de sarcini, construiesc un BT.

```
public class Conditional: BehaviourTask {
    private Func<bool> condition;
    private bool ExecuteTask(CancellationToken token) {
        Tasks[0].SetToken(token);
        return condition();
    }
    public static async UniTask<BehaviourStatus> ConditionalTask(BehaviourTask self,
CancellationToken token) {
        var conditional = self as Conditional;
        if (conditional.ExecuteTask(token))
            return await self.Tasks[0];
        return BehaviourStatus.Success;
    }
    public Conditional(Func<bool> condition) : base(ConditionalTask) {
        this.condition = condition;
    }
}
```

code 29 Declararea unei clase derivate condiționale

Declarațiile de noduri intermediare sunt accesate incluzând pachetul corespunzător tipurilor de noduri. De exemplu, pentru a obține un arbore complet funcțional, este necesar utilizarea pachetelor *EvoAgents.Behaviours.Actions*, respectiv *Composites*. Pentru a utiliza noduri decorator condițional, respectiv repetitiv, se include pachetul *Decorators*.

Declarația arborelui începe cu metoda *Begin*, apoi se alege tipul de nod ce trebuie utilizat. Dacă se va folosii un decorator, sau un nod compus, acesta va fi adăugat în stivă și nu va fi scos abia după apelarea metodei *End*, semnalând sfârșitul construcției unei ramuri. Un nod *Action*, prin apelul acestei metode, va adăuga la nodul din vârful stivei, dacă există, un nod nou *acțiune*. Nodurile *decorator* acceptă mai multe sub noduri, dacă se vrea, dar are acces doar la prima sarcină.

Astfel spus, un arbore de comportament este construit folosind abordarea „*fluent*”, abordare declarativă de a defini obiecte dintr-o anumită clasă. Orice parametru transmis metodelor apelate se va utiliza numai la etapa aceasta. Dacă se dorește și în timpul rulării unui arbore, ele trebuie accesate numai prin intermediul unor funcții *delegat*. *Exemplu de construire a unui BT* (*code 12*) prezintă modul în care se poate definii un arbore de comportament folosind o metodă ce primește doi parametrii, viteza și distanța agentului care se plimbă.

```
public BehaviourTask Wander(float speed, float distance) =>
    TaskGenerator.Begin()
        .Selector()
            .Conditional(() => (float)Blackboard["Energy"]>0&&!(bool) Blackboard["Resting"])
                .Sequencer()
                    .PlayAnimation(Animation, "Walking", AnimationCategory.WalkingAnimation)
                    .UpdateBlackboard(Blackboard, "Energy", () => {
                        float depletionRate = -0.05f * Random.value;
                        return depletionRate* Blackboard.Genome.MaxStamina * Time.deltaTime;
                    })
                    .Wander(Agent, speed, distance)
                .End()
            .End()
            .Conditional(() => (float)Blackboard["Energy"] < 0)
                .SetBlackboard(Blackboard, "Resting", () => true)
            .End()
        .Generate();
```

code 12 Exemplu de construire a unui BT (Vezi pagina 39)

Pentru a definii o structură de date asociată unui arbore de comportament este necesară implementarea interfeței *IBlackboard*, acesta este definită generic astfel încât fiecare agent inteligent să poată fi unul unic. De exemplu, un iepure nu deține aceleași capabilități precum un lup care pot să fie specifice acestuia. Tipul generic așteptat este tot un *IBlackboard*, și anume, acela care implementează această interfață.

De exemplu, clasa oferită de librărie este una universală și este declarată în felul următor:

```
public class Blackboard: MonoBehaviour, IBlackboard<Blackboard>
```

code 30 Antet declarație clasă Blackboard

Prin implementarea acestei interfețe cu tipul generic fiind însăși clasa care o implementează, aceasta poate să se aboneze numai la canale a căror abonați sunt de același tip (*code 31*). Dacă implementarea interfeței se va realiza peste un blackboard existent deja, clasa respectivă va fi una ce se poate abona numai la mediatori specifici. În acest fel, un agent inteligent ce se folosește de acest tip de structură este unul capabil să poată răspunde la mesaje de la alți agenți de acel tip, doar că informația salvată va fi una relevantă numai acelui agent.

```
public abstract class Rabbit : MonoBehaviour, IBlackboard<Blackboard> {
    public abstract object this[string key] { get; set; }
    public abstract string GUID { get; }
    public abstract IMediator<Blackboard> Mediator { get; }
    public abstract void BroadcastMessage(IMessage<Blackboard> message);
    public abstract bool Equals(Blackboard other);
    public abstract void ReceiveMessage(IMessage<Blackboard> message);
    public abstract void SendMessage(IMessage<Blackboard> message, Blackboard to);
    public abstract void SubscribeTo(IMediator<Blackboard> mediator);
    public abstract void UnsubscribeFrom(IMediator<Blackboard> mediator);
}
```

code 31 Clasă abstractă ce implementează un blackboard generic

Implementarea (*code 31*), deși abstractă, doar pentru exemplu, definește categoria de agenți inteligenți tip iepure dar care pot răspunde la mesaje de la orice tip de agent inteligent, chiar și de la lupi. Dacă interfața implementează comunicarea între iepuri, tipul generic trebuia să fie tipul implementat, astfel doar agenți de acest tip își pot trimite mesaje printr-un mediator.

Dacă dezvoltatorul dorește să permită unor agenți de un tip (*code 32*) să comunice atât printr-un canal privat, dar și prin cel public, o derivare din clasa *Blackboard* și implementarea unor metode ce permit comunicarea între agenți din clasa derivată este posibilă. Pentru a accesa canalul nou adăugat, obiectul derivat trebuie să fie convertit la interfața de comunicare specifică acelui canal.

