IMPLEMENTACIJA GOAP ALGORITMA I PROTIP IGRE S GOAP SUSTAVOM

Tafra, Ivan

SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Specijalistički diplomski stručni studij Informacijske tehnologije

IVAN TAFRA

Z A V R Š N I R A D

IMPLEMENTACIJA GOAP ALGORITMA I PROTOTIP IGRE S GOAP SUSTAVOM

Split, rujan 2021.

SVEUČILIŠTE U SPLITU
SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Specijalistički diplomski stručni studij Informacijske tehnologije

Predmet: Mobilne tehnologije

Z A V R Š N I R A D

Kandidat: Ivan Tafra

Naslov rada: Implementacija GOAP algoritma i prototip igre s GOAP sustavom

Mentor: Marina Rodić, predavač

Split, rujan 2021.

SADRŽAJ

Sažetak

U ovom je radu prikazan ciklus razvoja prototipa igre kao i implementacija pametnog algoritma za ponašanje agenata. *GOAP* (*Goal oriented action planner*) algoritam ponašanja odabran je i implementiran kao Unity paket. Posebna se pozornost posvetila na generičku implementaciju algoritma, kao i na verifikaciju istog, kako bi se paket mogao primijeniti na proizvoljni tip igre. Tip implementirane igre u ovom je radu simulator kuhanja s pogledom od gore. Igra implementira *GOAP* Unity paket napravljen u ovom radu, a može se igrati na PC i Android platformi s različitim ulaznim uređajima. Implementacija igre i paketa napravljena je u Unity razvojnom okruženju. Za verzioniranje je korišten Git. Cilj je rada prikazati kroz koje se sve izazove razvojni inženjer susreće prilikom implementacije igre i generičkih proširenja za igre.

Ključne riječi: GOAP, algoritam ponašanja, igra, Unity

Summary

Game prototype with GOAP system and implementation of GOAP algorithm

In this diploma thesis the game prototyping cycle, as well as the implementation of a smart algorithm for agent behavior is presented. The GOAP behavioral algorithm was selected and implemented as a Unity package. Considerable emphasis has been placed on the algorithm generic implementation and its verification in order to apply the package to any game type. The game type implemented in this thesis is a cooking simulator with a top - down view. The game implements the GOAP Unity package developed in this thesis, and can be played on PC and Android platforms with different input devices. The game and package were implemented in the Unity game engine. Git software was used for versioning. The purpose of this thesis is to present all the challenges a developer encounters when implementing a game and generic game extensions for game engine.

Keywords: GOAP, behavior selection algorithm, game, Unity

1. Uvod

Od početka razvoja videoigara korisnici su željeli dobiti više za uloženi novac. U počecima su razvojni inženjeri razvijali igre za specifičnu konzolu. Svaka konzola imala je ograničenja i svoj programski jezik, asembler (engl. *assembler*). U to vrijeme na razvojne inženjere gledalo se jednako kao i na radnike na proizvodnoj traci koji sastavljaju konzolu. Nisu se smjeli kreditirati, a i primanja su im bila fiksna, bez obzira na broj prodanih primjeraka igre.

1979. godine četiri razvojna inženjera pobunila su se i osnovala prvi neovisni razvojni studio. To je studio koji razvija igre, a ne potječe od proizvođača konzole. Bio je to i prvi put da jedan tim razvija igre za više različitih platformi. Uskoro će se mnogi razvojni inženjeri okušati u tome s obzirom na to da su u razvoju igara vidjeli sljedeću zlatnu groznicu. Zbog toga dolazi do značajnog pada kvalitete igara pa i kraha tržišta videoigara za konzole u Americi.

1986. godine Nintendo lansira NES (*Nintendo Entertainment System*) konzolu s posebnim čipom koji onemogućava neovisnim razvojnim inženjerima samostalan razvoj igara za tu konzolu. To je bila prva konzola za koju su neovisni razvojni inženjeri trebali zatražiti dopuštenje od Nintenda za razvoj igara na istoj te samim time poštovati odredbe koje im je pripisao Nintendo. Na taj je način Nintendo osiguravao kvalitetu igara koje su se izdavale za konzolu te je i dodatno zarađivao jer je uzimao postotak provizije od neovisnih izdavača. Igre su se i dalje pisale u asembleru, a razvojni su inženjeri od početaka razvoja videoigara učili jedni od drugih i nastojali su poboljšati mehanike viđene u drugim igrama. Tako je i prva igra za prvu Atari konzolu, Pong, poboljšana verzija Odyssey igre. U igru su dodali zvuk, ubrzanje lopte s protekom vremena i rezultat. Poslije se pojavila prva igra s bodovnim sustavom, Space Invaders. Mnoge su igre uzeli taj sustav i unaprijedili ga ili prilagodili za svoju igru. Nakon toga uslijedile su igre kod kojih se svijet pomicao bočno, kako se glavni lik pomiče, tzv. *side scroller* žanr. Treba napomenuti da razvojni inženjeri nisu dijelili mehanike, već su svaki put kod razvoja nove igre počinjali od početka, uz eventualno kopiranje ponekih dijelova iz svojih starijih naslova.

No, 1992. godine skupina razvojnih inženjera predvođeni Johnom Carmackom razvila je razvojno okruženje za igre (engl. *game engine*) koji ima mogućnost 3D prikaza, kao i prvu igru u 3D-u, Wolfenstein 3D. To je bio prvi put da su drugi razvojni inženjeri tražili otkup prava za korištenje Carmakova razvojnog okruženja. Navedeno je razvojno okruženje

razvijeno u programskom jeziku C, a neki su dijelovi razvijeni u asembleru. S korištenjem razvojnog okruženja postalo je moguće brže razvijati igre jer razvojna okruženja sadrže mnoge značajke ključne za razvoj igara poput sustava za fiziku, grafiku, umjetnu inteligenciju, zvuk, animaciju i slično. Kako bi korisnik stekao što veću imerziju s igrom, potrebno je razviti pametne agente.

Agenti su likovi u igri kojim upravlja računalo. Izrada pametnih agenta kompleksan je zadatak u kojem pomažu algoritmi odabira ponašanja (engl. *behavior selection algorithm*). Jedan je od najčešće korištenih algoritama selektivnog ponašanja u igrama FSM (*finite state machine*). Međutim, s porastom stanja u kojem se agent može nalaziti, kao i s porastom mogućih prijelaza među stanjima, taj sustav vrlo brzo postane kompleksan i težak za dodavanje novog stanja ili prijelaza bez prouzrokovanja grešaka u kodu. Kako bi se otklonile mane FSM-a, razvili su se mnogi algoritmi za selektivno ponašanje. GOAP je jedan od njih.

U ovom radu prikazat će se implementacija generičkog GOAP algoritma te njegova primjena u razvojnom okruženju Unity. Današnji način izrade igara bitno se razlikuje od onoga s početka njihova razvoja. Prije dolaska pametnih mobitela na tržište bilo je skupo i neisplativo samostalno ili u manjem timu razvijati igre. Okruženja za igre bila su dosta skupa, a ona jeftinija bila su limitirana. Danas je situacija bitno drugačija i postoje mnogo malih poduzeća koja rade na igrama, ali danas si samo najveće kompanije mogu priuštiti izradu razvojnog okruženja za igre. Za individualce i manje timove danas nema potrebe za tim jer se mnoga razvojna okruženja mogu koristiti besplatno uz određenu maržu kad se ostvari određena godišnja dobit. Danas je bitno što ranije dobiti povratnu informaciju od igrača. Iz tog razloga mnoge igre izlaze na tržište već u alfa stanju, gdje su pokrivene osnovne mehanike igre, kako bi se ispitalo tržište i zainteresiranost za igru. Ispituje se i analizira što bi igrači voljeli vidjeti implementirano, izbačeno, prioritizirano i slično. Veliku ulogu za uspjeh na tržištu danas ima komunikacija s igračima kao i volja razvojnih inženjera da prilagode svoju viziju viziji igrača. Osim toga, bitna su i česta ažuriranja koja će ispravljati greške ili dodavati nove stvari u igru.

Ovaj je rad podijeljen na pet cjelina. U ovom je poglavlju kratko opisna povijest razvoja igara kao i motivacija za ovaj rad. U drugom poglavlju opisane su tehnologije koje su korištene za izradu ovog rada. Treće poglavlje opisuje GOAP algoritam, a podijeljeno je na četiri potpoglavlja. U njima se opisuje što je i kako radi GOAP algoritam, kako se implementirani paket može koristiti u drugim projektima, implementacija algoritma i paketa

i verifikacija razvijenog algoritma. U četvrtom se poglavlju opisuje implementacija igre i osnovna igriva petlja igre. U posljednjoj cjelini doneseni su zaključci kao i moguća proširenja za igru i paket u budućnosti.

2. Korištene tehnologije

2. 1. Unity

Unity je razvojno okružene za igre koje je 2005. godine razvio Unity Technologies. Razvojno okruženje napisano je u C i C++ programskom jeziku. Unity se može pohvaliti kao razvojno okruženje koje je moguće razviti za više od 20 različitih platformi. Uz to je dostupan u preko 190 zemalja na svijetu. Na mjesečnoj bazi preko 2,5 milijarde ljudi igra neku igru ili koristi neku aplikaciju razvijenu u Unity razvojnom okruženju. Statistika je pokazala da je preko 50 % igara za mobitele, PC-jeve i konzole napravljeno upravo u Unity razvojnom okruženju, a čak 71 % od 1000 najboljih mobilnih igara razvijeno je u Unityju.

Unity razvojno okruženje implementira mnogo funkcionalnosti koji se mogu nadograđivati, konfigurirati i koristiti. Neke su od njih sustav za fiziku (engl. *physics engine*), sustav za prikaz (engl. *rendering engine*), sustav za zvuk (engl. *sound system*), navigaciju (engl. *navigation and pathfinding*), animaciju (engl. *animation*), korisničko sučelje (engl. *user interface*), skriptiranje (engl. *scripting*) i mnogi drugi. Za skriptiranje u Unityju danas je podržan C# programski jezik, točnije C# 8.0 od verzije Unityja 2020.3. U prošlosti su bili podržani i drugi jezici kao što je JavaScript, ali zbog malog broja ljudi koji su se koristili njima, izbačeni su kao podrška. Nažalost, Unity danas nema svoj sustav za mrežu, ali je isti trenutno u razvoju. Mnoga vanjska mrežna rješenja, koja se mogu implementirati u Unityju, imaju neizvjesnu sudbinu jer se temelje na starom Unity mrežnom rješenju koji više nije podržan.

Unity također podržava i instalaciju novih dodataka kao pakete (engl. *packet manager*) preko kojih je moguće proširiti Unity s novim funkcionalnostima ili skinuti paket datoteka te ih koristiti u projektu. Za potrebu izrade igre preuzeti su mnogi besplatni paketi koji će biti opisani u poglavlju o implementaciji igre. Korištena je besplatna inačica Unityja koju je moguće koristiti za individualce ili manje timove sve dok ukupna zarada od igara izrađena u Unity razvojnom okruženju ne prelazi 100 000 $ godišnje. [1, 2]

2. 2. Git

Git je softver za verzioniranje i praćenje promjena nad datotekama. Razvio ga je Linus Torvalds 2005. godine. Prvotno je razvijen za potrebe razvoja Linux jezgre (engl. *kernel*). Danas je najkorišteniji alat za verzioniranje. Neke su od prednosti gita u odnosu na druge alate za verzioniranje:

- Svaki git repozitorij sadrži potpunu povijest projekta, što znači da se git ne oslanja na neki udaljen server za obavljanje većine zadataka, a to ubrzava rad s njim.
- Integritet – nije moguće promijeniti neke datoteke, a da git toga nije svjestan. Razlog tome je što git nad svim objektima radi *checksum* i poslije se referencira koristeći taj *checksum*.
- Git skoro uvijek dodaje podatke, što povratak na raniju verziju sustava čini sigurnom. Kad se uspostavi git, zajamčena je sigurnost da se može vratiti na staro. Sigurnost se povećava ako se pohranjuju repozitoriji na neki vanjski servis kao što je GitHub.

Git je moguće koristiti iz komandne linije ili putem grafičke aplikacije. To je alat otvorenog koda i besplatan je za preuzimanje, a neki se grafički klijenti za git naplaćuju. Treba napomenuti da postoje i besplatne inačice grafičkih klijenata za git. U radu je korištena git komandna linija, a povremeno je korišten ugrađeni git klijenta u Visual Studiju razvojnom okruženju, kao i gitk za pregled promjena nad datotekama. [3, 4]

3. GOAP

3. 1. Osnovne informacije GOAP algoritma

GOAP je derivacija STRIPS (*Stanford Reasearch Institue Problem Solver*) algoritma koji je prvotno prezentiran 1970-ih godina. Implementacija GOAP algoritma prvi je put izvedena u igri F.E.A.R. koja je izašla 2005 godine. F.E.A.R. je u recenzijama bio hvaljen kao igra s odličnim AI sustavom u kojem „AI čini stvari koje ne očekuješ“[5]. Implementirao ga je Jeff Orkins koji je tada bio u Monolith razvojnom studiju.

Algoritam samostalno pronalazi put do cilja ovisno o akcijama koje je moguće izvršiti i o trenutnom stanju agenta, tj. potrebno je pronaći put koji će trenutno stanje agenta dovesti u stanje opisano ciljem. Svaka akcija ima cijenu, set preduvjeta i set efekata koje ostvaruje nakon što su njezini preduvjeti zadovoljeni, a akcija odrađena. Može se reći da akcija mijenja stanje agenta iz stanja preduvjeta u stanje efekta za određenu cijenu. Agenti na sebi mogu imati niz ciljeva koje žele ostvariti. Planer ciljeva mora, u odnosu na neku heurističku funkciju vrednovanja ciljeva, odrediti koji je cilj agentu najvišeg prioriteta. Kad se odabere željeni cilj, potrebno je utvrditi može li se, i ako da, s kojim određenim nizom akcija doći do cilja.

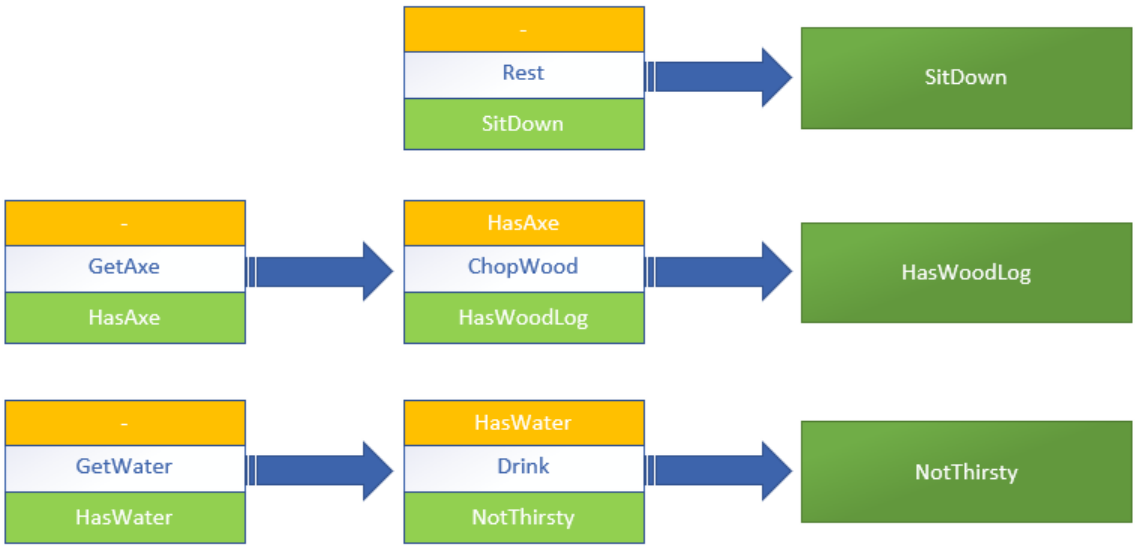
Za pretraživanje se koristi A* ili Dijkstrin algoritam. Postoje dva načina pretraživanja: pretraživanje unaprijed (od akcija čije preduvjete agent već zadovoljava) i pretraživanje unatrag (od cilja pa dok se svi preduvjeti akcija cilja ne zadovolje). Problem kod pretraživanja unaprijed jest što se razvijaju sve akcije za koje se imaju zadovoljeni preduvjeti. U slučaju velikog broja ciljeva i akcija na agentu, velika će količina toga biti redundantna jer velik broj akcija neće nikad voditi prema odabranom cilju. To se donekle može poboljšati korištenjem A* algoritma i pametnom heuristikom, no takvu je heuristiku vrlo teško implementirati da značajno pridonese smanjenu vremena izvršavanja u takvim situacijama. Također, uvođenjem heuristike moguće je da put do cilja neće uvijek biti optimalan što u nekim situacijama ne mora nužno biti loše. Bolja je rješenje graditi niz akcija od cilja prema potencijalnoj akciji čije preduvjete zadovoljavamo. Tako sve akcije koje sigurno ne zadovoljavaju cilj, neće biti uvrštene u pretraživanje.

3. 1. 1. Smjer pretraživanja GOAP algoritmom

Slika 1 prikazuje jednostavan primjer dolaska do željenog cilja. Narančasta su polja preduvjeti akcija. Vidi se da prva akcija `GetAxe` nema preduvjeta, stoga je odmah ispunjena, dok druga akcija `ChopWood` ima preduvjet `HasAxe`. Preduvjet te akcije ujedno je i efekt akcije `GetAxe`. To znači da se akcija `GetAxe` mora izvršiti prije akcije `ChopWood`. Konačni cilj vidljiv je s desne strane `HasWoodLog`, što je ujedno i efekt `ChopWood` akcije. Jednom kad agent izvrši donje dvije akcije u `GetAxe – ChopWood` redoslijedu, dođe se do željenog cilja.

Slika 1: Prikaz dolaska do cilja uz dvije akcije

Kod gornjeg primjera nije bilo moguće uvidjeti razliku između pretraživanja unaprijed i unatrag. Na sljedećoj slici (slika 2) može se jasno vidjeti razlika između dvaju načina pretraživanja.



Slika 2: Prikaz tri cilja s nizom akcija koji ih ostvaruju

U situaciji gdje je trenutni prioritetni cilj `HasWoodLog` sa slike, ako algoritam krene pretraživati unaprijed, u prvoj iteraciji algoritam izdvaja tri akcije koje je u stanju izvršiti.

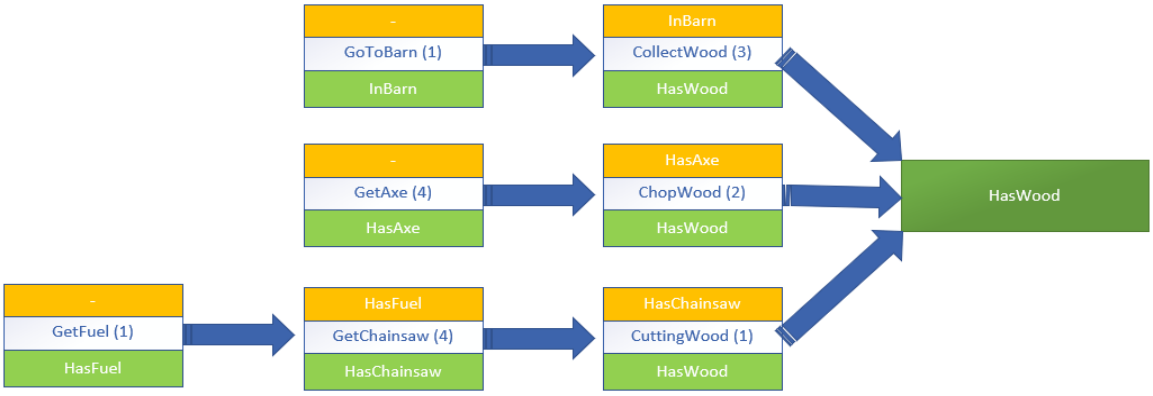
Te su akcije `Rest`, `GetAxe` i `GetWater`. Za svaku od tih akcija algoritam pokušava pronaći akciju koju je u stanju izvršiti nakon što se izvrši odabrana akcija. Tako, npr. kad se akcija `GetWater` izvrši, moguće je izvršiti i `Drink` akciju jer je izvršavanjem akcije `GetWater` ostvaren preduvjet `HasWater` akcije `Drink`. U drugoj iteraciji pretraživanjem unaprijed, algoritam uviđa da poslije izvršavanja akcije `Rest` nijednu drugu akciju nije moguće izvršiti, a da u isto vrijeme bude najjeftiniji put do cilja `HasWoodLog`. Izvršavanje akcije `Rest` ne dovodi do željenog stanja za ostvarivanje trenutnog cilja. Isti zaključak događa se i s akcijskim nizom `GetWater` i `Drink`. Jedini niz akcija koji dovodi optimalno do cilja jest akcijski niz `GetAxe` i `ChopWood`.