Construcția face conversie explicită la interfața implementată și transmite prin canalul acestuia mesajul către toți agenții de tip iepure. Trebuie avut grijă ca un agent să fie abonat la acest canal, fiind insuficient canalul generic deoarece mesajul va ajunge numai prin intermediul acestuia datorită conversiei.

```
public abstract class Rabbit : Blackboard, IBlackboard<Rabbit> {
    IMediator<Rabbit> IBlackboard<Rabbit>.Mediator { get; }
    public abstract void BroadcastMessage(IMessage<Rabbit> message);
    public abstract bool Equals(Rabbit other);
    public abstract void ReceiveMessage(IMessage<Rabbit> message);
    public abstract void SendMessage(IMessage<Rabbit> message, Rabbit to);
    public abstract void SubscribeTo(IMediator<Rabbit> mediator);
    public abstract void UnsubscribeFrom(IMediator<Rabbit> mediator);
}
```

code 32 Extensie la un blackboard astfel încât acesta să poată comunica și cu agenți iepure

Declarația (*code 33*) este un mediator a căror abonați sunt obiecte de tip iepure. Aceasta facilitează comunicarea între agenți inteligenți de acest tip. Pentru o comunicare publică, și privată, între obiecte se poate deriva o clasă din mediator ce implementează un canal mediator pentru iepuri

```
public abstract class RabbitMediator : MonoBehaviour, IMediator<Rabbit> {
    public abstract IReadOnlyCollection<Rabbit> Blackboards { get; }
    public abstract void AddSubscriber(Rabbit blackboard);
    public abstract void BroadcastMessage(IMessage<Rabbit> message);
    public abstract void RemoveSubscriber(Rabbit blackboard);
    public abstract void SendMessage(IMessage<Rabbit> message, Rabbit to);
}
```

code 33 Declararea unui mediator ce permite comunicarea între agenți de tip iepure

Dacă clasa este una derivată din mediator (*code 34*) și implementează o interfață pentru abonați de tip iepure aceasta permite comunicarea folosind două căi, una publică, în care toți agenții inteligenți primesc mesaje, dar și una privată, accesibilă prin conversia la interfața specifică, în care doar agenții de acest tip pot vedea și primii mesaje.

```
public abstract class RabbitMediatorPublic : Mediator, IMediator<Rabbit>  {
    IReadOnlyCollection<Rabbit> IMediator<Rabbit>.Blackboards { get; }
    public abstract void AddSubscriber(Rabbit blackboard);
    public void BroadcastMessage(IMessage<Rabbit> message) {
        foreach (var rabbit in ((IMediator<Rabbit>)this).Blackboards)
            rabbit.ReceiveMessage(message);
    }
    public abstract void RemoveSubscriber(Rabbit blackboard);
    public abstract void SendMessage(IMessage<Rabbit> message, Rabbit to);
}
```

code 34 Declararea unui mediator care permite comunicare publică dar și privată, între iepuri

Toate aceste clase sunt de tip *MonoBehaviour* deoarece sunt considerate *scripturi* de către Unity, ele trebuind atașate unor obiecte din scenă pentru a funcționa corect. Listele de abonați pot fi declarate atât dinamic, la rulare, cât și static, atunci când este realizată construcția scenei. Pentru a putea face acest lucru este necesară declararea unui câmp a cărei atribut este *SerializeField.*

În biblioteca *EvoAgents Behaviours* nu este necesară cunoașterea structurii mesaj, dar trebuie cunoscută modalitatea de desfacere și salvare a acesteia într-un *Blackboard*. Un mesaj (*code 35*) poate conține multiple câmpuri și este o structură. În acest fel, prin trimiterea unui mesaj de la un agent inteligent la altul, mesajul nu va genera informație care trebuie ștearsă din memorie după ce acesta a fost trimis și nu mai este folosit. Structura mesajului conține un nume cheie și o valoare ce este salvată, folosind o metodă oferită de interfață, într-un *blackboard*.

```
private readonly struct BlackboardMessage<Blackboard> : IMessage<Blackboard> {
    public string MessageName { get; }
    public object MessageValue { get; }

    public void UnpackMessage(Blackboard blackboard) {
        blackboard[MessageName] = MessageValue;
    }
    public BlackboardMessage(string name, object value) {
        MessageName = name;
        MessageValue = value;
    }
}
```

code 35 Declararea unei structuri de tip mesaj

Câmpul valoare al unui mesaj poate la rândul ei să conțină o colecție de informații, de exemplu, întreaga descriere a unui obiect perceput din mediu. Astfel acesta poate fi unul foarte complex, și utilizarea acestuia depinde numai de felul în care agentul inteligent este programat să îl folosească.

În exemplul de mai sus, un mesaj este o structură ce deține numele mesajului și valoarea salvată, acesta este despachetat și salvat numai în structura *Blackboard* căreia îi este destinat.

4.2 Cum se joacă EvoAgents – Rabbit Catcher

Interfața (*fig. 33*) este una foarte simplă, aceasta permițând jucătorului să vadă un scor. Acesta reprezintă numărul de iepuri capturați din numărul total generat atunci când jocul este pornit. În cazul în care, o parte din iepuri au fost prinși de lup scorul final va fi un număr mai mic decât cel total, acest moment reprezentând sfârșitul jocului. Utilizatorul, atât pe durata jocului, dacă se apasă tasta *escape*, cât și la finalul acestuia, poate să apese pe „Regenerate World”. Acest buton va reseta complet jocul (în afară de viața jucătorului) și va regenera o nouă lume în care vor exista un număr de cel puțin un iepure și un lup. Dacă jucătorul a prins toți iepurii înaintea lupilor acesta câștigă meciul.

fig. 33 Interfață joc video EvoAgents - Rabbit Catcher

O grijă de care jucătorul trebuie să țină cont o reprezintă pierderea vieții. Lupii sunt foarte fioroși, aceștia încep să atace imediat ce te apropii de ei. Dacă un lup te atacă, pierzi două puncte de viață la fiecare atac înregistrat. Dacă pierzi toată viața, jocul este terminat. Atunci când se intră în această stare, jucătorul se mai poate plimba în lume iar dacă prinde un iepure, acesta se adaugă la scor dar nu are importanță pentru deciderea stării jocului. Pentru a relua o nouă rundă, este necesară resetarea completă a lumii, astfel regenerând agenții din joc.