Pretragom unatrag algoritam bi kod prve iteracije zaključio da nijedna akcija osim `ChoopWood` ne zadovoljava efekt trenutnog cilja. Time algoritam već u prvoj iteraciji zaključuje da je to jedina akcija koja ima šanse ostvariti zadani cilj. U drugoj iteraciji algoritam pokušava pronaći akciju koja zadovoljava preduvjet akcije `ChoopWood`. Jedina je takva akcija `GetAxe`, a kako ona nema nikakav preduvjet, to bi bio kompletan niz akcija koje se trebaju izvršiti za postizanje cilja `HasWoodLog`.

Nije teško zamisliti složeniji primjer s desecima ciljeva i stotinama akcija koje agent može izvršiti, a da se uvidi da je pretraživanje unatrag bolje. To se posebno odnosi na situacije u kojima agent nije trenutno u stanju izvršiti preferirani cilj. S pretraživanjem unaprijed algoritam razvija sve moguće nizove akcija kako bi zaključio da nijedan ne može zadovoljiti cilj. Pretraživanjem unatrag, ako se ne može naći akcija koja zadovoljava preduvjet trenutne akcije, jasno je odmah da zadani cilj nije ostvariv.

3. 1. 2. Planiranje GOAP algoritmom

Kod GOAP algoritma svaka akcija ima svoju cijenu. Do sada su se sve akcije promatrale kao jednake, tj. cijene akcija bile su jednake. Slika 3 prikazuje primjer s različitim cijenama akcija do cilja `HasWood`.

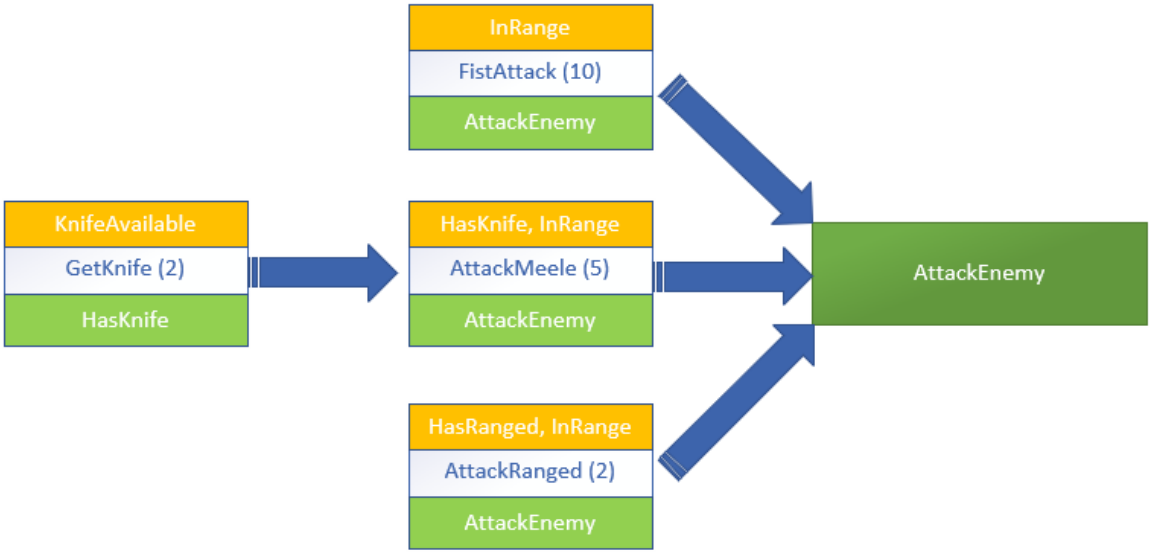


Slika 3: Prikaz mogućih puteva do cilja `HasWood`

Slika 3 iskoristit će se kako bi se prikazao način dolaska do cilja s algoritmom pretraživanja unatrag. Svaka akcija na slici označena je brojkom koja označava cijenu za izvršavanje te akcije. U prvoj iteraciji cilj mogu zadovoljiti tri akcije: `CuttingWood`, `ChopWood` i `CollectWood`. U sljedećoj iteraciji prvo se pokušava nastaviti najjeftiniji niz akcija. U slučaju prethodno navedenih akcija to bi bila `CuttingWood` akcija. Jedina akcija koja zadovoljava njezin preduvjet jest `GetChainsaw` akcija. Cijena akcije `CuttingWood` jest 1, a cijena akcije `GetChainsaw` jest 4. Ukupna cijena tog niza akcija u drugoj iteraciji iznosi 5. Algoritam gleda sljedeći najjeftiniji niz akcija, a to je niz akcija s jedinom akcijom `ChoopWood` ukupne cijene 2. Preduvjetu te akcije odgovara samo akcija `GetAxe` koja nema preduvjete, što znači da je cilj postignut. Međutim, prije potvrde postignutog cilja algoritam ide na sljedeću iteraciju gdje uzima sljedeći najjeftiniji niz akcija, u navedenom bi slučaju to bio `CollectWood` ukupne cijene 3. Akcija koja zadovoljava preduvjete te akcije jest `GoToBarn`. Kako ona nema preduvjete, može se reći i da je drugi put do cilja pronađen. Ukupna cijena ovog puta je 4, što je jeftinije nego prethodno pronađen put. Algoritam i dalje ne potvrđuje taj cilj već uzima sljedeći najjeftiniji niz akcija, a to je taj isti niz akcija `GoToBarn`, `CollectWood` ukupne cijene 4. Budući da su svi uvjeti za ovaj niz akcija ispunjeni, algoritam odabire ovaj niz akcija kao najjeftiniji niz za doseći zadani cilj sa slike.

Cilj je GOAP planera planiranje optimalnog niza akcija koje je potrebno izvršiti kako bi se doseglo željeno stanje. Ovisno o tipu igre, ponekad se ne traži optimalan put do zadanog cilja. Zbog toga je sasvim ispravno proširiti algoritam da ne vraća uvijek optimalan put, već možda sve moguće puteve na kojima korisnik ima mogućnost po vlastitom kriteriju odabirati

put. Dinamičnost odabira puta do cilja može se postići i na druge načine što će biti opisano nešto niže.



Slika 4: Prikaz mogućih puteva do cilja `AttackEnemy`

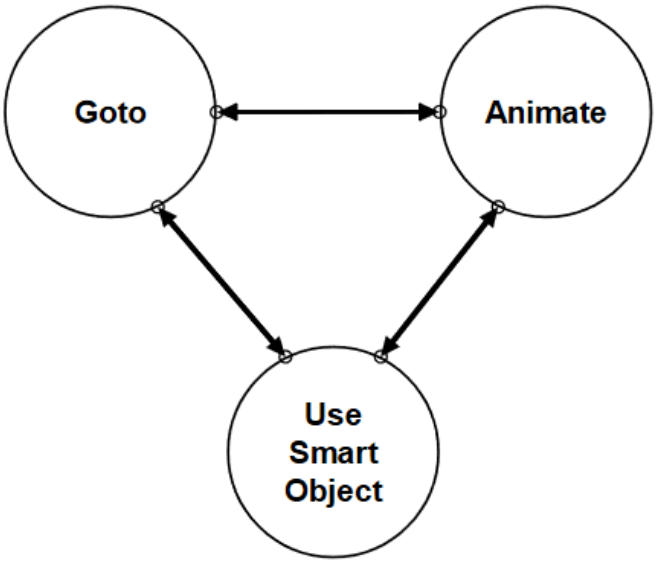
Osim navedenog, GOAP se sastoji i od trenutnih stanja agenta i/ili svijeta. Ovisno o implementaciji algoritma, svaki agent može imati neovisan i promjenjiv niz stanja u kojem se nalazi. S primjera sa slike 4 akcija `AttackRanged` zadovoljava cilj, ali njezini preduvjeti `HasRanged` i `InRange` ne mogu biti zadovoljeni nijednom akcijom sa slike. Akcija `AttackMeele` također ne može zadovoljiti preduvjet `InRange`, ali može zadovoljiti preduvjet `HasKnife` akcijom GetKnife. Konačan cilj i dalje nije dostižan jer preduvjet `InRange i KnifeAvailable` nije zadovoljen. Kod akcije `FistAttack` ima se preduvjet `InRange` koji nije zadovoljen skupom akcija sa slike. Međutim, uz drugačija stanja agenta može se postići da se agent dinamički prilagođava situaciji.

U situaciji gdje se stanja agenta postavljaju na sljedeći način:

- `InRanged` – kad je agent u dometu neprijatelja
- `HasRanged` – kad agent posjeduje oružje na daljinu
- `KnifeAvailable` – kad agent posjeduje saznanje gdje se nalazi nož

može se imati sustav u kojem će agent, ovisno o tome što posjeduje, odabrati na koji način napasti protivnika, a cijenom akcija mogu se uvjetovati kojem scenariju dati prednost ako je moguće izvršiti više scenarija.

GOAP sustav u sebi može sadržavati, i najčešće sadrži, jednostavan FSM. FSM služi kako bi raspodijelio rukovanje akcije na više razumljivih cjelina. Na slici 5 vidi se kako je FSM implementiran za igru F.E.A.R. FSM se sastojao od tri stanja: `Goto`, `Animate` i `UseSmartObject` s tim da je stanje `UseSmartObject` samo nadograđena implementacija stanja `Animate` od razvojnog tima Monolith. Svrha stanja `UseSmartObject` se, umjesto da se direktno kaže koja animacija će se pokrenuti, prosljeđivala kroz nekakve pametne objekte. Stoga se može smatrati da se FSM igre F.E.A.R. funkcionalno sastojao od samo dva stanja: `Goto` i `Animate`.[6]



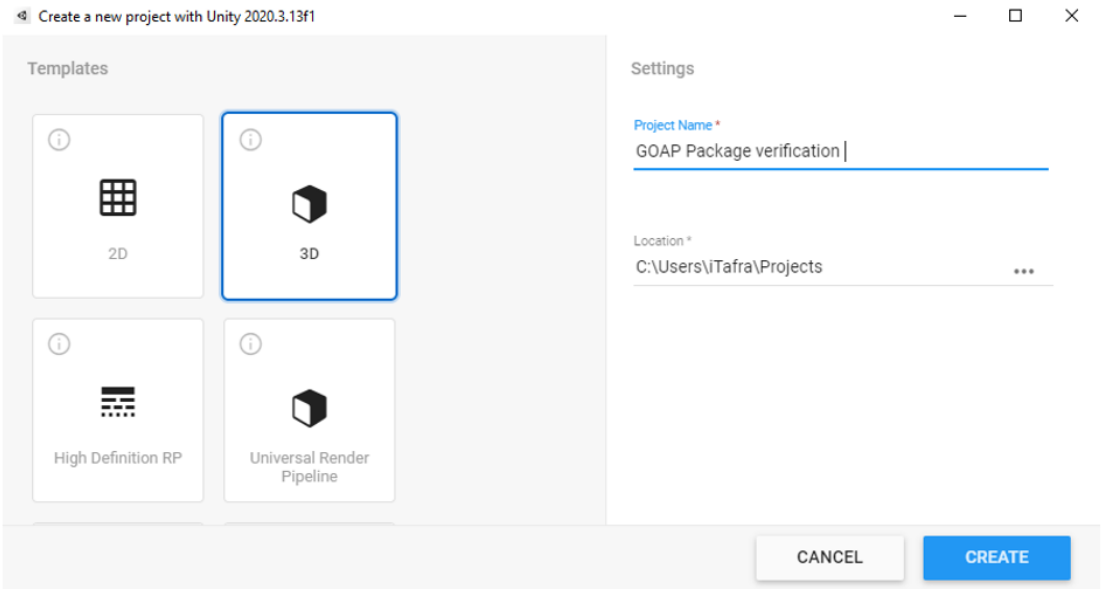
Slika 5: Implementacija GOAP FSM-a za igru F.E.A.R[6]

3. 2. Korištenje

3. 2. 1. Uvoz u projekt

GOAP algoritam implementiran je kao Unity *package* te se kao takav može povući i uključiti izravno u bilo koji Unity projekt. Unity *package* je kontejner koji može sadržavati razne dodatke koji se mogu koristiti u Unity projektu ili mogu biti proširenje za Unity ili njegov podsustav.[7] O implementaciji je više napisano u sljedećem potpoglavlju. U ovom je potpoglavlju prikazano na koji način korisnik može koristiti implementirani GOAP algoritam.

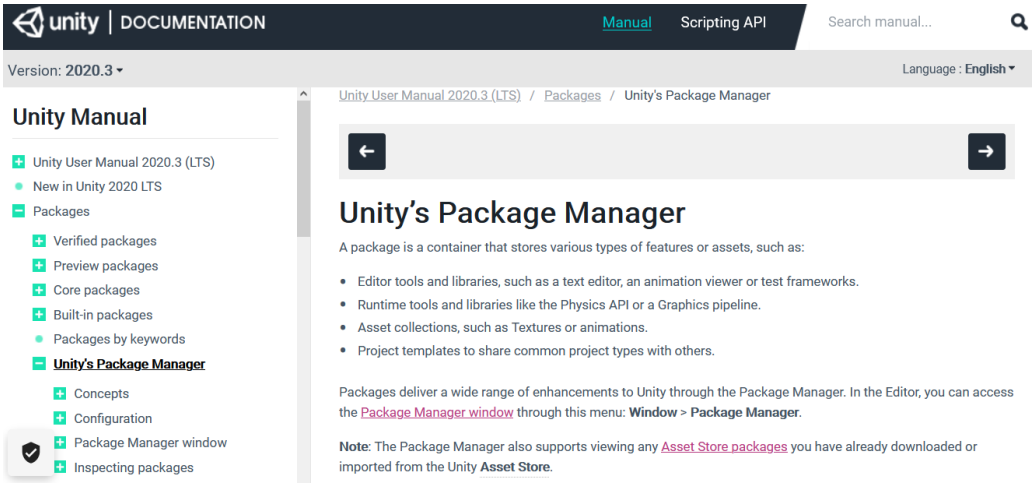
Da bi se mogao povući GOAP paket, prvo je potrebno kreirati ili otvoriti postojeći projekt u Unityju. Slika 6 prikazuje primjer kreiranja novog projekta s 3D predloškom u Unityju.



Slika 6: Kreiranje novog projekta u Unityju

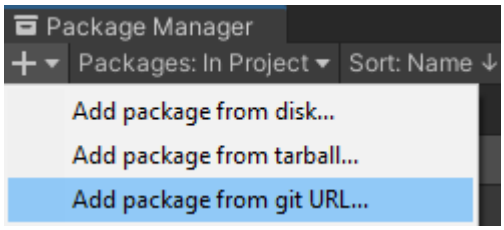
Nakon kreiranja projekta potrebno je potražiti *Package* manager u Unityju. Budući da se s novim verzijama Unityja takve stvari učestalo mijenjaju, najlakši način za pronaći kako do njega doći jest pregledom Unity službene dokumentacije za korištenu verziju Unityja.

Na slici 7 vidi se trenutni izgled dokumentacije za Unity 2020.3 koja prikazuje na koji način doći do *package managera* u Unityju. Dokumentacija sadrži i tražilicu s kojom se lako može pronaći proizvoljni pojam vezan za Unity i njegove pakete.



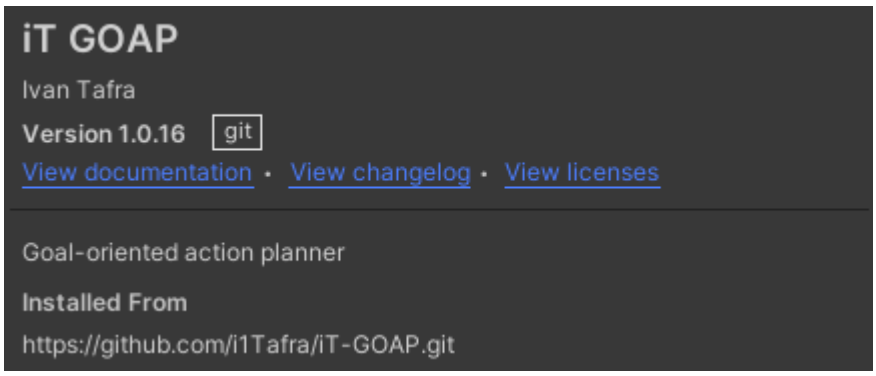
Slika 7: Unity dokumentacija za *package manager*

Zatim je potrebno odabrati opciju prikazanu na slici 8. U odabranom okviru potrebno je zalijepiti poveznicu na git projekt koji se želi uvesti u Unity. Nije moguće uvesti bilo kakav git projekt na ovaj način. Projekt treba biti oblikovan, konfiguriran i raspoređen na određen način kako bi se prepoznao kao Unity paket. Više o tome u implementacijskom opisu algoritma.



Slika 8: *Package manager* opcija za dodavanje Unity paketa izravno s gita

Poveznica na GOAP Unity paket je https://github.com/i1Tafra/iT-GOAP.git. Kad je sve odrađeno kako je opisano, prikazat će se zaslon kao na slici 9. Slika prikazuje trenutnu verziju paketa koji je uvezen (1.0.16). Za uvoz novije verzije GOAP paketa dovoljno je ponoviti prethodne korake koji su učinjeni. Nažalost, na ovaj način dodavanja paketa preko git repozitorija nije moguće automatski pratiti je li neka novija verzija paketa objavljena na git repozitoriju.



Slika 9: Uspješno uvezen GOAP paket u Unity

Dodavanjem paketa u direktoriju projekta pod poddirektorijem *Packages* sad bi se trebao nalaziti i poddirektorij „iT GOAP“ sa svim pripadnim datotekama paketa. Najmanje što je potrebno za korištenje jesu definiranje cilja, definiranje akcija koje dovode do željenog cilja i definiranje uvjeta za ciljeve i akcije. Potrebno je i kreiranje agenta i dodjeljivanje svih kreiranih elemenata na jedan Unity objekt.