Controlul personajului este unul obișnuit, stil *third–person,* folosind taste precum „W”, „A”, „S”, „D”, pentru a alege direcția de deplasare, respectiv, tasta „*Shift*” pentru a comuta între mers și alergare. Agenții inteligenți ai acestui joc sunt foarte rapizi, deci, șansele de a prinde un iepure sunt cu mult mai mare dacă fugi după ei. Pentru a te uita într-o direcție, mouse-ul este cel care permite acest lucru. Nu trebuie să folosești butoanele deoarece jocul nu are elemente de *shooter*, dacă un lup vine după tine trebuie să fugi și să îl eviți. Nu poți să îl ataci pentru a te apăra, singura cale de a scăpa de ele este alergatul.

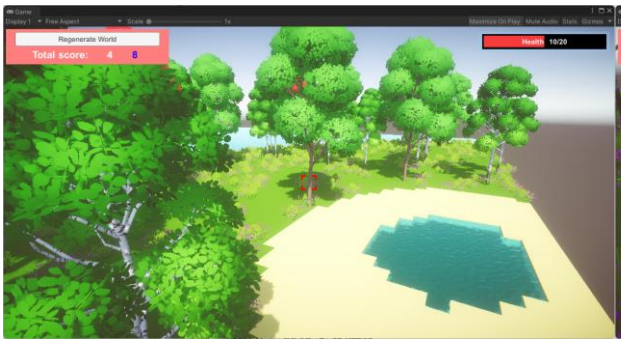


fig. 34 Jocul are posibilitatea de a putea zbura cu drona

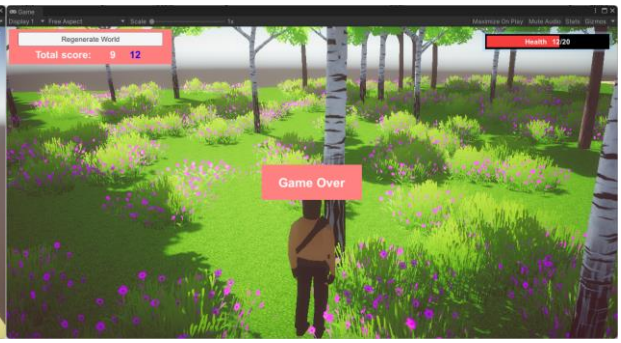


fig. 35 Imaginea de game over al jocului

Un mod util de a vedea unde se află animalele este acela de a comuta pe o dronă; o problemă este că devii vulnerabil dar nici nu poți prinde iepurii. Aceștia sunt considerați prinși atunci când ajungi foarte aproape de ei, fiind șterși din lume. Un iepure prins reprezintă un punct de scor.

Această dronă se poate controla folosind tastatura și mouse-ul. Dacă mișcați mouse-ul, atunci drona își va schimba direcția, iar cu tastele „W”, „A”, „S”, „D” o poți deplasa în lume. Pentru a putea ridica sau coborî drona, există posibilitatea de a folosii tasta „*Space*” (înalță drona), respectiv tasta „F” (coboară drona). Comutarea între jucător și aceasta se realizează folosind tasta „Q”.

Pentru a putea ieși din joc trebuie să fie apăsată tasta „*Escape*” (pentru a debloca mouse-ul) și din noua aceeași tastă pentru a închide jocul (sau combinația de taste Alt-F4).

CAPITOLUL 5 CONCLUZII FINALE

Utilizarea algoritmilor bazați pe arbori de comportament (Behaviour Trees, BT) aduce multe avantaje dezvoltatorilor care doresc să implementeze agenți inteligenți. Simplitatea, modularitatea și extensibilitatea sunt factorii cheie care au permis dezvoltarea accelerată a acestei ramuri a inteligenței artificiale.

În acest capitol se vor prezenta obiectivele atinse și exemple de aplicații și dezvoltări ulterioare. În încheiere se va prezenta experiența acumulată de către mine în timpul dezvoltării acestui proiect.

5.1 OBIECTIVELE ATINSE

În această lucrare mi-am propus, în primul rând, sa învăț dezvoltarea jocurilor, apoi am dorit să aflu cum se scriu agenți inteligenți specializați pe această ramură. În urma căutărilor realizate pe internet despre inteligența artificială, am descoperit că algoritmul principal, folosit cel mai des, în momentul actual, este cel al arborilor de comportament, deoarece sunt modulari, și permit dezvoltarea rapidă a unor comportamente. O altă cercetare ce se realizează foarte intens este aceea de a dezvolta agenți inteligenți bazați pe utilitate folosind învățarea prin întărire și rețelele neuronale artificiale.

Prin această lucrare de licență am dorit să dezvolt o nouă abordare de a scrie agenți inteligenți. Nu am știut absolut nimic despre cum se dezvoltă un joc video, și nici ideea de proiect nu am avut. Într-un final, după multe încercări, am reușit să dezvolt această bibliotecă ce permite scrierea de arbori de comportament într-un mod declarativ. Obiectivele propuse de mine au fost următoarele:

1) *Învățarea dezvoltării jocurilor video.* De mic am fost pasionat de jocurile pentru calculator, dar și de acest domeniu, al calculatoarelor. Din acest motiv am ales informatica și această temă de licență. Am dorit foarte tare să învăț cum se dezvoltă jocurile video și aplicațiile de calculator. Pot spune că mi-am atins acest obiectiv deoarece am realizat că este un domeniu vast și că învățarea nu înseamnă doar înțelegerea unui limbaj de programare, ci și înțelegerea unui mediu de dezvoltare precum Unity.

2) *Descoperirea modului de implementare a inteligenței artificiale.* Pentru că nu am cunoscut foarte bine ce doresc să dezvolt, tema aleasă a fost una generică ca să îmi permită schimbarea proiectului pe decursul implementării acestuia. Într-un final, am reușit să descopăr modul în care se pot dezvolta arborii de comportament. Pe piața de utilitare, pentru dezvoltare, există multe astfel de librării. Implementări de fațadă, vizuale, există peste tot și sunt scumpe. Utilizarea acestora nu te învață cum funcționează ele. Folosind documentații preluate de pe internet, pot spune că am reușit să învăț felul în care funcționează un astfel de arbore de comportament.