3. 2. 2. Primjer korištenja

Za demonstraciju je kreiran primjer sa slike 3. Prvo su kreirani potrebni uvjeti kako bi se isti mogli dodjeljivati akcijama i cilju. Ispis 1 prikazuje potpunu listu uvjeta koji se mogu dalje koristiti kao preduvjeti i efekti akcija i ciljeva. Radi jednostavnosti, svi su uvjeti kreirani u jednoj klasi kako je prikazano na ispisu. U nekoj kompleksnoj igri koja sadrži mnogo uvjeta, akcija, različitih agenata i slično, mogu se, uz pomoć različitih struktura klasa, jasno razdvojiti i definirati uvjeti ovisno o željenoj strukturi projekta, što značajno pridonosi čitljivosti i jasnoći koda.

```
public static class Conditions
{
    public static Condition HasAxe = new Condition("HasAxe");
    public static Condition HasWood = new Condition("HasWood");
    public static Condition InBarn = new Condition("InBarn");
    public static Condition HasFuel = new Condition("HasFuel");
    public static Condition HasChainsaw = new Condition("HasChainsaw");
}
```

Ispis 1: Definirani potrebni uvjeti sa slike 3

Ispis 2 prikazuje javne funkcionalnosti koje implementira klasa `Condition`. Za konstruirati `Condition` potrebno je dati ime uvjeta jer standardni konstruktor za ovu klasu ne postoji. Primjer konstruiranja uvjeta vidljiv je na ispisu 2. Ostale mogućnosti koje pruža ova klasa jesu čitanje numeričkog identifikatora koji je dodijeljen ovom uvjetu ili proslijeđeno ime koje je dano kod kreiranja `Condition` objekta. Pregažene (engl. *override*) su i metode `Equals`, `GetHashCode` i `ToString`.

```
public struct Condition
{
    public Condition(string name);

    public readonly int Id { get; }
    public readonly string Name { get; }

    public override bool Equals(object conObject);
    public override int GetHashCode();
    public override string ToString();
}
```

Ispis 2: Svojstva, metode i konstruktori definirane za `Condition` klasu

Sljedeći je korak definiranje ciljeva. Ispis 3 prikazuje svojstva, metode i konstruktore klase `Goal`, kao i sučelja (engl. *interface*) i klase koje klasa `Goal` nasljeđuje i implementira. O razlozima implementacija pojedinih sučelja i klasa bit će riječ u poglavlju o implementaciji algoritma. Klasa pruža mogućnost postavljanja vrijednosti bazne želje (`baseUrge`) za ostvarivanje nekog cilja prilikom nasljeđivanja, kao i mogućnost dinamičkog mijenjanja množitelja bazne želje (`UrgeMultiplier`). Prilikom nasljeđivanja potrebno je nabrojati efekte koje cilj postiže. Podaci koji se poslije konstruiranja mogu samo čitati, ali ne i mijenjati jesu ime cilja, `HashSet` efekata koje cilj postiže i trenutna snaga želje (`Urge`) koja je produkt bazne želje i množitelja bazne želje.

```
public class Goal : MonoBehaviour, IGoalInfo
{
    public Goal();

    public HashSet<Condition> Outcome { get; }
    public string Name { get; }
    public float Urge { get; }
    public float UrgeMultiplier { get; set; }
    protected List<Condition> Effects { get; set; }

    protected void SetBaseUrge(float baseUrge);
}
```

Ispis 3: Svojstva, metode i konstruktori definirane za `Goal` klasu

U slučaju primjera, za definiranje cilja dovoljno je navesti efekt koji cilj očekuje. Bazna želja i množitelj bazne želje nisu mijenjani jer će lista ciljeva koje agent može zadovoljiti sadržavati samo ovaj cilj. Prikaz implementacije cilja vidi se na ispisu 4.

```
public class HasWood : Goal
{
    Unity Message | 0 references
    private void Awake()
    {
        Effects.Add(Conditions.HasWood);
    }
}
```

Ispis 4: Prikaz definiranja cilja `HasWood` prikazanog na slici 3

U sljedećem koraku definiraju se akcije. Ukupno 7 akcija potrebno je definirati slijedeći primjer sa slike 3. Ispis 5 prikazuje sve mogućnosti `Action` klase.

```
public abstract class Action : MonoBehaviour, IActionlInfo
{
    protected Action();

    public bool Completed { get; protected set; }
    public string Name { get; }
    public HashSet<Condition> Preconditions { get; }
    public HashSet<Condition> Effects { get; }
    public float Cost { get; protected set; }
    public bool Abortable { get; protected set; }
    protected ActionState State { get; set; }

    public virtual void Abort();
    public void Execute(ActionArgs args);
    public void ResetState();
    public override string ToString();
    public void Transit();
    protected virtual void Perform(ActionArgs args);
    protected virtual void PostPerform(ActionArgs args);
    protected virtual void PrePerform(ActionArgs args);

    protected enum ActionState
    {
        PrePerform = 0,
        Perform = 1,
        PostPerform = 2
    }
}
```

Ispis 5: Svojstva, metode i konstruktori definirane za `Action` klasu

`Action` klasa definirana je kao apstraktna. Razlog tome je što akcija bez implementacije ponašanja nema smisla. U odnosu na prethodne klase, ova je najkompleksnija jer sadrži samu logiku neke akcije koju agent može izvesti. Jednostavan oblik implementacije akcije prikazan je na ispisu 6 koja prikazuje implementaciju akcije `CollectWood`.

```
public class CollectWood : Action
{
    private static float base_price = 3f;

    Unity Message | 0 references
    private void Start()
    {
        Preconditions.Add(Conditions.InBarn);
        Effects.Add(Conditions.HasWood);
        Cost = base_price;
    }
    0 references
    protected override void PrePerform(ActionArgs args)
    {
        Debug.Log($"{this}");
        base.PrePerform(args);
    }
    0 references
    protected override void Perform(ActionArgs args)
    {
        Debug.Log($"{this}");
        base.Perform(args);
    }
    0 references
    protected override void PostPerform(ActionArgs args)
    {
        Debug.Log($"{this}");
        base.PostPerform(args);
    }
}
```

Ispis 6: Implementirana akcija `CollectWood`

Sa ispisa je vidljivo da je kao preduvjet ovoj akciji postavljen `InBarn` uvjet, a kao efekt naveden je `HasWood`. Cijena izvršavanja ove akcije stavljena je kao `3f`, kako je i prikazano na slici 3. Tri metode koje su implementirane jesu: `PrePerform`, `Perform` i `PostPerform`. Prilikom izvršavanja akcije pokreće se interni FSM kojem je početno stanje `PrePerform`. Bazna implementacija je takva da će, u ukupno tri tranzicije, akcija biti izvršena, tj. bazna implementacija ide iz stanja `PrePerform` u stanje `Perform`, te zatim iz stanja `Perform` u stanje `PostPerform`. Nakon toga se interno postavlja svojstvo `Completed` u *`true`* i akcija je potpuno izvršena te se smatra da je agent postigao efekt zadane akcije.

Korisnik nužno ne mora implementirati sva tri stanja jer ista služe samo za preraspodjelu odgovornosti. Primjer bi bio da se u `PrePerform` stanju provjeri jesu li svi preduvjeti zadovoljeni. Nakon toga korištenjem `Transit` metode ili pozivom bazne metode mijenja se stanje u stanje `Perform`. U tom se stanju akcija može izvršiti, a u stanju

`PostPerform` može se očistiti kod od podataka koji više ne trebaju. Pozivom `Transit` metode u `PostPerform` stanju ili bazne implementacije `PostPerform` metode označava se akcija uspješno izvršenom. Implementacija je u ovom slučaju zapisivanje na konzolu za svako stanje. Tranzicija se vršila pozivom bazne klase.

Ako nisu potrebna tri stanja u FSM, nego je dovoljno manje stanja, moguće je završiti akciju ranije postavljanjem `Completed` svojstva u *`true`* u nekom od ranijih stanja. Mehanizam u pozadini koristi to svojstvo kako bi zaključio je li akcija u stanju izvršavanja ili je završila. Također, ako korisnik želi, može i manipulirati iz kojeg se stanja prelazi u koje, direktno postavljajući svojstvo `State`.

Metode `PrePerform`, `PostPerform` i `Perform` primaju kao argument `ActionArgs`, izveden na sličan način kao i C# `EventArgs` koji se koristi za C# događaje (engl. *events*). Isti se može proširivati kako bi akciji mogli proslijediti potrebne podatke. Primjer proširivanja bio bi sustav za pokretanje agenta kako bi akcija mogla narediti sustavu da dođe do određene točke.

Metoda `ResetState` pokreće se svaki put kad agent zaključi novi put do nekog cilja. Tad se za cijeli set akcija, koje je potrebno izvršiti za dostizanje cilja, pozove akcija `ResetState`. Sama akcija vraća postavke akcije u prvobitno stanje.

Metoda `Execute` se opetovano poziva od strane agenta jednom kad je plan isplaniran, sve dok akcija nije prekinuta ili izvršena.

Mogućnost koju pruža ova klasa jest prekid neke akcije prije kraja njezina izvršavanja. Korisnik bira koje akcije ili u kojim je stanjima pojedine akcije moguće prekinuti istu. To se postiže postavljanjem svojstva `Abortable` u *`true`*. Prekidom akcije upravlja agent, a o tome će više riječi biti poslije. U slučaju da agent zaključi da je akciju moguće prekinuti i da ju je potrebno prekinuti, prije prekida će pozvati `Abort` metodu koja pruža mogućnost da se naprave potrebne prilagodbe za prekid (npr. prirodno prebaciti animaciju u neku drugu, odložiti predmet s kojim se nešto radilo i slično). Ako korisnik pokuša pozvati `Abort` metodu diskretno, bez da je prekid podržan postavljanjem svojstva `Abortable` u *`true`*, ista će izbaciti iznimku (engl. *throw exception*). Implementacija agenta prikazana je na ispisu 7.

```
public class NPC : Agent
{
    Unity Message | 0 references
    private void OnEnable() => Init();
    Unity Message | 0 references
    private void OnDisable() => Clear();

    Unity Message | 0 references
    protected void Awake() => Debuging = true;

    Unity Message | 0 references
    protected void FixedUpdate() => Execute(ActionArgs.Empty);
}
```

Ispis 7:. Implementacija agenta

Potrebno je odabrati na koje će se Unity poruke (engl. *Unity message*) pojedina metoda iz `Agent` klase pozvati. Ukupan broj Unity poruka prevelik je da bi bio objašnjen ili prikazan ovdje. Jasna slika s redoslijedom izvođena pojedine Unity poruke može se pronaći na službenoj Unity dokumentaciji.

`Init` je potrebno pozvati samo jednom i zbog toga je pozvan na Unity poruku `OnEnable` s obzirom na to da se ista poziva samo kad je objekt uključen u scenu projekta. Ako se objekt isključi, poziva se Unity poruka `OnDisable` te se taj poziv povezao s metodom `Clear`. `Awake` je prva metoda koja se poziva u Unity sustavu poruka. `Awake` se poziva samo jednom, ona je iskorištena za uključivanje `Debugging` opcije kako bi se na konzoli vidjeli dodatni ispisi iz GOAP paketa. U ovom se primjeru akcije samo ispisuju na konzolu pa nije bitno u kojem se `Updateu` poziva `Execute` metoda. Međutim, prilikom stvarne implementacije važno je znati koji `Update` koristiti jer Unity ima niz `Update` poruka. Tako npr. ako se želi nešto obaviti s fizičkim podsustavom u Unityju, poželjno je to raditi unutar `FixedUpdate` poruke. Ako se ne slijede Unity preporuke, mogući su neočekivani događaji i greške.

Mogućnosti agenta nešto su veće nego prijašnje klase. Iste su prikazane na ispisu 8.

```
public class Agent : MonoBehaviour
{
    public Agent();

    public IEnumerable<Action> ActionQueue { get; }
    public IEnumerable<Action> Actions { get; }
    public bool HasPlan { get; }
    public IGoalInfo ActiveGoal { get; }
    public IActionlInfo ActiveAction { get; }
    public IEnumerable<IGoalInfo> Goals { get; }
    public IEnumerable<Condition> Conditions { get; }
    protected bool Debuging { get; set; }

    public bool AddCondition(Condition condition);
    public void AddGoal(Goal goal);
    public bool RemoveCondition(Condition condition);
    public void RemoveGoal(Goal goal);
    public void SetGoalUrgeMultiplier(IGoalInfo goalInfo, float urgeMultiplier);
    protected void Clear();
    protected void Execute(ActionArgs args);
    protected virtual Queue<Action> GetActionChain(List<Action> actions, HashSet<Condition> conditions, Goal goal);
    protected void Init();
    protected void Init(float abortThreshold, float abortTimer);

    public struct AgentState
    {
        public AgentState(Agent agent);

        public bool IsEqual(Agent agent);
        public void Set(Agent agent);
    }
}
```

Ispis 8: Svojstva, metode i konstruktori definirane za `Agent` klasu

Iz ispisa je vidljivo da je potrebno povezati sve zaštićene metode koje nisu virtualne s nekom `Unity` porukom. To je i prikazano na ispisu 7. Klasa pruža dva različita načina korištenja kroz dvije različite `Init` metode. Jedan `Init` je bez argumenata, a drugi s dva argumenta: `abortThreashold` i `abortTimer`.

Argument `abortThreashold` govori na koliku će razliku između dvaju ciljeva `abort` sustav reagirati. Ranije je spomenuto da svaki cilj ima trenutnu želju (`Urge`) da se izvrši. Ako prilikom izvršavanja nekog cilja želja za nekim drugim ciljem naraste toliko da bude veća od želje cilja koji se trenutno izvršava uvećan za definirani `abortThreashold`, agent će pokušati prekinuti izvršavanje trenutnog cilja. Agent će uspjeti jedino ako je to moguće za trenutnu akciju, tj. ako je akciji `Abortable` svojstvo postavljen u *`true`*. Prije pokušaja prekida agent će provjeriti postoji li putanja za izvršavanje novog cilja. Samo u slučaju da postoji, pokušat će prekinuti izvršavanje trenutnog cilja agenta.

Drugi argument `abortTimer` predstavlja vrijednost u sekundama svako koliko će agent provjeravati je li neki cilj postao prioritetniji za vrijednost trenutnog cilja uvećanog za `abortThreashold`.

Zaštićena virtualna metoda `GetActionChain` po baznoj implementaciji uvijek vraća najjeftiniju putanju do cilja. Ako se želi neka druga izvedba, može se odabrati iz

`Planner` klase ili se može napraviti vlastita implementacija. Sve što je potrebno jest da metoda vraća niz akcija koje je potrebno izvršiti u zadanom redoslijedu kako bi se postigao željeni cilj.

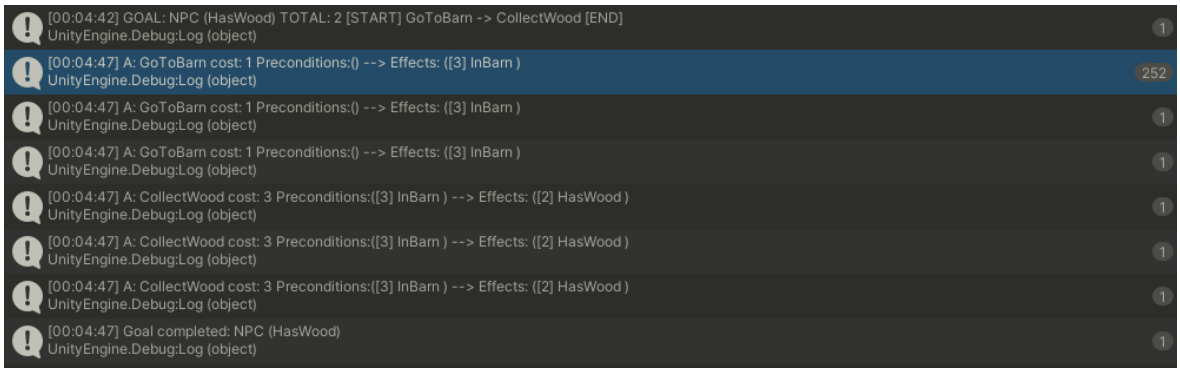
Veliki broj svojstava govori u kojem je trenutnom stanju agent. Tako svojstvo `ActionQueue` vraća niz akcija koje je još potrebno izvršiti poslije akcije koja se trenutno izvršava kako bi se dosegao cilj. Svojstvo `Actions` je nepromjenjiva lista svih mogućih akcija koje agent može izvršiti. `HasPlan` govori ima li trenutni agent zadani cilj ili ne. `ActiveGoal` i `ActiveAction` vraćaju koja se trenutno akcija izvršava za koji cilj. Oba svojstva vraćaju informacije o cilju ili akciji, ali ne i samu akciju ili cilj kako se iste ne bi nedozvoljeno mijenjale izvana. Više o tome bit će u dijelu o implementaciji. Zadnje informacije nepromjenjiv su niz mogućih ciljeva koje agent želi postići (`Goals`) i nepromjenjiv niz trenutnog stanja agenta (`Conditions`). `Debugging` svojstvo je promjenjivo. Po baznoj je implementaciji isključen, a ako je potrebno analizirati GOAP paket zbog problema ili neočekivanog ponašanja, može ga se uključiti kako bi se razumjelo što se događa. Moguće je dodavanje stanja, ciljeva i akcija na agenta kao i micanje istih. Agent pruža mogućnost dinamičkog mijenjanja množitelja bazne želje. Time se može upravljati da agent ima veću ili manju želju zadovoljiti neki cilj.

Nakon implementacije klasa potrebno je sve zajedno povezati u jedan Unity objekt. Slika 10 prikazuje povezane klase u Unity sceni u *edit* i *play* načinu rada (engl. *mode*).

Slika 10: Unity objekt s pripadnim klasama u *edit* načinu rada lijevo i *play* načinu rada desno

Cijenu pojedine akcije moguće je podešavati direktno iz editora. Treba napomenuti da, ako je cijena definirana i ako su u kodu naslijeđene klase u bilo kojoj Unity poruci, ta će vrijednost pregaziti onu postavljenu iz editora. To ponašanje prikazano je s desne strane slike 10. Klasa koja implementira agenta, u ovom slučaju `NPC` automatski će prilikom pokretanja povući sve akcije i ciljeve koje su dodane na Unity objekt.

Na slici 11 prikazana je konzola nakon što se pokrenula igra s agentom i implementiranim ciljem i akcijama. U prvoj liniji vidi se zapis koji je nastao od postavljanja parametra `Debugging` u `true` kod implementacije `NPC` klase. Vidi se da je agent odabrao cilj `HasWood` i da ukupno ima dvije akcije kako bi izvršio cilj. Te su dvije akcije `GoToBarn` i `CollectWood`. Nakon toga vidljiva su tri odvojena poziva `GoToBarn` gdje je ispisano kolika je cijena te akcije i koji su njezini preduvjeti i efekti. Tri se puta ispisuje jer se prolazi kroz tri stanja u internoj FSM. Nakon toga se isto ispisuje za sljedeću akciju `CollectWood`. Na kraju zapisa vidljivo je da se zadani cilj izvršio.

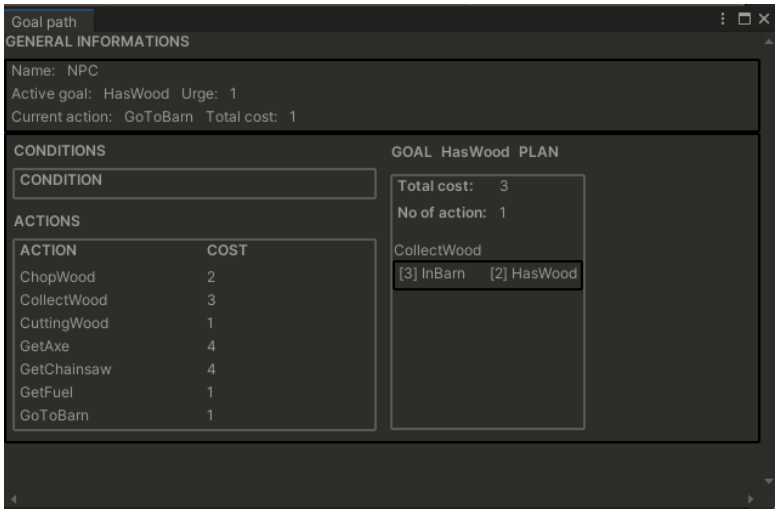


Slika 11: Zapis planiranja i izvršavanja cilja sa slike 3

`PrePerform` stanje akcije `GoToBarn` izvršilo se 252 puta jer je u implementaciji jedne od metoda FSM-a dodan brojač od 5 sekundi prije prelaska na sljedeće stanje. Tako je bilo moguće uhvatiti sliku 12 koja prikazuje proširenje za Unity editor.

Prilikom pokretanja igre u Unity editoru može se otvoriti prozor pod putanjom *Window* –> GOAP –> *Goal path*. Nakon toga dovoljno je kliknuti na agenta na sceni koji koristi GOAP paket i prikazat će se stanje agenta u realnom vremenu.

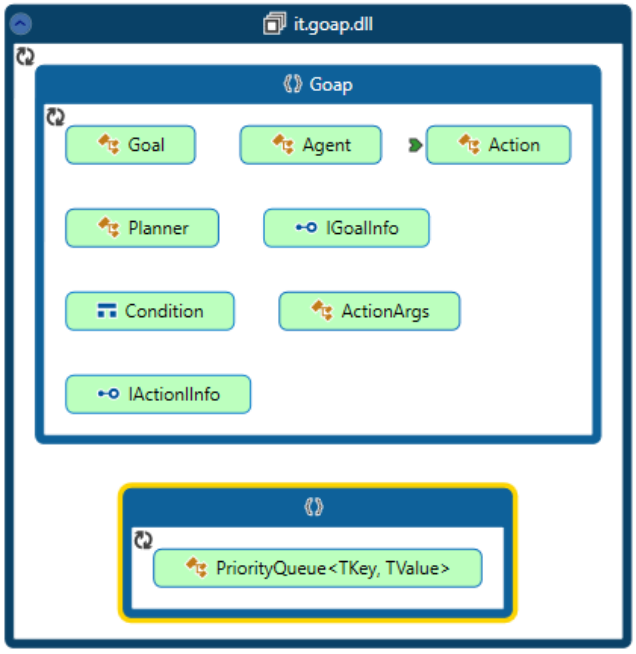
Na vrhu prozora nalaze se osnovne informacije o agentu – to su njegovo ime, trenutni cilj i njegova ukupna želja za ostvarivanje cilja. Ispod se nalazi lista stanja koja su trenutno primijenjena na agenta i listu svih akcija s pripadnim cijenama koje selektirani agent može izvršiti. S desne strane prozora na slici 12 prikazani su detalji trenutnog plana koji agent izvršava, ukupna cijena i broj akcija koje agent tek treba izvršiti, ne računajući trenutnu akciju koja se izvršava, i detalji o svakoj pojedinoj akciji. Detalji pojedine akcije prikazuju ime akcije ispod koje slijedi numerički identifikator i ime preduvjeta s lijeve strane i efekta s desne strane akcijskog prozora.



Slika 12: GOAP paket Unity editor proširenje

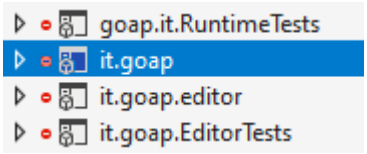
3. 3. Implementacija

U ovom poglavlju detaljno je opisano na koji se način GOAP algoritam implementirao kako bi se isti mogao koristiti kao Unity paket. Slika 13 prikazuje osnovne klase, strukture i sučelja GOAP implementacije.



Slika 13: Glavni dio GOAP implementacije, algoritam

Osim osnovne funkcionalnosti algoritma paket implementira još tri odvojena projekta ili dinamičke biblioteke prikazane na slici 14.



Slika 14: Potpuna implementacija koja čini GOAP Unity paket

Tablicom 1 opisana su zaduženja svake dinamičke biblioteke u GOAP paketu, a tablica 2 sadrži kratak opis zaduženja pojedinih klasa, struktura i sučelja definirana u `it.goap` dinamičkoj biblioteci u kojoj se nalazi glavni dio GOAP implementacije.

Tablica 1: Dinamičke biblioteke GOAP implementacije

IME	OPIS
`it.goap`	Nalazi se potpuna GOAP implementacija algoritma sa svim pripadnim klasama, strukturama i sučeljima.
`it.goap.editor`	Sadrži proširenje za Unity *editor*.
`it.goap.EditorTests`	Sadrži *unit* testove za implementirani GOAP algoritam.
`it.goap.RuntimeTests`	Sadrži *runtime* testove za implementirani GOAP algoritam.

Tablica 2: Opis klasa, struktura i sučelja implementacije GOAP algoritma

IME	TIP	OPIS
`Condition`	struktura	Zadužena je za definiranje stanja. Koristi se kod trenutnog stanja agenta, kao preduvjeti i efekti akcija i kao efekti ciljeva.
`IGoalInfo`	sučelje	Sučelje koje daje osnovne informacije o nekom cilju, implementira ga klasa `Goal`.
`Goal`	klasa	Opisuje cilj nizom efekata i trenutnom željom za izvršavanje cilja.
`IActionInfo`	sučelje	Sučelje koje daje osnovne informacije o nekoj akciji, implementira ga klasa `Action`.
`Action`	klasa	Definira koji su preduvjeti i efekti neke akcije, kolika je cijena izvršavanja te akcije i na koji se način akcija izvršava.
`ActionArgs`	klasa	Argumenti koje se prosljeđuju svim stanjima unutarnje FSM klase `Action`.
`PriorityQueue`	klasa	Pomoćna klasa koja implementira prioritetni red s mogućnošću rada s više elementa istog prioriteta.

`Planner`	klasa	Zadužen da na osnovu trenutnog stanja, liste mogućih akcija i željenog cilja pronađe put do cilja.
`Agent`	klasa	Zadužen za interakciju među svim dijelovima sustava. Povezuje cijeli sustav u jednu cjelinu. Zadaća mu je konstantno izvršavati akcije na osnovu trenutnih ciljeva i mogućih akcija koje može izvršiti.

3. 3. 1. Uvjet

`Condition` je pomoćna struktura kojoj je svrha opisivanje stanja ciljeva i akcija, kao i opisivanje preduvjeta akcija. Jedan element ove klase opisuje jedno stanje. Prethodno je na ispisu 2 prikazano od kojih se elemenata struktura sastoji. Svrha je struktura da bude brza pri izvođenju, planiranju puta, a u isto vrijeme čitka krajnjem korisniku iste. Prilikom konstruiranja elementa strukture dodjeljuje joj se jedinstveni numerički identifikator kako je i prikazano na ispisu 9. Uz identifikator korisnik određuje ime, pa je istu lako čitati i shvatiti koje se stanje željelo opisati istom.

```
private static int next_id = 1;
1 reference | 0 changes | 0 authors, 0 changes
private static int NextId { get => next_id++; }

99+ references | 0 changes | 0 authors, 0 changes
public Condition(string name)
{
    Id = NextId;
    Name = name;
}
```

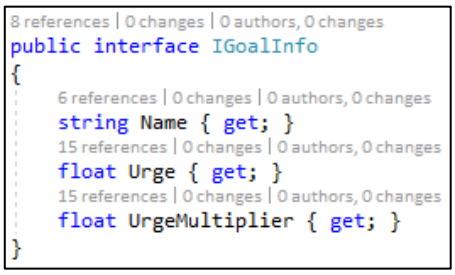
Ispis 9: Implementacija `Condition` konstruktora

3. 3. 2. Cilj

`Goal` je apstraktna klasa koja implementira osnovnu Unity klasu `MonoBehaviour`. `MonoBehaviour` je nužno implementirati ako se želi imati mogućnost stavljanja instance klase na Unity objekt. Uz to `Goal` implementira `IGoapInfo` sučelje koje pruža podatke o

imenu cilja, trenutnoj želji i trenutnom množitelju želje. Treba napomenuti da je trenutna želja produkt bazne želje i trenutnog množitelja želje pa se saznavanje vrijednosti bazne želje dobije jednostavnim dijeljenjem želje s množiteljem želje.

Implementirano sučelje može se vidjeti na ispisu 10. S ispisa je vidljivo da je sučelje samo za čitanje. Tako je moguće proslijediti informacije o cilju drugim objektima bez da mu se da mogućnost da isti mijenja ili nekako utječe na njega.



Ispis 10: `IGoalInfo` sučelje

Opisivanje efekta cilja može se postići samo kod nasljeđivanja. Na ispisu 11 vidi se na koji su način implementirani efekti cilja kako bi se smanjila mogućnost da istu netko slučajno modificira izvana. Korisnik kod nasljeđivanja puni `Effects` svojstvo klase `Goal`. Druge klase mogu samo pristupiti `HashSetu` stanja koja ima implementiranu lijenu inicijalizaciju, tj. `HashSet` se neće kreirati dok joj se ne pristupi prvi put. Neće se ni promijeniti ako se broj elemenata u listi `Effects` nije promijenio jer se smatra da je ista.

```
protected List<Condition> Effects { get; set; } = new List<Condition>();

private HashSet<Condition> outcome;

26 references | 0 changes | 0 authors, 0 changes
public HashSet<Condition> Outcome
{
    get
    {
        return (outcome?.Count != Effects.Count) ?
            outcome = new HashSet<Condition>(Effects) : outcome;
    }
}
```

Ispis 11: Implementacija efekta `Goal` klase

Bazna želja i množitelj bazne želje definirani su atributom `SerializeField` koji omogućuje da se ista vrijednost može mijenjati iz editora iako su privatne. Sa ispisa 12 vidljivo je da množitelj želje ima ograničeni opseg od 0 do 10.

```
[SerializeField]
private float baseUrge = 1f;

[SerializeField]
[Range(0, 10)]
private float urgeMultiplier = 1f;
```

Ispis 12: Varijable opisane atributom `SerializeField` mogu se mijenjati iz Unity editora

3. 3. 3. Akcija

Kao i `Goal`, klasa `Action` iz istih razloga implementira klasu `MonoBehaviour` i sučelje `IActionInfo`. `IActionInfo` nam daje podatke o imenu akcije i njezinoj cijeni izvršavanja. Implementacijski gledano, ova je klasa predložak za implementiranje ponašanja koje bi akcija trebala izvesti. Detaljnije informacije što sve klasa sadrži vide se na ispisu 5.

`Action` u sebi sadrži jednostavan FSM. Po baznoj implementaciji iz početnog stanja `PrePerform` prelazi se u `Perform` te potom u `PostPerfom`. Isto može biti modificirano ponovnom implementacija metoda `PrePerform`, `Perform` i `PostPerform`. Pokretanje FSM-a vrši se pozivom metode `Execute` koja prima `ActionArgs` kao parametar. Definicija istog vidljiva je na ispisu 13. Ako nisu potrebni nikakvi argumenti za obaviti neku akciju, može se koristiti `ActionArgs.Empty` svojstvo.

```
74 references | 0 changes | 0 authors, 0 changes
public class ActionArgs
{
    private static ActionArgs empty = new ActionArgs();
    40 references | 0 changes | 0 authors, 0 changes
    public static ActionArgs Empty { get => empty; }
}
```

Ispis 13: Definicija pomoćne klase za argumente FSM-a `Action` klase

3. 3. 4. PriorityQueue i Planner

`PriorityQueue` je pomoćna klasa koja se koristi prilikom planiranja putanje od trenutnog stanja do željenog stanja. Implementirana je generički, tako da se može upotrijebiti neovisno o tipu ključa i tipu vrijednosti koji se odabere. Bitna je stavka podrška za više elemenata s istim ključem, tj. istim prioritetom. U slučaju da dva elementa imaju isti prioritet, prvo će se vratiti onaj koji je prvi dodan u prioritetni red. Kod elemenata s istim

prioritetom prioritetni red radi na FIFO (*first in first out*) principu. Potpis klase vidljiv je na ispisu 14.

```
public class PriorityQueue<TKey, TValue> : IEnumerable<KeyValuePair<TKey, TValue>>,
    IEnumerable,
    IReadOnlyCollection<KeyValuePair<TKey, TValue>>,
    IReadOnlyDictionary<TKey, TValue>
```

Ispis 14: Potpis klase PriorityQueue

Klasa implementira nekoliko različitih sučelja kako bi se mogla koristiti kao i ostale generičke kontejner klase u C#-u. Klasa se koristi slično kao `Dictionary` klasa. Razlikuje ju interna izvedba, tako npr. `Dequeue` vraća `Tuple` prioriteta i vrijednosti. Razlog tome je što se, ako nije potreban, prioritet uvijek može ignorirati, a ponekad prioritet sadrži dodatnu vrijednost te je ovo bio najlakši, a i najbrži način da se isti dobije. Na ispisu 15 prikazan je način dohvaćanja najprioritetnijeg elementa iz `PriorityQueue` klase.

```
public (TKey key, TValue value) Dequeue()
{
    if (IsEmpty)
        return default;

    var firstElement = queue.First();
    var firstElementValue = firstElement.Value.Dequeue();

    if (firstElement.Value.Count == 0)
        queue.Remove(firstElement.Key);

    --count;
    return (firstElement.Key, firstElementValue);
}
```

Ispis 15: `PriorityQueue` `Dequeue` funkcionalnost

`Planner` je statička klasa kojoj je osnovna funkcija, na osnovu zadanih akcija, trenutnih efekata, cilja i maksimalne dopuštene dubine, izračunati najjeftiniji put ili sve puteve koje vode do određenog cilja. `Planner` implementira internu klasu `Node` prikazanu na ispisu 16.

```
77 references | 0 changes | 0 authors, 0 changes
static public class Planner
{
    43 references | 0 changes | 0 authors, 0 changes
    public class Node
    {
        7 references | 0 changes | 0 authors, 0 changes
        public Node Parent { get; set; }

        7 references | 0 changes | 0 authors, 0 changes
        public Action Action { get; set; }

        31 references | 0 changes | 0 authors, 0 changes
        public HashSet<Condition> IncompleteConditions { get; private set; } = new HashSet<Condition>();

        9 references | 0 changes | 0 authors, 0 changes
        public int Depth { get; private set; }

        30 references | 0 changes | 0 authors, 0 changes
        public Node(Action action, Node parent = default)...

        6 references | 0 changes | 0 authors, 0 changes
        public Queue<Action> ToActionQueue()...

        2 references | 0 changes | 0 authors, 0 changes
        public float TotalCost()...

        0 references | 0 changes | 0 authors, 0 changes
        public override string ToString()...
    }
```

Ispis 16: Implementacija klase `Node` unutar `Planner` statičke klase

Uloga je `Node` klase da predstavlja čvor s kojim se može graditi mapa interakcija akcija. `Node` klasa nije generički čvor (engl. *node*), već je specijalizirana za GOAP algoritam.

`Node` se konstruira s jednim ili dva argumenta, s instancom akcije i čvora roditelja. Argument čvora roditelja opcionalan je i ako se ne proslijedi, bit će postavljen kao `null` pokazivač. Klasa ima tri pomoćne metode: `ToActionQueue`, `TotalCost` i `ToString`.

- `TotalCost` – vraća ukupnu cijenu od trenutnog čvora do najstarijeg čvora koji je s njim povezan.
- `ToString` – generirat će čitljiv ispis akcija koje su povezane čvorovima.
- `ToActionQueue` – vraća sortirani red akcija; od akcije najstarijeg čvora do akcije trenutnog čvora.

```
30 references | 0 changes | 0 authors, 0 changes
public Node(Action action, Node parent = default)
{
    Action = action;
    Parent = parent;
    Depth = 1;

    if (parent != default)
    {
        Depth += Parent.Depth;
        IncompleteConditions.UnionWith(Parent.IncompleteConditions);
    }

    IncompleteConditions.UnionWith(Action.Preconditions);
    IncompleteConditions.ExceptWith(Action.Effects);
}
```

Ispis 17: Implementacija konstruktora `Node` klase

Svojstva `Depth` i `IncompleteConditions` postavljaju se samo s konstruktorom. Na ispisu 17 vidljivo je da svojstvo `Depth` označava trenutnu dubinu na kojoj se čvor nalazi, tj. broj čvorova spojenih na njega umanjen za 1 jer je inicijalna dubina 1.

`IncompleteConditions` predstavlja listu stanja koja nisu još zadovoljena. Svaki novi kreirani čvor nasljeđuje nezadovoljena stanja od čvora roditelja. Tome se pribrajaju i sva stanja koja su preduvjet akcije koju čvor predstavlja, ali se brišu ona stanja koja su zadovoljena efektima akcije koju čvor predstavlja.

`Planner` implementira tri javne metode. Potpis tih metoda prikazan je na ispisu 18.

```
/// <summary> Default maximum action depth to reach specified goal. That is also ...
private const uint max_depth = 15;

/// <summary> Obtain cheapest action queue based on desired goal, current applie ...
26 references | 0 changes | 0 authors, 0 changes
public static Queue<Action> Plan(List<Action> actions,
                                 Goal goal,
                                 HashSet<Condition> achievedConditions = null,
                                 uint maxDepth = max_depth)...

/// <summary> Obtain all available paths for provided goal to maximum depth of m ...
23 references | 0 changes | 0 authors, 0 changes
public static List<(float cost, Queue<Action> actions)> GetAllAchievablePaths(List<Action> actions,
                                                                               Goal goal,
                                                                               HashSet<Condition> achievedConditions = null,
                                                                               uint maxDepth = max_depth)...

/// <summary> Get a string representation of provided actionChain
1 reference | 0 changes | 0 authors, 0 changes
public static string PrintActionChain(IEnumerable<Action> actionChain)...
```

Ispis 18: Javne metode statičke klase `Planner`

Opis zaduženja javnih metoda:

- `Plan` – Na osnovu liste akcija, `HashSeta` stanja i odabranog cilja, vraća najjeftiniji sortirani red akcija koje treba izvršiti kako bi se iz trenutnog stanja prešlo u stanje

zadano odabranim ciljem. Četvrti je parametar opcionalan, a označava dubinu do koje će se pokušati pronaći put do cilja. Iz ispisa je vidljivo da će, ako četvrti parametar nije naveden, on biti 15.

- `GetAllAchievablePaths` – Na osnovu istih parametra kao kod metode Plan vraća, do zadane dubine, sortiranu listu svih mogućih putanja do cilja. Lista je sortirana tako da je na prvom mjestu najjeftiniji put kojim je moguće doći do cilja, a na zadnjem je mjestu najskuplji put koji je pronađen, a da i dalje vodi do ciljanog stanja. Svaki element liste sastoji se od dvaju parametara. Prvi je ukupna cijena tog puta, a drugi je sortirani red akcija koje treba izvršiti u zadanom redoslijedu kako bi se iz trenutnog stanja prešlo u stanje zadano odabranim ciljem.
- `PrintActionChain` – Na osnovi niza akcija samo za čitanje ispisuje putanju akcija od prve do zadnje. Izgled ispisa vidljiv je na prvom zapisu na slici 11.

Na slici 11 vidljivo je da javne funkcije imaju XML komentare za lakše razumijevanje na koji način koristiti istu. U trenutnoj verziji ovog paketa nisu implementirani XML komentari za sve javne i zaštićene funkcionalnosti, ali je planirano u nekoj od sljedećih iteracija to ubaciti.

`Plan` i `GetAllAchievablePaths` najkompleksnije su implementacije algoritma, a ujedno su, uz agenta koji je opisan dolje u tekstu, srž GOAP sustava. U nastavku teksta detaljno je razloženo na koji način radi svaka od metoda. Ispis 19 prikazuje potpunu implementaciju metode Plan.

```
public static Queue<Action> Plan(List<Action> actions,
                                 Goal goal,
                                 HashSet<Condition> achievedConditions = null,
                                 uint maxDepth = max_depth)
{
    if (actions == default || goal == default)
        return default;

    achievedConditions ??= new HashSet<Condition>();

    if (goal.Outcome.IsSubsetOf(achievedConditions))
        return new Queue<Action>();

    return GetPath(actions, achievedConditions, goal, maxDepth)?.ToActionQueue() ?? default;
}
```

Ispis 19: Implementacija `Planner.Plan` metode

S ispisa 19 vidljivo je da se srž algoritma planiranja nalazi u metodi `GetPath` koja je opisana niže. U ovoj metodi rade se osnovne provjere kako se aplikacija ne bi srušila ako korisnik pošalje neočekivane parametre. Ako lista akcija ili cilj bude `null`, pokazivač metoda vraća `null` pokazivač, što znači da cilj nije ostvariv. Ako je na mjestu parametra postignutih stanja proslijeđen `null` pokazivač, metoda kreira prazan `HashSet` stanja i nastavlja dalje s planiranjem putanje do cilja. Zadnja provjera provjerava zadovoljava li trenutno stanje agenta efekt cilja. Ako da, nije potrebno raditi nijednu akciju za ostvarenje tog cilja.

S valjanim podacima poziva se metoda `GetPath`. Ako ta metoda vrati `null` pokazivač, to će biti proslijeđeno i s `Plan` razine, a ako vrati `Node` pozvat će se `Node` metoda `ToActionQueue` kako bi se dobio red akcija. Ispis 20 prikazuje potpunu implementaciju privatne statičke metode `GetPath`. U daljnjem tekstu opisano je na koji način metoda računa put.

```
private static Node GetPath(List<Action> actions, Goal goal, HashSet<Condition> achievedConditions, uint maxDepth)
{
    PriorityQueue<float, Node> queue = PopulateGoalNodes(actions, goal);

    while (!queue.IsEmpty)
    {
        (var cost, var node) = queue.Dequeue();

        if (node.Depth > maxDepth)
            continue;

        if (node.IncompleteConditions.IsSubsetOf(achievedConditions))
            return node;

        foreach (var action in actions)
        {
            if (action.Effects.IsSubsetOf(node.IncompleteConditions))
            {
                var childNode = new Node(action, node);
                queue.Enqueue(cost + action.Cost, childNode);
            }
        }
    }

    return default;
}
```

Ispis 20: Implementacija planiranja najjeftinijeg puta

Pri ulasku se poziva privatna statička metoda `PopulateGoalNodes`. Ona, na osnovu danih akcija i cilja, vraća prioritetni red s cijenom svake akcije i čvorom koji vodi do izvršenja zadanog cilja. Metoda radi na način da provjerava za svaku akciju jesu li efekti

cilja nadskup efekata akcije. Ako je to zadovoljeno, konstruira se čvor za tu akciju i dodaju se u prioritetni red. Metoda vraća kreirani red.

Nakon toga prolazi se kroz `while` petlju dok god dati prioritetni red nije prazan. Ako je prazan, metoda vraća `default` vrijednost, što bi za ovaj tip bio `null` pokazivač. Iz reda se dobavlja prvi element i dobiva se njegova trenutna cijena i čvor. Ako je taj čvor na većoj dubini od dozvoljene, odbacuje se iako je moguće da bi u baš u toj iteraciji pronašao put. Ako je dubina manja, gleda se jesu li nedovršena stanja čvora podskup proslijeđenih stanja koja su proslijeđena kao `achievedConditions` parametar.

`AchievedConditions` može biti stanje svijeta, agenta ili zbrojeno, ovisno o implementaciji agenta. Ako je to zadovoljeno, znači da je to najjeftiniji čvor koji sadrži putanju akcija koje se trebaju izvršiti kako bi se prešlo u stanje zadano ciljem. Ako uvjet nije zadovoljen, to znači da postoje stanja koja još nisu zadovoljena i koja se ne mogu zadovoljiti `achievedConditions` stanjima. U tom slučaju algoritam traži ima li akcija koje zadovoljavaju sve ili dio stanja koja nisu zadovoljena. Algoritam prolaz kroz cijeli niz akcija i provjerava je li efekt akcije podskup nedovršenih stanja. Ako da, konstruira se novi čvor i ubacuje ga se u prioritetni red. To se radi za sve čvorove jer je moguće da pronađena akcija nije jedini put. Nakon toga slijedi sljedeća iteracija dok god nije zadovoljen uvjet da nema čvorova u prioritetnom redu ili je pronađen čvor kojem je lista nezadovoljenih stanja podskup proslijeđenih stanja ili prazna.