3) *Dezvoltarea unui agent inteligent bazat pe utilitate.* Acest obiectiv nu mi l-am atins deoarece agenții dezvoltați sunt bazați pe scop, evenimentele nu conduc în mod direct la bunăstarea agentului inteligent. Eu îmi propusesem să scriu un algoritm ce folosește o rețea neuronală artificială pentru ca agentul să învețe din experiență cum trebuie să se apere de inamic. În implementarea mea, acest comportament nu este dobândit, el fiind unul determinist. (acționează după un algoritm ce poate fi determinat)

4) *Dezvoltarea unei aplicații a inteligenței artificiale.* Obiectivul a fost acela de a prezenta intuitiv felul în care agenții inteligenți sunt dezvoltați în jocurile video. Mi-am dorit să aflu cum se realizează acest lucru, și după multe documentații citite am reușit să aflu. Dezvoltarea lor se realizează scriind, și cunoscând funcționalitatea, unor arbori de comportament, automate cu stări finite sau implementări ale unor rețele neuronale artificiale.

5.2 APLICAȚII ȘI ÎN ALTE DOMENII

Proiectul realizat are de fapt la bază domeniul roboticii, și nu cel al jocurilor video. Primele jocuri video nici nu foloseau arbori de comportament. Inițial utilizarea automatelor cu stări finite era cea mai utilizată în domeniul aferent lucrării de licență.

În robotică, arborii de comportament sunt utilizați pentru a definii un scop al unui robot. De exemplu, un braț robotic dorește să mută obiecte de pe o bandă rulantă, pe alta. Acesta este scopul lui. Arborele de comportament descrie modul în care detectează obiectul, îl apucă atunci când este în apropiere și îl pune în alt loc.

În planificarea automată a scopurilor unor agenți inteligenți autonomi, BT sunt utilizați folosind algoritmul de „Planifică și acționează folosind arbori de comportament" (PA-BT). Acesta este un algoritm ce permite planificarea unui scop folosind construcția inversă a unui arbore de comportament. În acest fel, un robot cunoaște doar setul de acțiuni și scopul la care dorește să ajungă. Experimentând, acesta în timp își dezvoltă un arbore de comportament care să îl ajute în luarea eficientă de decizii. Algoritmul este inspirat din cel de *tree pruning* aplicat asupra arborilor de decizie.

De exemplu, o mașinuță robot dorește să mute un cub verde într-o poziție cunoscută. El știe că poate să se deplaseze, să ridice un obiect, să împingă un obiect și să îl lase într-un loc. Inițial are ca plan deplasarea către obiect, ridicarea lui, deplasarea către locul în care trebuie să lase obiectul și eliberarea acestuia *(MoveTo(c)->Pick(c)->MoveTo(g)->Drop()).* În urma pașilor acesta întâlnește un eveniment neobișnuit, deplasarea unei mingi roșii înspre robot de către un alt agent extern. Această minge blochează calea agentului și el trebuie să se adapteze. Arborele său de comportament nu a întâmpinat această situație, astfel, prin utilizarea algoritmului PA-BT, agentul își va readapta BT astfel încât să poate ocolii obstacolul. Noul arbore va determina următorul set de acțiuni ce conduc la scop: *MoveTo(s)->Push(s)->MoveTo(c)->Pick(c)->MoveTo(g)->Drop().* (Michele & Petter, 2018)

În domeniul învățării automate, arborii de comportament sunt utilizați pentru a planifica scopul agentului ca apoi acesta să se folosească de arbore pentru a învăța un anumit comportament. Un mod prin care mai mulți agenți inteligenți pot să se adapteze folosind procedeul de învățare automată este acela de a utiliza algoritmi genetici. Fiecare agent inteligent are determinată o valoare de utilitate, fie calculată stocastic, fie în alt mod. Doi agenți sunt considerați potriviți dacă aceștia au un scor foarte mare. Prin procedeele de „*mutarea / alterarea genelor*" (*mutation*), respectiv „*combinarea lor*" (*cross-over*) se obține un nou BT complet unic și diferit dar care respectă trăsăturile părinților. Acest algoritm se numește „*programare genetică aplicată arborilor de comportament"* (GP-BT) (Michele & Petter, 2018)

5.3 DEZVOLTĂRI ULTERIOARE

Biblioteca software dezvoltată în acest proiect, aflată în stadiul actual, este una la început. Dorința mea a fost realizarea unei aplicații a arborilor de comportament. Implementările permise de aceasta sunt următoarele:

1) *Canale de comunicare între mai mulți agenți inteligenți.* Prin intermediul claselor mediator, un agent inteligent este capabil să interacționeze cu alți agenți. Fiecare eveniment care are loc în mediu este perceput de către un agent și îi poate informa pe restul prin mesaje. Un mesaj reprezintă un pachet de informație utilă pentru alegerea unei decizii care salvează agentul. Acest procedeu poate fi îmbunătățit prins rescrierea claselor sau extinderea acestora folosind noi metode sau funcții ce permit comunicarea.

2) *Clasele de tip Blackboard permit salvarea informației în etapa de planificare a deciziei.* Structură tip dicționar, un blackboard permite implementarea unei zone de memorie în care un agent inteligent salvează informația preluată din mediu. Acesta o va putea accesa pentru a putea lua noi decizii. Îmbunătățirea acestora ar însemna permiterea de a salva mai multe date pe un timp mai lung și chiar refolosirea acestora pentru a putea lua decizii mai bune.

3) *Arbori de comportament pot fi dezvoltați pentru a rezolva cerințe specifice unui joc.* Dezvoltatorii pot extinde implementarea arborilor de comportament pentru a se potrivi necesităților acestora. Realizarea de planuri pentru un agent și implementarea nodurilor specifice sunt elemente cheie ce permit scrierea agenților inteligenți.

Dezvoltării ulterioare ale acestui proiect pot fi extinderea funcționalităților librăriei API prin permiterea scrierilor de arbori mult mai complecși și îmbunătățirea algoritmilor utilizați. Noi funcționalități aduse pot fi implementarea unor rețele neuronale artificiale și utilizarea învățării prin întărire. Acest lucru permite scrierea agenților inteligenți capabili să se adapteze la condițiile de joc.

O altă dezvoltare ulterioară ar fi reorganizarea codului. Actual, acesta este unul limitat și greu de menținut. Pentru a putea extinde funcționalitatea acestuia, o posibilă rescriere ar putea însemna ca nodurile arborelui să fie reimplementate sub formă de clase de obiecte. Acest lucru permite utilizarea de metode specifice cazurilor particulare ce pot apărea.