`GetAllAchievablePaths` radi na sličan način kao i `Plan` metoda. Metoda je nešto kompleksnija pa je objašnjena u dijelovima. Ona prima istim redoslijedom iste parametre kao i `Plan` metoda. Prvi dio implementacije provjerava primljene podatke što se ne razlikuje od metode `Plan`. Na ispisu 21 vidi se početni dio metode `PopulateGoalNodes`. Vidi se da se poziva privatna statička metoda `PopulateGoalNodes` za dohvaćanje početnih čvorova koji direktno zadovoljavaju cilj ili dio cilja, kao i kod metode `Plan`.

```
if (actions == default || goal == default)
    return default;

achievedConditions ??= new HashSet<Condition>();

if (goal.Outcome.IsSubsetOf(achievedConditions))
    return new List<(float cost, Queue<Action> actions)>();

uint current_depth = 1;
var priorityQueue = PopulateGoalNodes(actions, goal);
var queue = new Queue<Node>();
var achievedNodes = new Queue<Node>();

foreach (var costNodePair in priorityQueue)
    queue.Enqueue(costNodePair.Value);
```

Ispis 21: Početni dio implementacije `GetAllAchievablePaths`

Ova metoda nema potrebu imati par cijena – čvor i zbog toga se, od povratne vrijednosti `PopulateGoalNodes`, puni red čvorova.

Drugi dio metode prikazan je na ispisu 22. Vidi se da metoda ulazi u `while` petlju koja više ne ovisi o tome koliko se čvorova ima, već se gleda trenutna i maksimalna dubina. Potencijalna nelogičnost može nastati ako je proslijeđena maksimalna dubina 0 ili neki broj manji od 0 jer će biti tretiran kao 1 što je i navedeno u XML komentarima potpisa metode. Razlog tome je što 0 ili broj manji od 0 nema smisla koristiti kao maksimalnu dubinu. Potom, kako je vidljivo na ispisu 22, metoda provjerava postoji li ijedan čvor. Ako ne, put nije moguć jer nijedna akcija potpuno ili parcijalno ne zadovoljava cilj. Ako čvorovi postoje, za svaki se gleda jesu li sva nedovršena stanja čvora ispunjena. To se postiže brisanjem stanja iz seta nedovršenih stanja koja su već zadovoljena parametrom primljenih `achievedConditions` stanja. Ako je to zadovoljeno, čvor se dodaje u red završenih čvorova, a ako nije, dodaje ga se u red za daljinu obradu.

U zadnjem dijelu petlje, prije uvećavanja trenutne dubine, prolazi se kroz listu akcija i provjerava jesu li efekti ijedne akcije podskup nedovršenih stanja čvorova. Za one čvorove koji imaju akciju koja zadovoljava gornji uvjet, konstatira se novi čvor i dodaje ga se u red. Ostali čvorovi koji ne zadovoljavaju uvjet, odbacuju se.

```
while (current_depth < maxDepth)
{
    if (queue.Count == 0)
        break;

    List<Node> nodesToProcess = new List<Node>();

    while (queue.Count != 0)
    {
        var node = queue.Dequeue();
        node.IncompleteConditions.ExceptWith(achievedConditions);

        if (node.IncompleteConditions.Count != 0)
            nodesToProcess.Add(node);
        else
            achievedNodes.Enqueue(node);
    }

    foreach (var node in nodesToProcess)
    {
        foreach (var action in actions)
        {
            if (action.Effects.IsSubsetOf(node.IncompleteConditions))
            {
                var nextNode = new Node(action, node);
                queue.Enqueue(nextNode);
            }
        }
    }

    current_depth++;
}
```

Ispis 22: Centralni dio implementacije `GetAllAchievablePaths`

Nakon što algoritam dođe do maksimalne dubine ili u redu više nema čvorova za obraditi, izlazi se iz petlje i dodaju se u listu svi oni čvorovi koji su prethodno dodani u red završenih čvorova. Prije dodavanja potrebno ih je pretvoriti u red akcija.

Zbog poštivanja dubine algoritam još jednom provjerava ima li ijedan čvor potpunu putanju koji je ostao u listi za obradu. Ako ima, i on se pretvara u red akcija i dodaje u povratnu listu. Na kraju, kao što je vidljivo na ispisu 23, sortiraju se nizovi akcija po cijeni tako da najjeftiniji niz akcija bude prvi element povratne liste, a najskuplji zadnji.

```
List<(float cost, Queue<Action> actions)> validPaths = new List<(float cost, Queue<Action> actions)>();

foreach (var pathNode in achievedNodes)
    validPaths.Add((pathNode.TotalCost(), pathNode.ToActionQueue()));

foreach (var pathNode in queue)
    if (pathNode.IncompleteConditions.IsSubsetOf(achievedConditions))
        validPaths.Add((pathNode.TotalCost(), pathNode.ToActionQueue()));

return validPaths.OrderBy(x => x.cost).ToList();
```

Ispis 23: Krajnji dio implementacije `GetAllAchievablePaths`

3. 3. 5. Agent

`Agent` je klasa koja objedinjuje algoritam u jednu cjelinu. Osnovna funkcionalnost joj je da na osnovi sadržanih akcija i ciljeva odabere onaj cilj za koji agent ima najveću želju da se izvrši. Agent potom poziva `Planner` klasu kako bi dobio put od trenutnog stanja do stanja opisanog ciljem. Ako je put pronađen, agent izvršava akcije jednu po jednu do postignutog cilja, a tad se proces ponovno ponavlja. Ako cilj nije moguće izračunati, agent odabire sljedeći cilj za koji ima najveću želju za izvršavanje dok ne pronađe cilj koji je moguće izvršiti. Ispis 24 prikazuje metode agenta koje bi prilikom nasljeđivanja `Agent` klase trebalo spojiti s nekim Unity porukama.

```
3 references | 0 changes | 0 authors, 0 changes
protected void Init()
{
    var goals = GetComponents<Goal>().ToList();
    ResetUrge(goals);

    foreach (var goal in goals)
        AddGoal(goal);

    var actions = GetComponents<Action>();
    this.actions.AddRange(actions);

    agentState = new AgentState(this);
}
2 references | 0 changes | 0 authors, 0 changes
protected void Init(float abortThreshold, float abortTimer)
{
    Init();
    this.abortThreshold = abortThreshold;
    InvokeRepeating("HandleAbortSequence", abortTimer, abortTimer);
}
3 references | 0 changes | 0 authors, 0 changes
protected virtual void Clear()
{
    goals.Clear();
    actions.Clear();
    conditions.Clear();
    actionChain?.Clear();
    activeAction = null;
    activeGoal = null;
    CancelInvoke();
}

3 references | 0 changes | 0 authors, 0 changes
protected void Execute(ActionArgs args)
{
    if (HasPlan || (!agentState.IsEqual(this) && GetActionPLan()))
        ExecutePlan(args);
}
```

Ispis 24: Metode agenta koje je potrebno spojiti s Unity pozivom

Potrebno je odabrati jedan od `Init` poziva, ovisno o tome želi li se imati mogućnost zaustaviti izvršavanje cilja koji je odabran ili ne. `Init` metodom automatski se povlače svi ciljevi i akcije koje su na istom Unity objektu kao i `Agent` klasa. Prilikom dohvaćanja ciljeva poziva se `ResetUrge` koji za svaki element iz liste ciljeva postavlja množitelja

bazne želje na 1 kako je prikazano na ispisu 25. Razlog je tome što se baznom željom izražava cilj kojem agent teži, a uloga množitelja zamišljena je da dinamički mijenja izgled za pojedini cilj, npr. situacija u kojoj se agent nakon obavljanja posla umara. U početku bi agent imao postavljenu malu želju za odmor tako da je želja za rad veća. Njegov se umor može povezati na množitelja želje za cilj odmor. Tako bi agent, nakon što dosegne neku kritičnu proizvoljnu točku određenu od strane implementatora, odabrao odmor umjesto rada, a kad bi se odmorio, mogao bi nastaviti s radom. Za promjenu množitelja bazne želje iz `Agent` klase potrebno je pozvati metodu `SetGoalUrgeMultiplier` i navesti cilj ili `IGoalInfo` koji se želi promijeniti, kao i vrijednost na koju se želi postaviti množitelj bazne želje.

```
private void ResetUrge(List<Goal> goals) => goals.ForEach(g => g.UrgeMultiplier = 1);
```

Ispis 25: Postavljanje množitelja bazne želje na 1

Nadalje, sa ispisa 24 vidljivo je da se nakon resetiranja množitelja bazne želje dodaju ciljevi s objekta pozivom metode `AddGoal`. Ta se metoda brine da svi ciljevi budu dodani redoslijedom, od onoga s najvećom ukupnom željom, do onoga s najmanjom.Zatim se konstruira struktura `AgentState` prosljeđivanjem `Agenta`. `AgentState` je pomoćna privatna struktura koja služi kod optimizacije agenta i koja je pojašnjena niže u tekstu.

Kod `Init` metode s dva parametra: `abortThreshold` i `abortTimer` i dalje se poziva `Init` metoda bez parametra, ali će se uz to postaviti `abortThreshold` na zadanu proslijeđenu vrijednost. Podsjetimo, `abortThreshold` označava najmanju razliku trenutnog i potencijalnog cilja za izvršavanje. Drugi parametar, `abortTimer` označava svako koliko će agent provjeriti postoji li cilj sa željom većom od želje trenutnog cilja uvećanog za `abortThreshold`. To se ostvarilo pomoću metode `InvokeRepeating` koja opetovano poziva zadanu metodu nakon `abortTimer` vremena. Metoda koja se poziva je `HandleAbortSequence`. Implementacija metode može se vidjeti na ispisu 26.

```
private void HandleAbortSequence()
{
    if ((!activeAction?.Abortable) ?? true)
        return;

    foreach (var goal in goals)
    {
        var urgeDiff = goal.Urge - activeGoal.Urge;

        if (urgeDiff < abortThreshold)
            return;

        if (goalsUnachievable.Contains(goal))
            continue;

        Queue<Action> actionChain = Planner.Plan(actions, goal, conditions);

        if (actionChain == null || actionChain.Count == 0)
        {
            goalsUnachievable.Add(goal);
            continue;
        }

        AbortGoal();
        this.actionChain = actionChain;
        ActivateNewGoal(goal);
        return;
    }
}
```

Ispis 26: Implementacija `HandleAbortSequence` metode

Metoda `HandleAbortSequence` prvo provjerava postoji li trenutna akcija koja se izvršava i, ako postoji, može li se ista prekinuti. Ako je na ijedno od tih pitanja odgovor ne, metoda završava s radom. Ako je odgovor na sve da, metoda prolazi kroz petlju svih ciljeva iz liste. Iako su ciljevi raspoređeni po trenutnoj želji, nije dovoljno provjeriti samo prvi cilj. Razlog tome je što je moguća situacija gdje cilj s najvećom željom nije moguće izvršiti, ali neki cilj s idućom najvećom željom, čija je želja i dalje veća od trenutnog cilja uvećanog za `abortThreshold`, moguće je izvršiti. Potom se računa razlika želja trenutnog aktivnog cilja i cilja u nizu. Ako je ona veća od `abortThresholda`, nastavlja se s izvršavanjem metode, u suprotnom, metoda završava jer svi sljedeći u listi sigurno ne zadovoljavaju `abortThreshold`. Na sljedećoj liniji provjerava se sadrži li lista `goalsUnachievable` odabrani cilj. Ako je odgovor da, to znači da je u nekoj ranijoj provjeri već utvrđeno da ne postoji put za taj cilj. Kako se ništa nije promijenilo u agentu, ne računa se ponovo te se izlazi iz metode.

U suprotnom, koristeći `Planner` klasu, pokušava se pronaći put do cilja. U slučaju da put ne postoji ili je cilj već zadovoljen, dodaje se u listu nedostižnih ciljeva i metoda nastavlja dalje kroz petlju. Ako se pronađe cilj koji je moguće ostvariti, trenutna se akcija može prekinuti. Poziva se prekid te akcije i kreće s izvršavanjem novog plana. Prekid akcije izaziva događaj `OnGoalAborted`. Treba napomenuti da kod prekida izvršavanja nekog cilja, isti i dalje ostaje u listi ciljeva.

S ispisa 24 vidljiva je implementacija metode `Execute`. Prije provedbe plana u petlji provjerava se ima li agent trenutni aktivni plan, ako nema, sljedeće što se provjerava jest da se stanje agenta promijenjeno u odnosu na prije. Stanje agenta „snima“ se u dva slučaja: u prvom slučaju kad se aktivira novi cilj, a u drugom slučaju kad nijedan cilj nije ostvariv s trenutnim stanjem agenta. Budući da `Execute` metoda mora biti spojena na neku Unity `Update` poruku kako bi se akcije mogle izvršavati, to omogućava da se ne troše resursi na ponovno računanje ciljeva za koje ne postoji put. `AgentState` struktura pamti broj ciljeva, akcija, stanja agenta kao i trenutni aktivan cilj. Zbog toga će se, ako se broj ciljeva, stanja ili akcija promjeni, ponovo pokušati pokrenuti neki cilj.

Jednom kad se pronađe put do nekog cilja, metoda poziva `ExecutePlan` metodu dok god se ne izvrši zadani cilj. Implementacija `ExecutePlan` metode prikazana je na ispisu 27. Treba napomenuti da se za sve akcije u redu za izvršavanje, prije izvršavanja ijedne akcije, pozove metoda `ResetState`. Metoda se može pregaziti iz `Action` klase.

```
private void ExecutePlan(ActionArgs args)
{
    if (activeAction.Completed)
        if (!actionChain.Any())
        {
            if (Debuging)
                Debug.Log($"Goal completed: {activeGoal}");

            OnGoalCompleted?.Invoke(this, new GoalInfoEventArgs { Goal = activeGoal });

            RemoveGoal(activeGoal);
            activeGoal = null;
            activeAction = null;

            return;
        }
        else
        {
            activeAction = actionChain.Dequeue();
        }

    activeAction.Execute(args);
}
```

Ispis 27: Implementacija `ExecutePlan` metode

Metoda `ExecutePlan` provjerava je li trenutna aktivna akcija dovršena, ako je, provjerava ima li akcija u nizu koje je potrebno izvršiti za postići efekt cilja. Ako je to slučaj, uzima se sljedeća akciju iz reda. Neovisno o tome je li trenutna akcija dovršena ili je odabrana nova akcija, poziva se `Execute` metoda iz `Action` klase. Samo u slučaju kad je trenutna akcija dovršena i nijedna akcija nije ostala u nizu, proglašava se postignut cilj i poziva se događaj `OnGoalCompleted` te se uklanja završeni cilj iz liste ciljeva.

Na ispisu 28 vidljiv je potpis `GetActionPlan` metode. Po implementaciji agenta on poziva `Planner.Plan` metodu, međutim, metoda se može pregaziti kod nasljeđivanja agenta te se može drugačije implementirati logika odabira cilja. Jedan od načina na koji to može biti ostvareno jest korištenjem metode `Planner.GetAllAchievablePaths`. Ona vraća sve moguće puteve za neki cilj i omogućuje proizvoljno biranje puta do zadanog cilja. Po baznoj implementaciji metoda vraća najkraći put do cilja.

```
protected virtual Queue<Action> GetActionChain(List<Action> actions, HashSet<Condition> conditions, Goal goal)
```

Ispis 28: Potpis `GetActionChain` metode

Agentu je moguće dodavati ili brisati nova stanja i ciljeve. Isto nije implementirano za akcije. Razlog je što bi to omogućilo neka nedefinirana stanja i ponašanja te bi se tako povećala mogućnost greške. Primjerice, u slučaju implementacije trebalo bi odlučiti što će se dogoditi ako se izbriše akcija koja se trenutno izvršava ili koja je u redu za izvršavanje. Donesena implementacijska odluka o tome možda ne bi bila svima intuitivna. Najbolji je pristup staviti na agenta sve akcije koje će agent ikad moći izvršiti, a za akcije koje se želi limitirati, npr. kad je agent u posebnom dijelu razine ili slično, potrebno je koristiti preduvjete akcije i stanja agenta. Primjer: ako se želi postići da na razini 3 igrač može napraviti akciju X, može se definirati uvjet `level_3` koji će biti preduvjet akciji X. Jednom kad igrač dođe do razine 3, može se svim agentima postaviti stanje level_3 te će tako ta akcija biti „otključana“ agentima za korištenje. Treba napomenuti da, ako se izbriše neko stanje agenta koje je bilo potrebno u planiranju trenutnog puta cilja koji se izvršava, ili se pak izbriše cilj koji se trenutno izvršava, isti se neće prekinuti. Razlog tome je što se ne može predvidjeti koje se akcije mogu, a koje ne mogu prekinuti. Međutim, ako korisnik želi da se cilj prekine i sve su akcije postavljene kao akcije koje se mogu prekinuti, tada se to može ostvariti pozivom metode `SetGoalUrgeMultiplier` i postavljanjem množitelja bazne

želje na 0 za trenutni cilj. Moguće je da će korisnik morati povećati množitelj za neki drugi cilj kako bi se pokrenula sekvenca prekida izvršavanja trenutnog cilja.

3. 3. 6. Unity editor proširenje

Osim algoritma kroz `iT GOAP` paket dodano je i proširenje za Unity *editor*. Ono je definirano kao posebna dinamička biblioteka `it.goap.editor` što je vidljivo na slici 14. Prikaz tog proširenja prikazan je na slici 12. Glavna mu je uloga da prilikom testiranja igre iz Unity editora može, u stvarnom vremenu, kroz jednostavan grafički prikaz, vidjeti stvarno stanje odabranog agenta.

Unity je razvijao mnoge različite Unity funkcionalnosti. Dogodi se da Unity odustane od neke funkcionalnosti nakon nekoliko godina kako bi istu zamijenio nekom novom, modernijom implementacijom. Zbog toga Unity danas nema internu funkcionalnost za mrežnu komponentu iako je istu moguće ostvariti pomoću Unity paketa. Problem je kod toga što se mrežni paketi oslanjaju na staru Unity implementaciju mrežne komponente koja će u budućnosti u potpunosti biti izbačena jer je trenutno je u razvoju nova mrežna funkcionalnost u Unityju.

Nešto je drugačija situacija s funkcionalnošću korisničkog sučelja, UI-a (*user interface*). Trenutno postoje tri različite izvedbe UI sustava u Unityju i sve su još u upotrebi. Da je situacija složena, može se vidjeti na slici 15 koja je izvučena iz Unity dokumentacije za verziju 2020.3.[8]

Type of UI	UI Toolkit	Unity UI(uGUI)	IMGUI	Notes
Runtime (debug)	✓*	✓	✓	This refers to temporary runtime UI used for debug purposes.
Runtime (in-game)	✓*	✓	Not Recommended	For performance reasons, Unity does not recommend IMGUI for in-game runtime UI.
Unity Editor	✓	✗	✓	You cannot use Unity UI to make UI for the Unity Editor.

Slika 15: Unity 2020.3 UI sustavi [8]

Unity UI sustavi mogu se koristiti na tri različita načina. Ne mogu se svi UI sustavi koristiti u svim načinima, a i drugačije su preporuke za različite sustave. Kod implementiranja Unity editor proširenja korišten je UI Toolkit. Problem je tog sustava što je još u razvoju. Iako je u potpunosti implementiran za editor, za korištenje kao glavni UI u Unity aplikaciji potrebno je instalirati neke pakete koji su još u razvoju i stoga nedostaju mnoge funkcionalnosti. Iz tog razloga kod implementacije igre korišten je Unity UI (uGUI)

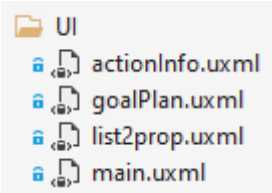
koji je još uvijek standard kod implementiranja UI-a u Unity igrama. Unity UI nije moguće koristiti za Unity *editor*, stoga se prije dolaska UI Toolkita morao koristiti IMGUI paket. Zamišljeno je da UI Toolkit u budućnosti u potpunosti zamijeni ostale UI Unity sustave. Inspiraciju za razvoj vukao je iz web tehnologija današnjice pa su osnovni koncepti slični današnjim konceptima kod razvoja web tehnologija. [9]

UI Toolkit sastoji od tri komponente:

- UXML (*Unity Extensible Markup Language*)
- USS (*Unity style sheets*)
- Implementacija interakcija pomoću C# klasa.

U usporedbi s web tehnologijama UXML bi predstavljao HTM komponentu s kojim je i inspiriran uz XAML i XML. UXML se koristi kako bi se definiralo sučelje UI-a, a USS se može koristiti kako bi se definirao izgled i stil sučelja. USS je jasno inspiriran CSS-om. [10, 11]

U implementaciji proširenja nisu korišteni USS klase, već je sve implementirano preko UXML-a. Za implementaciju je bilo potrebno definirati 4 UXML dokumenta. Slika 16 prikazuje listu UXML dokumenata implementiranu za GOAP proširenje, a tablica 3 prikazuje opis zaduženja za svaki definirani UXML dokument.