5.4 EXPERIENȚA ACUMULATĂ

Dezvoltarea acestui proiect a ajutat la acumularea unei experiențe bogate ce mă va ajuta pe viitor. Pot spune că am reușit în felul acesta să fac o practică de specialitate. La un loc de muncă se cere o experiență de cel puțin doi ani. Pentru a putea dovedi această experiență, proiectele personale reprezintă singura modalitate prin care se poate acumula această experiență.

Lucrul într-un mediu de dezvoltare popular permite învățarea abilităților de a implementa jocuri video la un nivel acceptat de angajatori. Multe firme de dezvoltare de jocuri video precum *Ubisoft* folosesc medii proprii. Cunoașterea unui *game engine* permite învățarea rapidă a ceea ce se folosește pe piață.

Dezvoltarea unei implementări a arborilor de comportament a ajutat la înțelegerea funcționalității acestora. Știind felul de operare a lor permite o ușoară utilizare a librăriilor deja utilizate. Unele implementări dispun de explicații scurte despre cum se folosesc aceasta. Nu este suficientă cunoașterea utilizării deoarece este necesar și modul în care acestea operează. Nodurile *compuse*, *decoratoare* și *acțiuni* sunt complexe și felul de utilizare a lor este dependentă de experiența programatorului.

Agenții inteligenți bazați pe scop sunt cei mai utilizați în industria jocurilor video deoarece sunt cel mai ușor de implementat. *Modularitatea* de care dispune un BT este avantajoasă pentru menținerea agenților inteligenți mereu actualizați cu cerințele publicului. Experiența dobândită prin implementarea acestora ajută dezvoltatorii să cunoască moduri alternative de scriere a agenților inteligenți.

Învățarea dezvoltării jocurilor, mi-au permis să aflu cum se scriu agenți inteligenți de tipul personajelor controlate de un calculator. În urma căutărilor realizate pe internet despre acest domeniu, am descoperit că modul principal, în momentul actual, de a realiza astfel de personaje sunt arborii de comportament, deoarece sunt modulari, și permit dezvoltarea rapidă a unor comportamente. Experiența dobândită îmi va permite pe viitor să cunosc cerințele pe piața muncii și voi știi ce va trebui să fac dacă întâmpin astfel de dificultăți.

BIBLIOGRAFIE

Blom, L. v. (2007). Map-Adaptive Artificial Intelligence for Video Games. *Research Gate*.

CySharp - UniTask. (2021, Septembrie). Preluat de pe GitHub: https://github.com/Cysharp/UniTask

Donkey Kong. (2021, Septembrie). Preluat de pe Wikipedia.org: https://en.wikipedia.org/wiki/Donkey_Kong

Indy 800. (2021, Septembrie). Preluat de pe Wikipedia.org: https://en.wikipedia.org/wiki/Indy_800

Lammers, S. (1989). *Programmers at Work: Interviews with 19 programmers who shaped the computer industry.* Tempus Books. doi:10.1002/9781119418504

Luond, D. (2019). Finite-state machine for turn-based combat in video games. *Digital Ideation, Lucerne University of Applied Sciences and Arts "Hochschule Luzern"*.

Mateas, M. (2003). *Expressive AI: Games and Artificial Intelligence.* The Georgia Institute of Technology, Atlanta, GA USA. Preluat pe 08 26, 2021, de pe https://users.soe.ucsc.edu/~michaelm/tenurereview/publications/mateas-digra2003.pdf

Michele, C., & Petter, O. (2018). *Behaviour Trees in Robotics and AI.* doi:arXiv:1709.00084

Millington, I., & Funge, J. (2006). *Artificial Intelligence for Games.*

Nicoară, S. (2020, Octombrie). Inteligentă artificială (curs). Ploiești, Prahova, Romania.

Pac-Man. (2021, Spetembrie). Preluat de pe Wokipedia.org: https://ro.wikipedia.org/wiki/Pac-Man

Ramiro, A., Sebastian, G., & Alejandro, G. (2019). An event-driven behavior trees extension to facilitate non-player multiagent coordination in video games. *Expert Systems with Applications*.

Smuts, A. (2005). Are video games art? *Contemporary Aesthetics, vol 3*. Preluat de pe https://digitalcommons.risd.edu/liberalarts_contempaesthetics/vol3/iss1/6/

Stack Exchange - Event-driven finite state machine. (2021, Septembrie). Preluat de pe Stack Exchange: https://codereview.stackexchange.com/questions/143726/event-driven-finite-state-machine-dsl-in-kotlin

Super Mario World. (2021, Septembrie). Preluat de pe Wikipedia.org: https://en.wikipedia.org/wiki/Super_Mario_World

Time Magazine - Duck Hunt. (2021, Septembrie). Preluat de pe Time.com: https://time.com/3640170/nintendo-wii-u-duck-hunt/

Wolf, J. (2008). *The video game Explosion: A history from PONG to PlayStation and beyond.* London: Greenwood Press.