Slika 16: Definirane UXML datoteke za GOAP Unity *editor* proširenje

Tablica 3: Opis zaduženja UXML dokumenata

IME	OPIS
`main`	Kontejner kojemu je krajnji rezultat prikazan na slici 12. Struktura mu je podijeljena na dva dijela. Gornji dio definira izgled sučelja osnovnih podataka o agentu, a donji se dio grana na tri dijela. S lijeve su strane dvije liste, `condition` i `action` lista. Svaka lista koristi `List2prop.uxml` datoteku za prikaz informacija. S desne su strane informacije o trenutnom cilju kao i akcije koje agent treba izvršiti uz akciju koja se trenutno izvršava. Koristi `goalPlan.uxml` datoteku za prikaz informacija o cilju.
`goalPlan`	Kontejner koji sadrži strukturu za prikaz osnovnih informacija o cilju, kao i listu akcija koju je potrebno izvršiti kako bi cilj bio zadovoljen. Svaka akcija sadrži zasebnu `actionInfo.uxml` datoteku za prikaz.
`actionInfo`	Sadrži predložak za prikaz jedne akcije, može se vidjeti ime akcije na vrhu, a sa strana u kvadratu sadrži preduvjete i efekte akcije, za što koristi `List2prop.uxml` datoteku. S lijeve strane u kvadratu prikazani su preduvjeti, dok su efekti prikazani s desne strane.
`List2prop`	Generički predložak za jednostavan prikaz liste s dvije strane, lijeve i desne. Koristi se za prikaz stanja agenta i akcija koje agent može izvršiti u `main.uxml` datoteci kao i za prikaz preduvjeta i efekata za pojedinu akciju u `actionInfo.uxml` datoteci.

UXML se može pisati u bilo kojem tekstualnom editoru kako je prikazano na slici 17. Na slici je vidljivo da stilovi nisu izdojeni u posebne USS datoteke te da su svojstva svih elemenata zapisana u UXML-u. Za iste se može definirati klasa unutar USS datoteke te koristiti klasa kao kod CSS-a da se opiše element.

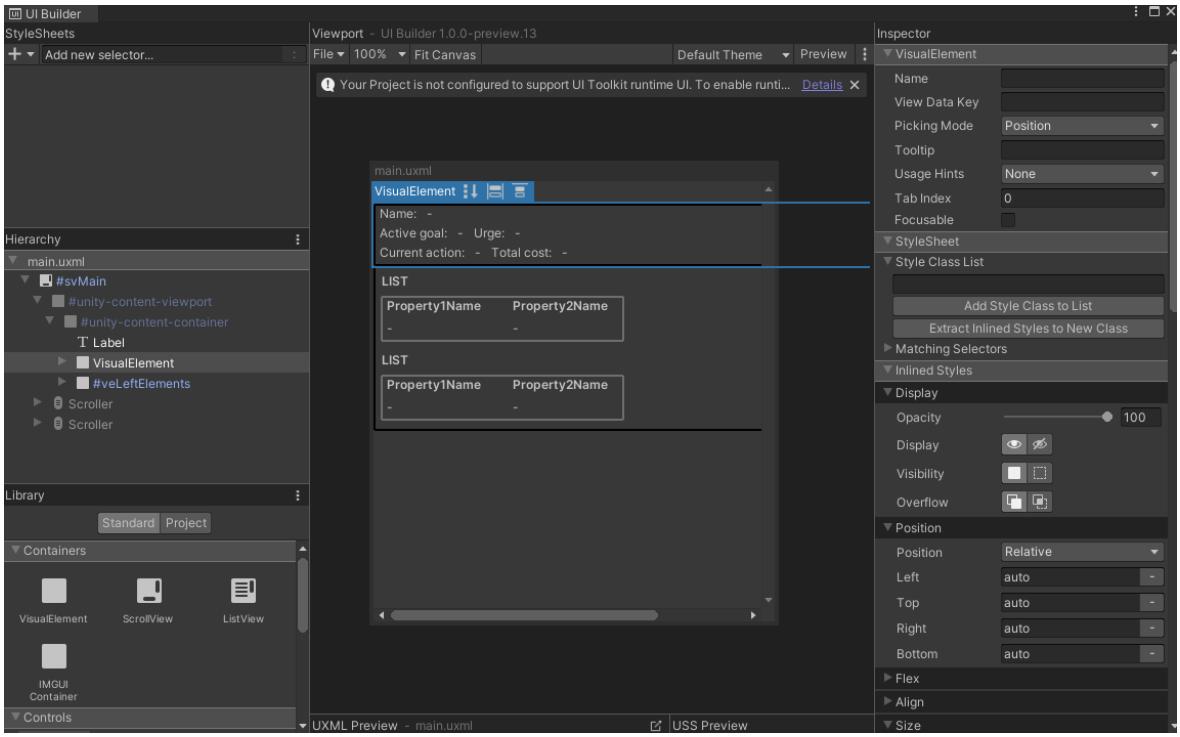
```
<ui:UXML xmlns:ui="UnityEngine.UIElements" xmlns:uie="UnityEditor.UIElements">
    <ui:Label text="Label" name="txtName" />
    <ui:VisualElement style="border-left-width: 2px;
border-right-width: 2px;
border-top-width: 2px;
border-bottom-width: 2px;
border-top-left-radius: 2px;
border-bottom-left-radius: 2px;
border-top-right-radius: 2px;
border-bottom-right-radius: 2px;
border-left-color: rgb(0, 0, 0);
border-right-color: rgb(0, 0, 0);
border-top-color: rgb(0, 0, 0);
border-bottom-color: rgb(0, 0, 0);
flex-direction: row;
min-width: auto;
width: 100%;">
        <ui:VisualElement name="vePrecondition" style="flex-direction: column;
margin-bottom: 3px;
width: 50%;
padding-left: 2px;
padding-right: 1px;
padding-top: 1px;
padding-bottom: 1px;" />
        <ui:VisualElement name="veEffects" style="flex-direction: column;
margin-bottom: 3px;
width: 50%;
padding-left: 1px;
padding-right: 2px;
padding-top: 1px;
padding-bottom: 1px;" />
    </ui:VisualElement>
</ui:UXML >
```

Slika 17: UXML implementacija actionInfo

Osim ručnog editiranja UXML i USS datoteke u tekst *editoru* može se instalirati UI Builder Unity paket koji je u razvoju. UI Builder je Unity proširenje za grafičko editiranje UXML i USS datoteka. Slika 18 prikazuje editiranje main UXML datoteke koristeći UI Builder proširenje u Unityju.

S lijeve strane od vrha prema dnu definirani su stilovi koji se mogu koristiti na elementima. Nakon toga slijedi hijerarhija elemenata, te na dnu, biblioteka elemenata koja se može koristiti za definiranje prikaza. U sredini se nalazi vizualna reprezentacija cijele datoteke. S desne strane prikazana je lista svojstava za svaki element. Jednom definirani stil elemenata može se jednostavno izvući kao nova klasa stilova u nekoj USS datoteci.

Na slici je vidljivo kako je zamišljeno da se UXML koristi kao predložak pozicija elemenata koji se dinamički popunjava iz koda. U ovom proširenju zamišljeno je da se selektiranjem na objekt koji sadrži Agent klasu popuni ovaj predložak s informacijama o trenutno selektiranom agentu.



Slika 18: Editiranje `main` uxml datoteke koristeći UI Builder proširenje

Dinamičko popunjavanje predloška realizirano je implementacijom klase `EditorWindowGoalPath`. Na ispisu 29 vidljivo je na koji je način implementirano dodavanje putanje na glavnu traku u Unity *editoru*. Dovoljno je postaviti atribut `MenuItem` s punom putanjom do proširenja. U kodu se zatim konstruira novi prozor s `EditorWindowGoalPath` klasom.

```
[MenuItem("Window/GOAP/Goal path")]
0 references | 0 changes | 0 authors, 0 changes
public static void OpenEditorWindow()
{
    var window = EditorWindow.GetWindow<EditorWindowGoalPath>(main_title);
    window.Show();
}
```

Ispis 29: Definiranje putanje na kojoj se nalazi proširenje za Unity

`EditorWindowGoalPath` nasljeđuje od Unity klase `EditorWindow` koja ima drugačije Unity poruke u odnosu na MonoBehaviour klasu. Svaka `EditorWindow` klasa ima `rootVisualElement` varijablu koja predstavlja vizualni korijen dokumenta. Za prikaz predloška potrebno je dodati isti u korijen. To je prikazano na ispisu 30 kod `OnEnable` poruke.

```
private void OnEnable()
{
    var root = rootVisualElement;

    visualTree = Resources.Load<VisualTreeAsset>(main_resources_location);
    visualTree.CloneTree(root);
}
```

Ispis 30: `OnEnable` kopiranje main UXML-a u korijen

Sa slike je vidljivo kako se dohvaća `main.uxml` datoteka s putanje na kojoj se nalazi te se potom klonira u korijen elementa `EditorWindow` klase.

Zadnja Unity poruka koja je implementirana prikazana je na ispisu 31.

```
Unity Message | 0 references | 0 changes | 0 authors, 0 changes
private void Update()
{
    var agent = Selection.activeGameObject?.GetComponent<Agent>();

    try
    {
        if (agent)
        {
            SetupGeneralInformation(agent);
            SetupConditionList(agent);
            SetupActionList(agent);
            DrawMainPlan(agent);
        }
    }
    catch (Exception e)
    {
        Debug.LogError(e.ToString());
    }
```

Ispis 31: Implementacija `Update` Unity eventa

U prvoj liniji pokušava se dohvatiti Agent klasa iz trenutno selektiranog elementa na sceni. Ako ništa nije selektirano, ne dohvaća se `Agent` klasa. Ako agent skripta postoji, šalje se u različite metode zadužene za popunjavanje pojedinih elemenata `main` predloška. Zaduženja su pojedinih metoda:

- `SetupGeneralInformation` – zadužena za popunjavanje osnovnih informacija o agentu
- `SetupConditionList` – zadužena za popunjavanje svih stanja agenata

- `SetupActionList` – zadužena za popunjavanje svih akcija koje agent može izvesti, s pripadnim cijenama akcija
- `DrawMainPlan` – zadužena za ispisivanje osnovnih informacija o trenutno odabranom cilju, kao i za crtanje redoslijeda akcija koje se izvršavaju sljedeće. Za svaku akciju prikazuju se osnovne informacije o akciji.

Sve zajedno je omeđeno u `try-catch` blok iako su unutarnje metode pisane defenzivno te se ne očekuje ulaz u `catch` blok. Primjer jedne od gore navedenih metoda dan je na ispisu 32.

```
private void SetupActionList(Agent agent)
{
    var root = rootVisualElement;

    var conditions = root.Q<TemplateContainer>(action_list);
    conditions.Q<Label>(_name).text = "ACTIONS";

    var list = conditions.Q<VisualElement>(EditorWindowGoalPath.list);
    list.Clear();

    VisualElement header = visualTree.CloneTree().Q<VisualElement>(list_header);
    header.Q<Label>(prop_1_name).text = "ACTION";
    header.Q<Label>(prop_2_name).text = "COST";
    list.Add(header);

    foreach (var action in agent.Actions)
    {
        VisualElement item = visualTree.CloneTree().Q<VisualElement>(list_item);
        item.Q<Label>(prop_1_name).text = action.Name;
        item.Q<Label>(prop_2_name).text = action.Cost.ToString();
        list.Add(item);
    }
}
```

Ispis 32: Implementacija metode za prikaz akcija koje agent može izvršiti

Prvo se iz korijena dohvaća `TemplateContainer` imena `action_list`. To se radi koristeći `UQuery` (set metoda za dohvaćanje elemenata iz vizualnog stabla, korijena stabla; optimiziran je i za mobilne aplikacije). [12] U našem slučaju element koji se dodaje predstavlja kontejner koji se popunjava s akcijama koje agent može izvršiti. Nakon dodavanja zaglavlja u korijen, postavlja se tekst u zaglavlje i tekst tog dijela kontejnera. Prolazeći kroz listu akcija, dodaje se jedan po jedan red za svaku akciju u korijen. Prije dodavanja popune se njezini elementi imenom akcije s lijeve strane i cijenom s desne strane.

3. 3. 7. Kreiranje Unity paketa

Za kreiranje Unity paketa potrebno je posložiti strukturu foldera na način kako to preporučuje Unity dokumentacija. Na slici 19 vidi se preporučena struktura.

```
<root>
  ├── package.json
  ├── README.md
  ├── CHANGELOG.md
  ├── LICENSE.md
  ├── Third Party Notices.md
  ├── Editor
  │    ├── Unity.[YourPackageName].Editor.asmdef
  │    └── EditorExample.cs
  ├── Runtime
  │    ├── Unity.[YourPackageName].asmdef
  │    └── RuntimeExample.cs
  ├── Tests
  │    ├── Editor
  │    │    ├── Unity.[YourPackageName].Editor.Tests.asmdef
  │    │    └── EditorExampleTest.cs
  │    └── Runtime
  │         ├── Unity.[YourPackageName].Tests.asmdef
  │         └── RuntimeExampleTest.cs
  ├── Samples~
  │         ├── SampleFolder1
  │         ├── SampleFolder2
  │         └── ...
  └── Documentation~
       └── [YourPackageName].md
```

Slika 19: Preporučena struktura direktorija za Unity paket

Prije implementacije testova za paket, radi jednostavnosti testiranja i ažuriranja paketa, kreiran je novi Unity projekt. U njemu je GOAP paket dodan kao git podmodul projekta koristeći git *submodule* funkcionalnost. Git *submodule* je funkcionalnost koja omogućava da git projekt sadrži druge git projekte koji se ažuriraju neovisno o glavnom projektu. Tako nema duplikata i neovisno se može mijenjati sadržaj jednog ili drugog git projekta. [13] Slika 20 prikazuje listu podmodula novokreiranog projekta.

```
$ git submodule
878d0479ae1397c8f7ef0c57a43382eafce89b46 goap (heads/d108)
```

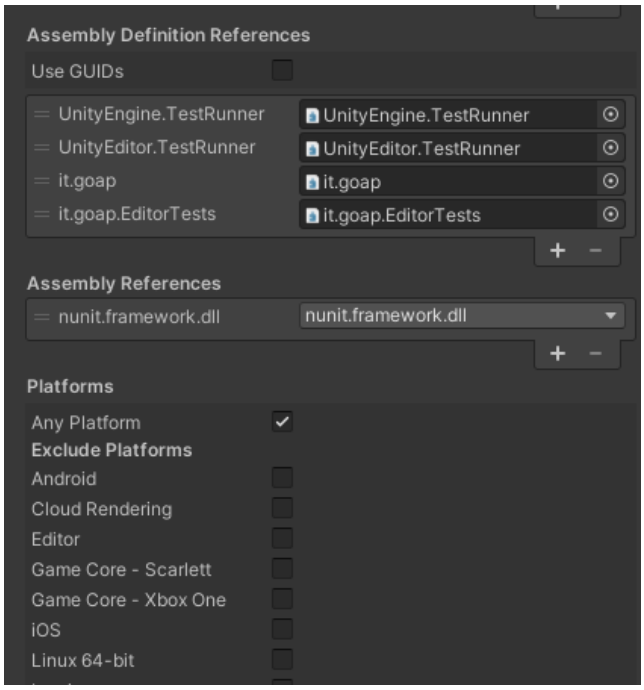
Slika 20: GOAP projekt kao git podmodul

Isto se može provjeriti u novokreiranoj datoteci .gitmodules. U toj datoteci zapisani su svi konfiguracijski podaci git podmodula. Podaci za GOAP podmodul mogu se vidjeti na slici 21.

```
[submodule "Packages/goap"]
        path = Packages/goap
        url = https://github.com/i1Tafra/iT-GOAP.git
```

Slika 21: Konfiguracijski podaci GOAP submodula

Nadalje, na slici 14 vidljivo je da je ovaj paket podijeljen u četiri odvojene biblioteke. To nije opcionalno, nego paket mora sadržavati posebne dinamičke biblioteke, što je i vidljivo sa slike 11. *Assembly definition* je konfiguracijska datoteka s uputama na koji način izgraditi dinamičku biblioteku.

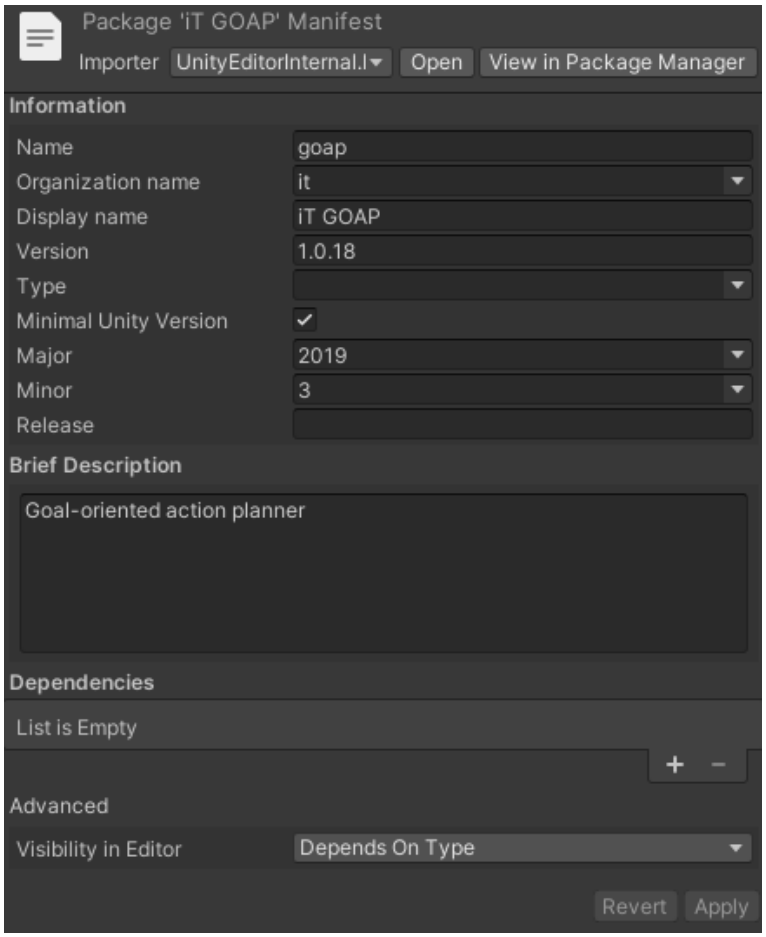


Slika 22:. Dio konfiguracije za dinamičku biblioteku Editor testova

Slika 22 prikazuje dio konfiguracijske datoteke za kreiranje *runtime* testova na zadanoj platformi. U gornjem dijelu slike potrebno je povezati dinamičke biblioteke koje se koriste u ovoj dinamičkoj biblioteci. `It.goap` je dinamička biblioteka u kojoj su

implementirane klase koje se testiraju, a `it.goap.EditorTests` implementira neke pomoćne funkcionalnosti koje se koriste kod testiranja. Za testiranje se koristi NUnit, a to je razvojni okvir (engl. *framework*) za testiranje u C# programskom jeziku. Zadnja opcija sa slike definira na kojim se platformama mogu pokrenuti testovi. U slučaju sa slike odabrane su sve platforme. Da se, primjerice, radilo o *editor* testovima ili proširenju, bilo bi potrebno označiti samo *editor* platformu. Ostale konfiguracijske datoteke slične su ovoj. Sve konfiguracijske datoteke mogu se uređivati korištenjem Unity grafičkog sučelja ili u bilo kojem tekstualnom *editoru*.

Zadnja datoteka koju je potrebno definirati kako bi paket bio prepoznat jest *package manifest*. Na slici 23 vidljivo je kako je implementiran za `iT GOAP` Unity *package*.



Slika 23: *Package manifest* iT GOAP Unity paketa

U manifestu je potrebno ispuniti podatke poput imena, imena organizacije, ime pod kojim se paket prezentira krajnjem korisniku i slično. Opcionalno se može navesti i kojeg je tipa paket, kratki opis paketa i minimalna verziju Unity razvojnog okruženja da bi paket ispravno funkcionirao. Verzija 2019.3 navedena je kao minimalna jer je za Unity proširenje

potreban UI Toolkit kojeg nije bilo u starijim verzijama, a i Unity je nedavno prošao kroz modernizaciju C# kompajlera te se danas mogu koristiti i značajke C# 8 standarda.
Zadnje stavke koje su bitne za navesti jesu ovisnost o drugim paketima i trenutna verzija. Ovaj paket nema vanjskih ovisnosti pa je lista prazna, a verzija se povećava svakim novim git *commitom*. Tako korisnik može ponovo navesti putanju do *master* grane ovog git projekta kako bi Unity zaključio da se na *master* grani nalazi novija verzija paketa te se povlači nova i briše stara verzija.