ANEXE

```
public enum BehaviourStatus { Running, Success, Failure };
public delegate UniTask<BehaviourStatus> InvokeAction(BehaviourTask self, CancellationToken token =
default);
```

```
public partial class BehaviourTask {
    private readonly List<BehaviourTask> _tasks;
    private InvokeAction OnInvoke { get; }
    private CancellationToken Token { get; set; }
    public List<BehaviourTask> Tasks { get => _tasks; }
    public BehaviourStatus Status { get; private set; }
    protected BehaviourTask(InvokeAction _invoke, CancellationToken _token = default) {
        OnInvoke = _invoke;
        Token = _token;
        Status = BehaviourStatus.Running;
        _tasks = new List<BehaviourTask>();
    }
    public Awaiter GetAwaiter() {
        return new Awaiter(this);
    }
    public void SetToken(CancellationToken token) => Token = token;
    public static BehaviourTask Run(InvokeAction _invoke, CancellationToken _token = default) {
        return new BehaviourTask(_invoke, _token);
    }
}
```

```
public readonly struct Awaiter : ICriticalNotifyCompletion {
    readonly UniTask<BehaviourStatus> task;
    readonly UniTask<BehaviourStatus>.Awaiter awaiter;

    public Awaiter(BehaviourTask _task) {
        task = _task.OnInvoke(_task, _task.Token);
        awaiter = task.GetAwaiter();
    }

    public bool IsCompleted { get => task.Status != UniTaskStatus.Pending; }
    public BehaviourStatus GetResult() {
        return awaiter.GetResult();
    }
    public void OnCompleted(Action continuation) {
        ((INotifyCompletion)awaiter).OnCompleted(continuation);
    }
    public void UnsafeOnCompleted(Action continuation) {
        ((ICriticalNotifyCompletion)awaiter).UnsafeOnCompleted(continuation);
    }
}
```

```
public static partial class BehaviourTaskExtensions {
    public static BehaviourTask Register(this BehaviourTask task, BehaviourTask child) {
        task.Tasks.Add(child);
        return task;
    }
    public static BehaviourTask Unregister(this BehaviourTask task, BehaviourTask child){
        task.Tasks.Remove(child);
        return task;
    }
    public static BehaviourTask Unregister(this BehaviourTask task) {
        task.Tasks.RemoveAt(task.Tasks.Count - 1);
        return task;
    }
}
```

```
[DisallowMultipleComponent]
[AddComponentMenu("EvoAgents/Behaviours/Blackboard")]
public class Blackboard: MonoBehaviour, IBlackboard<Blackboard> {
    [SerializeField] private AgentGenome _genome;
    [SerializeField] private Mediator _mediator;
    [SerializeField] private Guid guid;
    private Dictionary<string, object> _blackboard;
    public string GUID => guid.ToString();
    public Genome Genome => _genome.Genome;

    private readonly struct BlackboardMessage : IMessage<Blackboard> {
        public string MessageName { get; }
        public object MessageValue { get; }
        public void UnpackMessage(Blackboard blackboard) {
            blackboard[MessageName] = MessageValue;
        }
        public BlackboardMessage(string name, object value) {
            MessageName = name;
            MessageValue = value;
        }
    }
    public static IMessage<Blackboard> CreateMessage<TObject>(string name, TObject value)
        => new BlackboardMessage(name, value);
    public object this[string name] {
        get {
            if (_blackboard.ContainsKey(name))
                return _blackboard[name];
            return null;
        }
        set {
            if(name != "") {
                if (_blackboard.ContainsKey(name)) {
                    if (value is null)
                        _blackboard.Remove(name);
                    else _blackboard[name] = value;
                }
                else _blackboard.Add(name, value);
            }
        }
    }
    public IMediator<Blackboard> Mediator => _mediator;

    public bool Equals(Blackboard other) {
        return guid.Equals(other.GUID);
    }
    public void ReceiveMessage(IMessage<Blackboard> message) {
        message.UnpackMessage(this);
    }
    public void SendMessage(IMessage<Blackboard> message, Blackboard to) {
        Mediator.SendMessage(message, to);
    }
    public void SubscribeTo(IMediator<Blackboard> mediator) {
        mediator.AddSubscriber(this);
        _mediator = (Mediator)mediator;
    }
    public void UnsubscribeFrom(IMediator<Blackboard> mediator) {
        mediator.RemoveSubscriber(this);
        _mediator = null;
    }
    private void Awake() {
        guid = Guid.NewGuid();
        _blackboard = new Dictionary<string, object> { { "", true } };
    }
    public void BroadcastMessage(IMessage<Blackboard> message) {
        Mediator.BroadcastMessage(message);
    }
    public void Clear() {
        _blackboard.Clear();
        _blackboard.Add("", true);
    }
    private void OnDestroy() {
        UnsubscribeFrom(_mediator);
    }
}
```

```
[DisallowMultipleComponent]
[AddComponentMenu("EvoAgents/Behaviours/Mediator")]
public class Mediator : MonoBehaviour, IMediator<Blackboard> {
    [SerializeField] private List<Blackboard> subscribers;
    public IReadOnlyCollection<Blackboard> Blackboards => subscribers;
    public Blackboard this[int index] => subscribers[index];

    public void AddSubscriber(Blackboard blackboard) {
        if (subscribers is null) subscribers = new List<Blackboard> { blackboard };
        else subscribers.Add(blackboard);
    }

    public void BroadcastMessage(IMessage<Blackboard> message) {
        if (subscribers is null) return;
        foreach (var subscriber in subscribers)
            subscriber.ReceiveMessage(message);
    }

    public void RemoveSubscriber(Blackboard blackboard) {
        if (subscribers is null) return;
        var index = subscribers.FindIndex((target) => target == blackboard);
        if(index > -1) subscribers.RemoveAt(index);
    }

    public void SendMessage(IMessage<Blackboard> message, Blackboard to) {
        if (subscribers is null || subscribers.Count == 0) return;
        Blackboard found = null;
        foreach (var subscriber in subscribers)
            if (subscriber.GUID == to.GUID) found = subscriber;
        if (found is null) return;
        found.ReceiveMessage(message);
    }
}
```

```
public interface IBlackboard<TBlackboard>: IEquatable<TBlackboard>
where TBlackboard: IBlackboard<TBlackboard> {
    public string GUID { get; }
    public object this[string key] { get; set; }
    public IMediator<TBlackboard> Mediator { get; }
    public bool CompareTag(string tag);
    public void SubscribeTo(IMediator<TBlackboard> mediator);
    public void UnsubscribeFrom(IMediator<TBlackboard> mediator);
    public void SendMessage(IMessage<TBlackboard> message, TBlackboard to);
    public void BroadcastMessage(IMessage<TBlackboard> message);
    public void ReceiveMessage(IMessage<TBlackboard> message);
}
```

```
public interface IMessage<TBlackboard> where TBlackboard : IBlackboard<TBlackboard> {
    public void UnpackMessage(TBlackboard blackboard);
}
public interface IMediator<TBlackboard> where TBlackboard : IBlackboard<TBlackboard> {
    public IReadOnlyCollection<TBlackboard> Blackboards { get; }
    public void AddSubscriber(TBlackboard blackboard);
    public void RemoveSubscriber(TBlackboard blackboard);
    public void SendMessage(IMessage<TBlackboard> message, TBlackboard to);
    public void BroadcastMessage(IMessage<TBlackboard> message);
}
```

```
[System.Serializable]
[CreateAssetMenu(fileName = "New Genome", menuName = "EvoAgents/Agent Genome")]
public class AgentGenome : ScriptableObject {
    [SerializeField] private Genome m_Genome;
    public Genome Genome => m_Genome;
}
```

```
[System.Serializable]
public struct Genome {
    [Tooltip("Which distance this agent sense food/pray?")]
    public float Scent;

    [Tooltip("Which distance this agent sense predators")]
    public float Awareness;

    [Tooltip("How many seconds this agent wander before tiredness?")]
    public float MaxStamina;

    [Tooltip("Chance to attack another agent (in %)")]
    [Range(0f, 1f)]
    public float Aggression;

    [Tooltip("Dominance of this agent. Higher value means more dominant than others.")]
    public float Dominance;
}
```

```
namespace EvoAgents.Behaviours {
    using BehaviourStack = Stack<BehaviourTask>;
    public struct TaskGenerator
    {
        private readonly BehaviourStack currentPointer;
        public BehaviourTask Task => currentPointer.Peek();
        public static TaskGenerator Begin() { return new TaskGenerator(false); }
        private TaskGenerator(bool _ = false) => currentPointer = new BehaviourStack();

        public TaskGenerator PushTask(BehaviourTask _task) {
            currentPointer.Push(_task);
            return this;
        }
        public TaskGenerator End() {
            if(currentPointer.Count >= 2) {
                var task = currentPointer.Pop();
                currentPointer.Peek().Register(task);
            }
            return this;
        }
        public BehaviourTask Generate() => currentPointer.Pop();
    }
}
```

```
public static class BehaviourTaskDecorator {
    public static TaskGenerator Conditional(this TaskGenerator builder, Func<bool> condition) =>
        builder.PushTask(BehaviourTask.Run(async (self, token) => {
            if (token.IsCancellationRequested) return BehaviourStatus.Failure;
            if (!condition()) return BehaviourStatus.Failure;
            return await self.Tasks[0];
        }));

    public static TaskGenerator Conditional<TBlackboard>(this TaskGenerator builder, TBlackboard
blackboard, string key)
        where TBlackboard : IBlackboard<TBlackboard> =>
        builder.Conditional(() => {
            bool? result = (bool?)blackboard[key];
            return result.HasValue && result.Value;
        });

    public static TaskGenerator RepeatForever(this TaskGenerator builder) =>
        builder.PushTask(BehaviourTask.Run(async (self, token) => {
            self.Tasks[0].SetToken(token);
            while (true) {
                await self.Tasks[0];
                if (token.IsCancellationRequested) break;
                await UniTask.WaitForEndOfFrame();
            }
            return BehaviourStatus.Failure;
        }));
    public static TaskGenerator RepeatUntil(this TaskGenerator builder, BehaviourStatus
behaviourStatus) =>
        builder.PushTask(BehaviourTask.Run(async (self, token) => {
            self.Tasks[0].SetToken(token);
            var status = await self.Tasks[0];
            while (status != behaviourStatus && !token.IsCancellationRequested) {
                status = await self.Tasks[0];
                await UniTask.WaitForEndOfFrame();
            }
            return behaviourStatus;
        }));
    public static TaskGenerator RepeatWhile(this TaskGenerator builder, BehaviourStatus
behaviourStatus) =>
        builder.PushTask(BehaviourTask.Run(async (self, token) => {
            self.Tasks[0].SetToken(token);
            var status = await self.Tasks[0];
            while (status == behaviourStatus && !token.IsCancellationRequested) {
                status = await self.Tasks[0];
                await UniTask.WaitForEndOfFrame();
            }
            return status;
        }));
}
```

```
public static class BehaviourTaskComposite
{
    public static TaskGenerator Sequencer(this TaskGenerator builder)
    {
        return builder.PushTask(BehaviourTask.Run(async (self, token) => {
            foreach (var task in self.Tasks) {
                if (token.IsCancellationRequested)
                    return BehaviourStatus.Failure;

                task.SetToken(token);
                var status = await task;
                if (status == BehaviourStatus.Failure)
                    return BehaviourStatus.Failure;
            }
            return BehaviourStatus.Success;
        }));
    }

    public static TaskGenerator Selector(this TaskGenerator builder)
    {
        return builder.PushTask(BehaviourTask.Run(async (self, token) => {
            foreach (var task in self.Tasks) {
                if (token.IsCancellationRequested)
                    return BehaviourStatus.Failure;

                task.SetToken(token);
                var status = await task;
                if (status == BehaviourStatus.Success)
                    return BehaviourStatus.Success;
            }
            return BehaviourStatus.Failure;
        }));
    }
    public static TaskGenerator RandomSelector(this TaskGenerator builder)
    {
        return builder.PushTask(BehaviourTask.Run(async (self, token) => {
            if (token.IsCancellationRequested)
                return BehaviourStatus.Failure;
            var task = self.Tasks[UnityEngine.Random.Range(0, self.Tasks.Count)];

            task.SetToken(token);
            var status = await task;
            if (status == BehaviourStatus.Success)
                return BehaviourStatus.Success;
            return BehaviourStatus.Failure;
        }));
    }

    public static TaskGenerator Parallel(this TaskGenerator builder)
    {
        return builder.PushTask(BehaviourTask.Run(async (self, token) => {
            var tasks = new UniTask<BehaviourStatus>[self.Tasks.Count];
            for (int i = 0; i < self.Tasks.Count; i++)
                tasks[i] = UniTask.Create(async () => await self.Tasks[i]);
            await UniTask.WhenAll(tasks);
            return BehaviourStatus.Success;
        }));
    }
}
```

```
public BehaviourTask Wander(float speed, float distance) =>
    TaskGenerator.Begin()
        .Selector()
            .Conditional(() => (float)Blackboard["Energy"]>0&&!(bool)Blackboard["Resting"])
                .Sequencer()
                    .PlayAnimation(Animation, "Walking", AnimationCategory.WalkingAnimation)