Bitno je imati na jednom mjestu jednostavan pregled promjena kroz revizije paketa. Zbog toga je implementirana i datoteka s promjenama (engl. *changelog*) u kojoj se zapisuje što je sve dodano, promijenjeno i izbrisano iz paketa. Na ispisu 33 vidljive su promjene za verziju 1.0.18 i 1.0.17.

```
## [1.0.18]  2020-07-12
### Changed
- Update Action FSM enum to match method names
- Make Goal abstract as Action is abstract and it doesn't make sense to have it non-abstract
- Updated wrong version stated in changelog

## [1.0.17]  2020-07-11
### Added
- Added scenario to verify that path always return cheapest path

### Removed
- Removed AgentCheapest
- Removed cheapest path calculation as Path calculation was already doing that
```

Ispis 33: Dio datoteke za promjene

Struktura datoteke za promjene bazirana je na primjeru s `keepachangelog.com` web stranice. Verzija koja je korištena je 1.0.0. [16]

3. 4. Verifikacija

Ni jedna igra koja implementira GOAP paket neće biti dovoljna da pokrije sve moguće scenarije ovog paketa i da potvrdno odgovori na pitanje radi li paket u svim scenarijima. Postavlja se pitanje kako biti siguran da implementiran sustav radi na očekivan način i na koji način jasno dokumentirati korisniku kakvo je ponašanje očekivano i zamišljeno da tako funkcionira, a kakvo je ponašanje potencijalna greška u kodu koju je potrebno prijaviti i ispraviti. Jedini način da bi se uvjerilo u ispravnost jest pisanje testova.

UTF (*Unity test framework*) omogućava pisanje i pokretanje testova u e*dit* i *play* načinu rada. Testovi se mogu pokrenuti i na platformi za koju se razvija, kao što su Android, Windows, Play Station i slično. U pozadini UTF koristi NUnit dinamičku biblioteku, što je vidljivo na slici 22. [14] U daljnjem tekstu ukratko je objašnjena razlika između e*dit* i *play* načina testiranja u Unity razvojnom okruženju.

Edit način testiranja omogućava pokretanje testova samo u *editoru*, ali omogućava korištenje i testiranje *editor* klasa, poput `EditorWindowGoalPath` klase. *Edit* testovi pokreću se kao nezavisne metode jedna iza druge.

Play način testiranja omogućava pokretanje u *editoru,* ali i na odabranoj platformi. Test se pokreće kao `coroutine` (to je metoda koja može prekinuti svoje izvršavanje te ga nastaviti poslije određenog vremena ili događaja). Svaki *play* test je posebna *coroutinea*. Tako se može simulirati prolazak vremena u testu. Ako test ne koristi te značajke, tj. ako test ne ovisi o vremenu, preporučeno je koristiti *edit* način testiranja.

U daljnjem tekstu u tablicama su opisani testovi za pojedine klase. Uz naziv testa prikazan je i kratak opis koji objašnjava što je testirano u testu. Prvo su prikazani svi *edit* testovi poredani po klasama. Testovi za pomoćnu klasu `Planner.Node` nalaze se unutar testova za `Planner`.

Prikaz *edit* testova raspoređen je po sljedećim tablicama:

- Tablica 4 – testovi klase `Condition`
- Tablica 5 – testovi klase `Action`
- Tablica 6 – testovi klase `Goal`
- Tablica 7 – testovi klase `PriorityQueue`
- Tablica 8 – testovi klase `Planner`

Tablica 4: *Edit* testovi `Condition` klase

IME TESTA	OPIS
`SameConditionsAreEqual`	Isti su objekti klase jednaki.
`DifferentConditions AreNotEqual`	Dva različita objekta klase nisu jednaka, neovisno o tome zovu li se isto.
`NameCoresponds`	Dodijeljeno ime u konstruktoru klase jednako je `Name` svojstvu objekta klase.

Tablica 5: *Edit* testovi `Action` klase

IME TESTA	OPIS
`NameIsClassName`	Kreirani objekt ima svojstvo `Name` koje nije `null` pokazivač ili prazno te odgovara imenu klase.
`DefaultAction EmptyPreconditons`	Kreirani objekt ima praznu listu preduvjeta.
`DefaultActionEmptyEffects`	Kreirani objekt ima praznu listu efekata.
`DefaultActionCost DefinedTo1`	Cijena je novokreirane akcije 1.
`DefaultActionNotCompleted`	Akcija pri kreiranju markirana je kao nedovršena.
`ActionCompleted After3Transitions`	Nakon poziva metode `Transit` tri puta, akcija je markirana kao dovršena.
`AbortThrowsIfNotSupported`	Ako se `Abort` pozove bez postavljanja svojstva `Abortable` u *`true`*, metoda `Abort` podići će iznimku.
`Abort DoesNotThrowIfSupported`	Ako se `Abort` pozove s postavljanim svojstvom `Abortable` u *`true`*, metoda `Abort` ne podiže iznimku.
`StateResetToDefaultState`	Pozivom metode `ResetState`, bez obzira u kojem se stanju FSM-a nalazi, resetira akciju na početno stanje.

StateResetOnlyState	Reset metoda resetira FSM i Completed svojstvo.
ActionInfo CorrespondsToAction	IActionInfo sučelje odgovara elementima akcije i promjenom akcije mijenjaju se i podaci na sučelju.

Tablica 6: *Edit* testovi Goal klase

IME TESTA	OPIS
DefaultUrgeDefinedTo1	Kreirani objekt ima baznu želju postavljenu na 1.
DefaultUrgeMultiplier DefinedTo1	Kreirani objekt ima množitelj bazne želje postavljen na 1.
DefaultOutcomeEmpty	Prilikom kreiranja cilja, nema definiranih efekata.
NameIsClassName	Kreirani objekt ima svojstvo Name koje nije null pokazivač ili prazno te odgovara imenu klase.
UrgeValid WhenBaseUrgeSerialized	Testira *serializaciju* bazne želje.
UrgeValid WhenBaseUrgeInherited	Testira postavljanje bazne želje prilikom nasljeđivanja.
UrgeMultiplierValid	Ukupna želja cilja odgovara baznoj želji i množitelju iste.
UrgeCombinations	Ukupna želja uvijek odgovara umnošku različite vrijednosti bazne želje i množitelju iste kroz različite kombinacije.
EffectOutcomeEqual	Dodani je efekt ispravno dohvaćen kroz Outcome svojstvo.
EffectsAreUnique	Dodavanje više istih efekata ne utječe na Outcome svojstvo. Svaki je efekt u Outcome svojstvu jedinstven.
OutcomeChange WithEffectsChangePositive	Dodavanje više efekata ispravno se reflektira na Outcome svojstvo.

`OutcomeChange WithEffectsChangeNegative`	Dodavanje pa brisanje više efekata ispravno se reflektira na `Outcome` svojstvo.
`OutcomeNotChanging IfCountRemainsSame`	`Outcome` svojstvo koristi lijenu inicijalizaciju. Ako se doda X elemenata, te se potom pristupi svojstvu `Outcome`, on će biti inicijaliziran. Ako se potom izbriše jedan efekt i doda drugi, isti se neće odraziti na `Outcome` svojstvo jer radi na principu broja elemenata.
`IgoalInfo CorrespondsToGoal`	`IGoalInfo` sučelje odgovara elementima cilja. Promjenom cilja mijenjaju se i podaci na sučelju.

Tablica 7: *Edit* testovi `PriorityQueue` klase

IME TESTA	OPIS
`DefaultEmpty`	Novokreirani je objekt prazan, vrijednost je `Count` svojstva 0, a svojstva `Keys` i `Values` nemaju elemenata.
`EnqueueNewKeyUpdatesCount`	Dodavanjem novog ključa uvećava se `Count` svojstvo.
`KeyNotFound`	`TryGetValue` i `ContainsKey` vraćaju *`false`* ako ključ ne postoji.
`ThisGetInvalidKeyThrows`	Pri pokušaju pristupanja ključu koji ne postoji s `this[ključ]`, podiže se iznimka `KeyNotFoundException`.
`GetValueAndThis ReturnsFirstKeyElement`	Pristupanjem nekom ključu s `this[ključ]`, uvijek vraća prvi element tog ključa, tj. element koji je prvi dodan u red tog prioriteta. Na isti način radi i `TryGetValue`.
`CountUpdates AccordingToElementNumber`	Dodavanjem novih elemenata, s istim ili različitim ključem, uvijek uvećava broj elemenata za 1.
`KeysReceivedInOrder`	Svojstvo `Keys` uvijek vraća ključeve poredane po prioritetnom redu ključa.

ValuesReceivedInOrder	Svojstvo Values uvijek vraća vrijednosti poredane po prioritetnom redu ključa i redoslijedu dodavanja za vrijednosti koje dijele ključ, tj. prioritet.
PeekOnEmtpy	Metoda Peek na prazan prioritetni red vraća null pokazivač i za ključ i za vrijednost.
PeekDoesNotRemoveElements	Metoda Peek ne miče element iz reda.
PeekShowsLowestKeyElement	Peek uvijek pokazuje najprioritetniju vrijednost u prioritetnom redu.
PeekShowsLowestKey ElementInOrderOfInsertion	Peek uvijek pokazuje onu vrijednost koja je prva dodana ako više vrijednosti dijele isti ključ.
EnqueueDequeueOneElement	Testira ispravnost dodavanja i micanja elementa iz prioritetnog reda.
EnqueueDequeueOneKey MultipleElements	Testira ispravnost dodavanja i micanja više elementa istog ključa iz prioritetnog reda.
EnqueueDequeue MultipleKeyMultipleElements	Testira ispravnost dodavanja i micanja elementa iz prioritetnog reda niza elemenata s istim ili različitim ključevima.
EnumerateMultiple KeyMultipleElements	Testira korištenja foreach petlje koja u pozadini koristi Enumerator s više elemenata različitih i istih ključeva.
GetEnumberatorOnEmpty	Testiran ispravnost Enumeratora na prazni prioritetni red.
GetEnumerator MultipleKeyMultipleElements	Testira korištenja Enumeratora izravno s više elemenata različitih i istih ključeva.

Tablica 8: *Edit* testovi Planner klase

IME TESTA	OPIS
NodeDefaultParentNull	Korištenje konstruktora samo s akcijom, rezultirat će postavljanjem Parent svojstva u null pokazivač.

NodeDefaultActionInserted	Akcija čvora, objekta klase Node jednaka je onoj proslijeđenoj u konstruktoru.
NodeDefaultIncompleteConditions0	Prazna akcija nema preduvjeta pa je broj elemenata svojstva IncompleteConditions 0.
NodeDefaultDepth1	Pri kreiranju čvora bez roditelja dubina tog čvora je 1.
NodeDepthStackByParents	Dodavanje čvora s roditeljem znači da je dubina novokreiranog čvora jednaka dubini roditelja uvećana za 1.
IncompleteConditions AreActionPreconditions	Pri kreiranju čvora s akcijom koja ima preduvjete svojstva IncompleteConditions, preuzima te preduvjete na sebe.
IncompleteConditions UniqueAcrossNodeList	Ako se kreira više čvorova s istom akcijom koja ima X preduvjet, s tim da su čvorovi povezani u roditeljskoj vezi, tada svojstvo IncompleteConditions zadnje kreiranog čvora sadržava samo jedan X preduvjet. Preduvjeti IncompleteConditions jedinstveni su kroz cijeli povezani čvor.
ActionEffect NullfiyEqualPrecondition	Čvor čija akcija ima iste preduvjete i efekte, poništava se tako da IncompleteConditions ne sadržava nijedan element.
ChildEffectNullfiy ParentEqualPrecondition	Ako čvor s akcijom efekta X, i roditelj tog čvora ima preduvjet X, tada čvor dijete poništava preduvjet roditelja.

`GrandchildEffect NullfiyParentEqualPrecondition`	Ako čvor s akcijom efekta X, i roditelj roditelja tog čvora ima preduvjet X, tada čvor dijete poništava preduvjet roditelja od roditelja.
`ParentEffectDoesNot NullfiyChildPrecondition`	Ako roditelj ima akciju s nekim efektom X, a dijete čvor ima preduvjet X, tada roditelj ne poništava preduvjet djeteta i `IncompleteConditions` sadržava preduvjet X.
`NodeCollects MultipleActionPreconditions`	Ako je više čvorova 1–N povezano kao dijete–roditelj i svaki od njih ima različiti preduvjet Xn, tada svojstvo `IncompleteConditions` zadnjeg čvora djeteta sadržava ta sve preduvjete roditelja 1Xn–NXn.
`ActionQueueOrderChildToParent`	`ToActionQueue` metoda vraća niz akcija, od akcije zadnjeg čvora djeteta do prvog čvora roditelja.
`PlanDefaultArguments`	`Planner` metoda `Plan` prima tri obavezna argumenta. Testiraju se različite kombinacije kad se pošalje `default` argument za jedan argument, sve argumente ili bilo koju kombinaciju između njih.
`AllPathsDefaultArguments`	`Planner` metoda `GetAllAchievablePaths` prima tri obavezna argumenta. Testiraju se različite kombinacije kad se pošalje `default` argument za jedan argument, sve argumente ili bilo koju kombinaciju između njih.

`UnableToMeetGoal`	Testira da, ako cilj nije moguće isplanirati, metoda Plan vraća `null` pokazivač, a `GetAllAchievablePaths` vraća praznu listu puteva.
`GoalIsAction`	Testira najjednostavniji slučaj planiranja, a to je da je sama akcija cilj. Testiraju se `Path` i `GetAllAchievablePaths` metode.
`ActionQueue DoesNotPickUnnecessaryActions`	Testira da obje metode `Plannera` neće prilikom pronalaženja puta uvrstiti akcije koje nisu potrebne.
`GoalIsCheapestAction`	Jednostavan scenarij kad je jedna akcija dovoljna za ostvarivanje cilja. Testira se da je odabrana akcija za ostvarivanje cilja najjeftinija u metodi `Planner.Plan`.
`GoalAlreadyAchieved`	Ako je cilj prazan, `Planner.Plan` vraća praznu listu akcija, a `Planner.GetAllAchievablePaths` vraća prazan niz.
`GoalAchieved AllPathsListedCheapestFirst`	Jednostavan scenarij kad je jedna akcija dovoljna za ostvarivanje cilja. Testira se da metoda `Planner.GetAllAchievablePaths` vrati sve putanje do cilja, poredane od najjeftinijeg do najskupljeg puta.
`GoalAchieved TroughAchievedConditions`	Ako cilj ima jedan efekt i ako se metodi `Planner.Plan` kao parametar `achievedConditions` proslijedi taj efekt, tada metoda vraća prazan niz.

`GoalAchieved TroughAchievedConditions AllPaths`	Ako cilj ima jedan efekt i ako se metodi `Planner.GetAllAchievablePaths` kao parametar `achievedConditions` proslijedi taj efekt, tada metoda vraća praznu listu.
`MultipleActionsToAchieveGoal`	Testira se jednostavan slučaj u kojem je potrebno povezati više akcija kako bi se došlo do cilja. `Plan` i `GetAllAchievablePaths` testirane su uz provjeru redoslijeda akcija.
`MultipleActionsToAchieveGoal WithAchievedConditions AtBeginning`	Testira se jednostavan slučaj u kojem je potrebno povezati više akcija kako bi se došlo do cilja. `Plan` i `GetAllAchievablePaths` testirane su uz provjeru redoslijeda akcija. Razlika je od gornjeg testa u tome što neke preduvjete prosljeđuju kroz trenutno ostvarena stanja, tj. kroz parametar `achievedConditions`. Proslijeđeno stanje preduvjet je prve akcije.
`MultipleActionsToAchieveGoal WithAchievedConditions Middle`	Isti test kao gore. Razlika je da se akcija, čije preduvjete nije moguće zadovoljiti drugom akcijom, već proslijeđenim stanjem, nalazi u sredini plana.
`MultipleActionsToAchieveGoal WithAchievedConditions MultiMiddle`	Testira proslijeđena stanja kad više akcija treba proslijeđena stanja.

`SearchDepth CorrespondsActionCount`	Broj akcija koje se dobiju isplaniranim putem jednak je dubini, tj. ako su za doći do cilja potrebne tri akcije, tada `Plan` i `GetAllAchievablePaths` dolaze do tog puta s postavljenom maksimalnom dubinom tri.
`GoalUnreachableByDepth`	Za prethodni slučaj postavljena je dubina 2, algoritam obje metode nije pronašao put.
`MultipleMulti ActionsPathsToAchieveGoal`	Više puteva kroz različite akcije vode do cilja. Testira da `Plan` vraća najjeftiniji put i da su akcije ispravno poredane. `GetAllAchievablePaths` vraća sve puteve redom od najjeftinijeg do najskupljeg. Provjerava i da su sve akcije u svakom putu ispravno poredane.
`MultiConditionGoalAndAction`	Efekti cilja i efekti akcije sastoje se od više efekata. Testira uspješan pronalazak puta s obje metode.
`MultiConditionGoalBranchAction`	Efekti cilja sastoje se od više efekata. Nijedna akcija ne može u potpunosti zadovoljiti cilj. Potrebno je izabrati više akcija kako bi se cilj u potpunosti ostvario. Testiraju se obje metode.
`MultiConditionGoal HalfActionHalfWorld`	Efekti cilja sastoje se od više efekata. Nijedna akcija ne može u potpunosti zadovoljiti cilj. Testira uspješan pronalazak puta s proslijeđenim `achievedConditions` stanjima

	koja nedostaju kako bi se potpuno zadovoljio cilj. Testiraju se obje metode.
`MultiAction PreconditionsBranch`	Akcija u sredini plana ima više preduvjeta koje u potpunosti ne mogu biti zadovoljene s jednom akcijom. Testira da se plan uspješno isplanira. Plan je na dijelu puta razgranat na dva puta te se do kraja spaja u jednu putanju. U ovakvim slučajevima redoslijed puta ne može biti provjeren jer nije bitno koja se putanja do prvog preduvjeta izvrši prva, bitno je samo da, prije izvršavanja akcije s više preduvjeta, svi njezini preduvjeti moraju biti zadovoljeni.
`TestFastSlowPlanner DifferentPath`	Testira situaciju kad više akcija ima potpuno isti efekt. Obje su metode testirane, kao i redoslijed akcija.

Ispis 34 prikazuje jednostavan prikaz *edit* testa. Kod *edit* tipa testa koristi se atribut `Test` iznad imena metode. U prvoj liniji kreira se novi objekt i dodaje se `TestAction` komponenta na taj objekt. Moguće je direktno kreirati objekt putem konstruktora, ali to nije preporučeno ako klasa nasljeđuje od `MonoBehaviour` klase. Kreiranje objekta konstruktorom izbacit će upozorenje. `TestAction` je samo pomoćna metoda (engl. *wrapper*) za `Action` klasu s obzirom na to da je ista apstraktna. U testu je postavljeno svojstvo `Abortable` na *`true`* i testira se da pozivanje metode `Abort` neće prouzročiti iznimku. `Assert` klasa dio je NUnit-a i ima mnogo metoda za provjeravanje stanja objekta koje su korištene prilikom testiranja. Neki su testovi kratki i jednostavni kao ovaj, dok su drugi duži jer se provjerava sekvenca akcija i slično. Ispis 35 prikazuje primjer *play* testa.

```
[Test]
0 references | 0 changes | 0 authors, 0 changes
public void AbortDoesNotThrowIfSupported()
{
    var action = new GameObject().AddComponent<TestAction>();

    action.Abortable = true;

    Assert.DoesNotThrow(() => { action.Abort(); });
}
```

Ispis 34: Jednostavan primjer *edit* testa

```
[UnityTest]
0 references | 0 changes | 0 authors, 0 changes
public IEnumerator AbortCurrentGoalThresholdChange()
{
    var go = new GameObject();
    var agent = go.AddComponent<TestAgent>();

    var minAbortDifference = 0.000001f;

    var conditionX = new Condition("x");
    var conditionZ = new Condition("z");

    var actionX = go.AddComponent<ActionTest.TestAction>();
    actionX.Effects.Add(conditionX);

    var actionZ = go.AddComponent<ActionTest.TestAction>();
    actionZ.Effects.Add(conditionZ);
    actionZ.Abortable = true;

    var goalX = go.AddComponent<GoalTest.TestGoal>();
    goalX.SetUrge(1f);
    goalX.Effects.Add(conditionX);

    var goalZ = go.AddComponent<GoalTest.TestGoal>();
    goalZ.SetUrge(2f);
    goalZ.Effects.Add(conditionZ);

    agent.AbortableInit(1 + minAbortDifference, MinInvokeTimer);

    for (int i = 2; i < 3; i++)
    {
        agent.SetGoalUrgeMultiplier(goalX, i);

        yield return new WaitForSeconds(MinWaitInvokeTime);

        Assert.IsTrue(agent.HasPlan);
        Assert.AreEqual(goalZ, agent.ActiveGoal);
        Assert.AreEqual(actionZ, agent.ActiveAction);
    }

    agent.SetGoalUrgeMultiplier(goalX, 3f + minAbortDifference);

    yield return new WaitForSeconds(MinWaitInvokeTime);

    Assert.AreEqual(goalX, agent.ActiveGoal);
    Assert.AreEqual(actionX, agent.ActiveAction);

    agent.Execute(ActionArgs.Empty);

    Assert.AreEqual(goalX, agent.ActiveGoal);
    Assert.AreEqual(actionX, agent.ActiveAction);
}
```

Ispis 35: Primjer *play* testa

U *play testu* testira se hoće li se cilj promijeniti kad je akciju moguće prekinuti i kad ukupna želja nekog drugog cilja bude veća od postavljenog limita za prekid. Kao i kod prethodnog testa prvo se dodaju komponente, zatim postavlja minimalna razlika koja je potrebna za prekid te je time je prikazana preciznost `Abort` sustava. Nakon toga definirana su dva stanja X i Z, i dvije akcije. Akcija X ima efekt stanje X, a akcija Z ima efekt stanja Z. Također dodaju se dva različita cilja, X i Z. Želja za ostvarivanje cilja Z je 2, dok je želja za ostvarivanje cilja X 1. Nakon toga poziva se `AboratbleInit` metoda kojoj su proslijeđena dva parametra. Prvi parametar govori na kojoj će se razlici između želje dvaju ciljeva prvi pokušati prekinuti. Ta je vrijednost postavljena na 1 uvećana za preciznost `Abort` sustava koji je 0.000001. Drugi je argument svako koliko će se provjeravati je li se uvjet za prekid zadovoljio. On je postavljen na 0.00002. Nakon toga povećava se ukupna želja cilja X na 2, pa zatim na 3 i svaki se put čeka minimalno vrijeme 0.00003. Ovo je ujedno i glavna razlika *play* tipa testa.