                    .UpdateBlackboard(Blackboard, "Energy", () => {
                        float depletionRate = -0.05f * Random.value;
                        return depletionRate* Blackboard.Genome.MaxStamina * Time.deltaTime;
                    })
                    .Wander(Agent, speed, distance)
                .End()
            .End()
            .Conditional(() => (float)Blackboard["Energy"] < 0)
                .SetBlackboard(Blackboard, "Resting", () => true)
            .End()
        .Generate();
```

```
public static class BehaviourTaskActions
{
    public static TaskGenerator Action(this TaskGenerator builder, InvokeAction invokeAction) {
        builder.Task.Register(BehaviourTask.Run(invokeAction));
        return builder;
    }

    public static TaskGenerator Action(this TaskGenerator builder, BehaviourTask taskAction) {
        builder.Task.Register(taskAction);
        return builder;
    }

    public static TaskGenerator WaitFor(this TaskGenerator builder, float waitTimeSeconds) =>

        builder.Action(async (self, token) => {
            if (token.IsCancellationRequested)
                return BehaviourStatus.Failure;

            await UniTask.Delay(TimeSpan.FromSeconds(waitTimeSeconds), DelayType.DeltaTime,
PlayerLoopTiming.Update);
            return BehaviourStatus.Success;
        });

    public static TaskGenerator SetBlackboard<TBlackboard, TObject>(this TaskGenerator builder,
TBlackboard blackboard, string key, Func<TObject> factory)
        where TBlackboard : IBlackboard<TBlackboard> =>

        builder.Action((self, token) => {
            if (token.IsCancellationRequested) return UniTask.FromResult(BehaviourStatus.Failure);
blackboard.ReceiveMessage((IMessage<TBlackboard>)Blackboard.CreateMessage(key, factory()));
            return UniTask.FromResult(BehaviourStatus.Success);
        });

    public static TaskGenerator Broadcast<TBlackboard, TMessage>(this TaskGenerator builder,
TBlackboard blackboard, TMessage message)
        where TBlackboard : IBlackboard<TBlackboard>
        where TMessage : IMessage<TBlackboard> =>

        builder.Action((self, token) => {
            if (token.IsCancellationRequested) return UniTask.FromResult(BehaviourStatus.Failure);
            blackboard.BroadcastMessage(message);
            return UniTask.FromResult(BehaviourStatus.Success);
        });

    public static TaskGenerator SendMessage<TBlackboard, TMessage>(this TaskGenerator builder,
TBlackboard from, TMessage message, Func<TBlackboard> to)
        where TBlackboard : IBlackboard<TBlackboard>
        where TMessage : IMessage<TBlackboard> =>

        builder.Action((self, token) => {
            if (token.IsCancellationRequested) return UniTask.FromResult(BehaviourStatus.Failure);
            from.SendMessage(message, to());
            return UniTask.FromResult(BehaviourStatus.Success);
        });
}
```

```
// Rabbit behaviour tree implementation
public override BehaviourTask Behaviour =>
    TaskGenerator.Begin()
        .RepeatForever()
            .Selector()
                .Action(Rest())
                .Conditional(() => !(bool)Blackboard["Enemy"])
                    .Parallel()
                        .Action(Wander(2f, 18f))
                        .Action((self, token) => { /* Check for enemy */ })
                    .End()
                .End()
                .Selector()
       .Conditional(() =>
(bool)Blackboard["Enemy"]&&(float)Blackboard["Energy"]>0&&!(bool)Blackboard["Resting"])
                        .Parallel()
                            .Action((self, token) => { /* Find Enemy */ })
                            .Sequencer()
                            .PlayAnimation(Animation, "Running", AnimationCategory.WalkingAnimation)
                                .UpdateBlackboard(Blackboard, "Energy", () => { . . . })
                                .Wander(Agent, 6f, 18f)
                                .ApplyForce(Agent, 1f, () => {
                                    if (Blackboard["EvadeEnemy"] is null) return Vector3.zero;
                                    return (Vector3)Blackboard["EvadeEnemy"];
                                })
                            .End()
                        .End()
                    .End()
                    .Conditional(() => (float)Blackboard["Energy"] < 0)
                        .SetBlackboard(Blackboard, "Resting", () => true)
                    .End()
                .End()
            .End()
        .Generate();
```

```
public BehaviourTask Rest() =>
    TaskGenerator.Begin()
        .Conditional(() => (float)Blackboard["Energy"]<0&&(bool)Blackboard["Resting"])
            .Sequencer()
                .StopAnimation(Animation)
                .StopMovement(Agent)
                .WaitUntil(() => (float) Blackboard["Energy"] > .99f*Blackboard.Genome.MaxStamina)
                    .UpdateBlackboard(Blackboard, "Energy", () => {
                        float restoreRate = 0.25f * Random.value;
                        return restoreRate * Blackboard.Genome.MaxStamina * Time.deltaTime;
                    })
                .End()
                .Conditional(() => (float) Blackboard["Energy"] > .99f*Blackboard.Genome.MaxStamina)
                    .SetBlackboard(Blackboard, "Resting", () => false)
                .End()
            .End()
        .Generate();
```

```
// Wolf behaviour tree implementation
public override BehaviourTask Behaviour =>
    TaskGenerator.Begin()
        .RepeatForever()
            .Selector()
                .Action(Rest())
                .Conditional(() => !(bool)Blackboard["Pray"])
                    .Parallel()
                        .Action(Wander(2.3f, 25f))
                        .Probability(Blackboard.Genome.Aggression)
                            .Action((self, token) => {/* Check for pray */})
                        .End()
                    .End()
                .End()
                .Selector().Conditional(() =>
(bool)Blackboard["Pray"]&&(float)Blackboard["Energy"]>0&&!(bool)Blackboard["Resting"])
                        .Parallel()
                            .Action((self, token) => { /* Find Pray */ })
                            .Sequencer()
                            .PlayAnimation(Animation, "Running", AnimationCategory.WalkingAnimation)
                                .UpdateBlackboard(Blackboard, "Energy", () =>{ . . . })
                                .Wander(Agent, 4.8f, 25f)
                                .ApplyForce(Agent, 1f, () => { . . . })
                            .End()
                        .End()
                    .End()
                    .Conditional(() => (float)Blackboard["Energy"] < 0)
                        .SetBlackboard(Blackboard, "Resting", () => true)
                    .End()
                .End()
            .End()
        .Generate();
```