Svaka je metoda definirana kao `coroutine metoda` i može se čekati proizvoljno vrijeme te se poslije vratiti izvršavati ostatak metode. Kod ovog tipa testa potrebno je staviti atribut `UnityTest` iznad imena `coroutine` metode. Kad je ukupna želja cilja X jednaka 2, tada je on izjednačen sa željom cilja Z i trenutni se cilj i akcija ne mijenjaju. Sljedeća iteracija razlika želja cilja X i Z je 1, što nije dovoljno da se trenutni cilj prekine kako je granica postavljena na razliku od 1.000001. Sljedeći korak izjednačava razliku želje na postavljenu granicu i čeka se minimalno vrijeme. Poslije toga se u testu očekuje da su se trenutni cilj i akcija promijenili. Za kraj se pokreće jedna akcija i očekuje da je trenutni cilj X, s trenutno aktivnom akcijom X. Tablica 9 prikazuje *edit* i *play* testove za klasu `Agent`. Klasa `Agent` jedina je klasa koja ima potrebu za *play* tipom testova.

Tablica 9: Edit i Play testovi `Agent` klase

IME TESTA	OPIS
`DefaultAgent PublicEmptyOrDefault`	Testira da pristupanje javnim svojstvima poslije kreiranja Agent klase radi bez podizanja iznimke i da su sve vrijednosti ispravne.

`AgentParseGoalsFromGoOnInit`	Testira da prilikom poziva `Init` metode agent doda sve ciljeve koje su dodane na isti Unity objekt kao i agent.
`AgentParseActionsFromGoOnInit`	Testira da prilikom poziva `Init` metode agent doda sve akcije koje su dodane na isti Unity objekt kao i agent.
`GoalUrgeMultiplierResetOnInit`	Pri pozivu `Init` metode svi množitelji bazne želje resetiraju se na 1. Razlog tome je što je množitelj bazne želje zamišljen kao nešto što će se dinamički mijenjati ovisno o stanju agenta, a bazna je želja nešto čime se želi izraziti prioritet cilja u standardnim okolnostima.
`AddCondition`	Provjerava funkcioniranja dodavanja novog stanja agentu.
`RemoveCondition`	Provjerava funkcioniranja dodavanja i micanja novog stanja agentu.
`RemoveNonExistingCondition`	Provjerava funkcioniranja micanja stanja koji trenutno ne postoji na agentu.
`AgentGoalsOrderByHighestUrge`	Testira da su ciljevi agenta uvijek poredani od onoga s najvećom do onoga s najmanjom željom.
`ExecuteOnInitDoesNotThrow`	Pozivanjem `Execute` metode prije `Init` metode, tj. odmah nakon kreiranja agent objekta, neće prouzročiti iznimku.
`ExecuteStartValidGoal`	Provjerava planiranje i pokretanje izvršavanja prve akcije odabranog cilja.
`ExecuteStart ValidGoalMultiAction`	Provjerava planiranje i pokretanje izvršavanja prve akcije odabranog cilja kod cilja s više akcija. Provjerava da su ostale akcije u ispravnom redoslijedu u nizu akcija za izvršavanje.

`PickMostUrgentGoal`	Agent s više ciljeva odabire i planira onaj koji ima najveću želju.
`PickMostUrgentAchievableGoal`	Agent s više ciljeva odabire i planira onaj koji ima najveću želju. Ako se cilj s najvećom željom ne može isplanirati, bit će odabran cilj s najvećom željom koji se može isplanirati.
`ThreeStatesToCompleteAction`	Za izvršiti cilj s jednom akcijom potrebna su tri poziva `Execute` metode kako bi se prešla sva FSM stanja akcije. Nakon toga provjerava se da agent nema nikakav plan na sebi.
`AfterActionCompletes HandlesNextActionInChain`	Provjerava isto što i gornji test, samo što se cilj sastoji od više akcija. Za svaku od akcija očekuje se isto ponašanje.
`GoalFinishedAndRemoved HandleNextMostUrgentGoal`	Provjerava se da se, nakon što se akcije izvrše, pozivom `Execute` metode isti cilj uklanja kao izvršen. Ponovnim pozivom te metode agent shvaća da je moguće izvršiti sljedeći cilj s najvećom željom, uspješno se isplanira i počinje izvršavati.
`AgentClear`	Testira da nakon `Inita` agent počne izvršavati cilj. Ako se pozove metoda `Clear`, isti će se prestati izvršavati, tj. agent više neće sadržavati trenutni cilj, trenutnu akciju, stanje, akcije i ciljeve.
`AgentClearBeforeInit DoesNotRemove GoalsAndActionsOnGo`	Ako se pozove `Clear` prije poziva `Inita`, i dalje će se očistiti sve što se dodalo, ali prilikom `Inita` dodaju se ciljevi i akcije s Unity objekta, stoga će isti funkcionirati kao da `Clear` nije pozvan. Nije

	preporučljivo dodavanje ciljeva, akcija ni stanja agenta prije `Inita`.
`AgentClearActiveGoal`	Provjerava da metoda `Clear` čisti trenutni cilj i akciju.
`AgentAddGoalFromDifferentGo`	Agent uspješno planira cilj ako je isti dodan koristeći `AddGoal` metodu poslije `Inita`.
`RemoveGoal`	Testira micanje cilja koji postoji na agentu.
`RemoveNonExistingGoal`	Testira micanje cilja koji ne postoji na agentu.
`RemoveActiveGoal`	Micanjem cilja koji je aktivan, neće prekinuti izvršavanje istog ako je isti isplaniran i u izvršavanju prije micanja.
`SetUrgeMutli BeforeInitNoEffect`	Postavljanje množitelja bazne želje prije `Init` metode nema efekta s obzirom na to da se isti resetira u 1 prilikom `Inita`.
`SetUrgeMutliAfterInit`	Postavljanje množitelja bazne želje nakon `Init` metode utječe na to koji će se cilj isplanirati.
`AbortCurrentGoal`	Testira metodu `AboratbleInit` i prekid trenutnog cilja te planiranje idućeg na osnovi razlike bazne želje.
`AbortValid WithAddedCondition`	Testira metodu `AboratbleInit` i prekid trenutnog cilja te planiranje idućeg na osnovu razlike bazne želje. Sljedeći cilj s većom baznom željom nije moguće isplanirati te se trenutni cilj ne prekida. Jednom kad sljedeći cilj bude moguć, u ovom slučaju dodavanjem stanja agentu, stari će se cilj prestati izvršavati, a novi će se cilj isplanirati i početi izvršavati.
`ActiveGoalCannotAbortItself`	Testira da mijenjanjem množitelja bazne želje aktivnog cilja, koji prelazi

	`abortThreshold AboratbleInit` metode, ne prekida trenutno aktivni cilj.
`ActionNotSetAsAbortable`	Ako svojstvo `Abortable` trenutne akcije koja se izvršava nije postavljeno u `true`, trenutni cilj neće se moći prekinuti. Nije važno jesu li ostali uvjeti ispunjeni.
`AbortCurrentGoal ThresholdChange`	U slučaju minimalne razlike između dvaju ciljeva od 0.000001, trenutni će se cilj i dalje prekinuti ako su ostali uvjeti zadovoljeni.
`AbortedGoal ExecutededAfterPrirotiy`	Nakon prekida trenutnog cilja koji se izvršava, izvršavat će se onaj s većom željom, tj. prioritetom. Jednom kad on završi i ako je po želji sljedeći na redu izvršavanja cilj koji je bio prekinut, izvršavat će se on. Cilj koji je prekinut ne uklanja se s agenta, dok se cilj koji je završen uklanja.
`AbortFailedIfNoOtherGoal`	Ako nema drugih ciljeva, prekid trenutnog cilja nije moguće pokrenuti.
`AbortFailed IfNoOtherGoalPossible`	Ako nema nijednog cilja koji se može isplanirati, prekid trenutnog cilja nije moguće izvršiti.
`AbortFailedIf NoGoalWithHigherUrgePossible`	Ako nema nijednog cilja koji zadovoljava uvjet razlike želja izvršavanja, prekid trenutnog cilja nije moguće izvršiti.
`AbortCurrentGoal OnAbortableActionOnly`	Cilj se prekida samo na akciji koju je moguće prekinuti. Ako je uvjet razlike želja zadovoljen, a akciju nije moguće prekinuti, ista će biti dovršena do kraja. Ako se sljedeća akcija može prekinuti i još je uvijek

	uvjet zadovoljen, trenutni će cilj biti prekinut.

4. Igra „The Dream“

U ovom poglavlju opisuje se implementacija prototipa igre. Važno je napomenuti da se ovdje ne može govoriti o potpuno razvijenoj igri, već o prvoj verziji igre u alfa fazi. Alfa faza nudi verziju igre koja je u mnogočemu nedovršena i nepolirana te mnoge funkcionalnosti koje bi trebale biti implementirane u krajnjoj verziji igre nedostaju. Cilj je alfa verzije dobiti povratne informacije (engl. *feedback*) od manje skupine ljudi kojoj se omogućuje isprobavanje prototipa igre.

U prošlosti su se igre značajno drugačije razvijale te su u svom opsegu bile dosta jednostavnije. Također su i očekivanja igrača bila manja. Danas su očekivanja prosječnog igrača značajno narasla i slobodno se može reći da su velika. Povratne informacije igrača trebale bi odgovoriti na nekoliko važnih pitanja. Prva bi pitanja bila je li igra zanimljiva i ima li potencijala postati zanimljiva, ako u trenutnoj fazi nije. Sasvim je moguće, i čest je slučaj, da nešto izgleda zanimljivo na papiru, međutim, jednom kad se implementira, to ne bude tako. Razvojni inženjeri i osobe koje rade na igri ne mogu objektivno ocijeniti svoj rad, i ponekad se, pogotovo ako igru rade iz strasti i iz osobne ideje, a ne nakon istraživanja tržišta, može dogoditi da precjenjuju istu. Iz tog razloga prvo ispitivanje treba dati odgovor na pitanje smatraju li drugi da je igra zanimljiva ili vide li barem potencijal u istoj ako bi se nešto promijenilo, nadogradilo i slično. Ako igrači ne vide igru zanimljivom, trebalo bi se saznati zašto je tome tako i što se može učiniti da se projekt spasi. Nakon toga, uz naučena saznanja i izmjene, ponavlja se ciklus. Sljedeće pitanje na koje treba odgovoriti jest koliko je igra intuitivna. Ovo je možda najteže pitanje za razvojne inženjere. Razlog tome je što su oni ti koji su igru implementirali, stoga je sasvim logično da stvari koje su implementirali i s kojima se susreću svakodnevno vide kao intuitivne. Kod ove faze trebalo bi promatrati igrače kako igraju i obratiti pozornost što i kako rade u igri. Ako mnogi rade istu stvar, trebalo bi razmisliti je li to možda intuitivniji način te treba li i, ako da, što promijeniti kako bi ista bila intuitivnija igračima. Također bi trebalo analizirati rade li igrači nešto što nije predviđeno i kako to utječe na igrivost, mogu li igrači uraditi nešto što nije poželjno, mogu li doći negdje gdje nisu očekivani i slično. Na kraju bi trebalo saslušati prijedloge i savjete o tome kako oni vide da bi se igra trebala dalje razvijati, što bi željeli vidjeti implementirano drugačije, što izmijeniti, što dodati. Srećom, danas je relativno lako dobiti povratnu informaciju za prototip igre i savjete za istu, no pritom treba paziti da nisu svi savjeti jednaki.

Razvojni su inženjeri ti koji trebaju analizirati povratne informacije i napraviti plan za sljedeću iteraciju igre, a možda i plan za buduće iteracije.

Iako je cilj ovom igrom prikazati i kako funkcionira implementirani i detaljno opisan Unity paket GOAP, on nije glavni cilj prototipa. Razlog tome je što, bez obzira na to kakav AI sustav imali u igri, ako osnovna igriva petlja (engl. *gameplay loop*) nije zanimljiva, nitko se neće zadržati dovoljno dugo da bi ga primijetio. Stoga je, uz implementaciju AI sustava koristeći GOAP sustav, naglasak stavljen i na igrivu petlju.

Kod implementacije ove igre bio sam inspiriran tipom igre kao što je Overcooked u kojoj je potrebna brza reakcija i planiranje kako bi se ispunio cilj. U sljedećim potpoglavljima opisana je osnovna igriva petlja igre, na koji je način implementirana i na koji se način može jednostavno proširiti s dodatnim razinama (engl. *levels*). Igra je implementirana za PC i Android platformu te se može igrati s *gamepad* kontrolerom, tipkovnicom ili uz pomoć ekrana na dodir.

4. 1. Osnovna igriva petlja

Ova igra zamišljena je da se igra po kampanjama. Svaka kampanja je neovisna i sastoji se od nekoliko razina koje su međusobno ovisne. Zamišljena je jednostavna priča za igru u kojoj bi glavni protagonist imao san jednog dana otvoriti svoj restoran i time ostvariti svoj san.

Prva razina prve kampanje jest da protagonist, još kao dijete, kreće u ostvarivanje svojih snova tako da otvori štand za prodaju limunade ispred svoje kuće. Prije ulaska u detalje prve kampanje, treba razjasniti na koji je način zamišljeno da kampanje i razine u igri funkcioniraju. Generalno, pojedina kampanja podijeljena je u nekoliko razina. Razine jedne kampanje povezane su tako da svaka razina ima minimalnu razinu novaca koju igrač mora ostvariti na toj razini. Ako igrač ostvari tu ili veću razinu novca na toj razini, može ići na sljedeću razinu te kampanje. Ako ne ostvari minimalni zadani iznos, igrač mora ponoviti tu razinu. Sa svakom razinom dolazili bi novi recepti i ponekad nova mehanika. To posebno važi za prvu kampanju kako bi igra igrača postepeno učila mehanici igre bez da se igrač osjeća „kao da je bačen u vatru“. Svaka od razina neke kampanje nalazila bi se na istoj lokaciji, ali bi jednom kad igrač bude na razini većoj od prve, igrač imao fazu pripreme. U fazi pripreme igraču je ponuđeno da potroši sve novce koje je zaradio do te razine u toj kampanji. Igrač bi tako mogao kupiti bolje kuhinjske uređaje ili alate, ili pak kupiti dodatan

stol kako bi mogao primiti više gostiju i slično. Igrač na svakoj razini iznova troši ukupan zbroj novca u toj kampanji tako da se kupljene stvari na razinama mogu uvelike razlikovati. Zamišljeno je da kupnja igraču olakšava da dođe do minimalne razine novaca za neku težinu i da poslije bude gotovo nemoguće ili nemoguće prijeći razinu bez da je igrač potrošio barem dio novca na poboljšavanje svojih izgleda za uspjehom.

Jednom kad igrač prvi put pokrene igru, dočekuje ga prizor prikazan na slici 24 koja prikazuje igru u verziji 0.0.1. Moguća su poboljšanja ili izmjene u novijim verzijama igre.

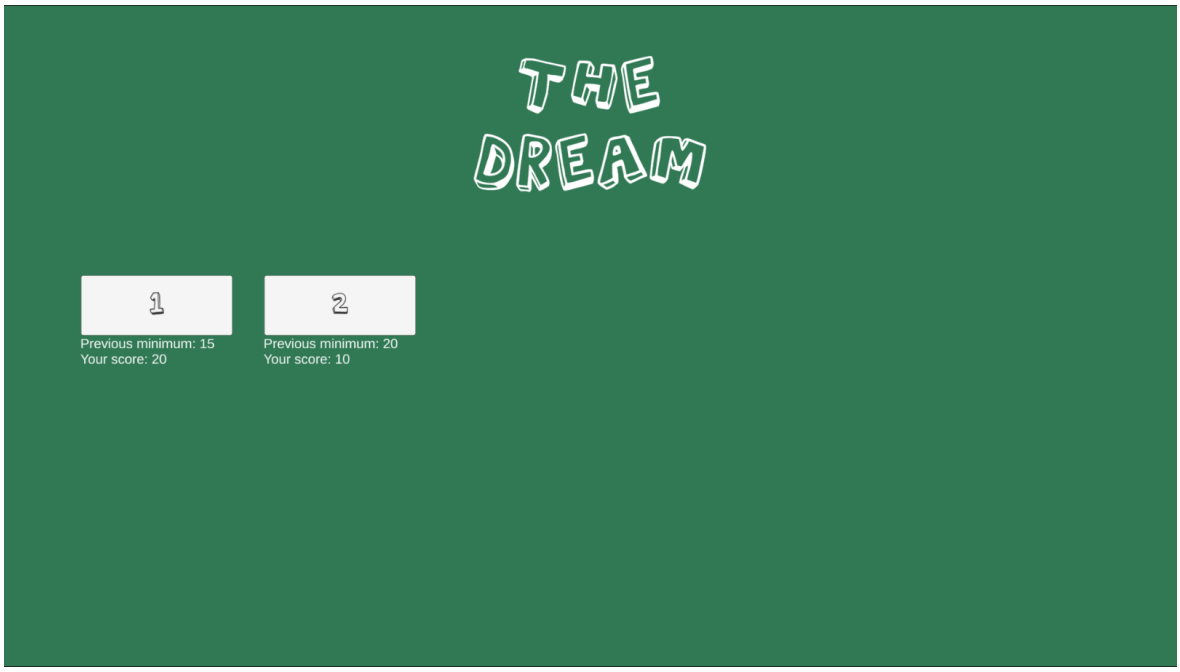
Slika 24: Glavni izbornik igre „*The dream*“ u verziji 0.0.11

Igra ima radni naziv „*The Dream*“ iz prethodno objašnjenje priče u kojoj protagonist od malih nogu sudjeluje u kulinarskim djelatnostima s ciljem da jednog dana otvori svoj vlastiti restoran. Ono što se sa slike ne može vidjeti jest da je glavni izbornik dinamičan i popraćen glazbom. Skupine oblaka pomiču se različitom brzinom na način da se ne ponavljaju iste situacije.

Zvuk, neki materijali, teksture i 3D modeli korišteni u igri skinuti su najčešće s Unity trgovine (engl. *Unity asset store*). Prilikom preuzimanja preko Unity trgovine ili nekom drugom metodom pazilo se na tip licence *asseta*. Većina skinutih 3D modela došla je iz različitih paketa te ih je bilo potrebno prilagoditi na neki način kako bi se mogli koristiti u igri. Razvoj igre i dizajniranje razina zahtjeva pomno planiranje i podešavanje mnoštva sitnih detalja kako bi se dobio zadovoljavajući efekt. Razlog je tome što se igra sastoji od

mnogo različitih komponenti koje imaju ogromnu dubinu i sve komponente moraju djelovati zajedno kao cjelina. Kako osobno nemam puno iskustva u izradi razina za igre ili podešavanja tekstura, 3D modela i slično, ovaj projekt bio mi je popriličan izazov. Prilikom izrade igre sve su skripte izrađene samostalno i nisu korištene vanjske skripte.

Kad igrač pokrene igru pritiskom na dugme Start na glavnom izborniku, pojavit će se ekran kampanja i razina koji je prikazan na slici 25.

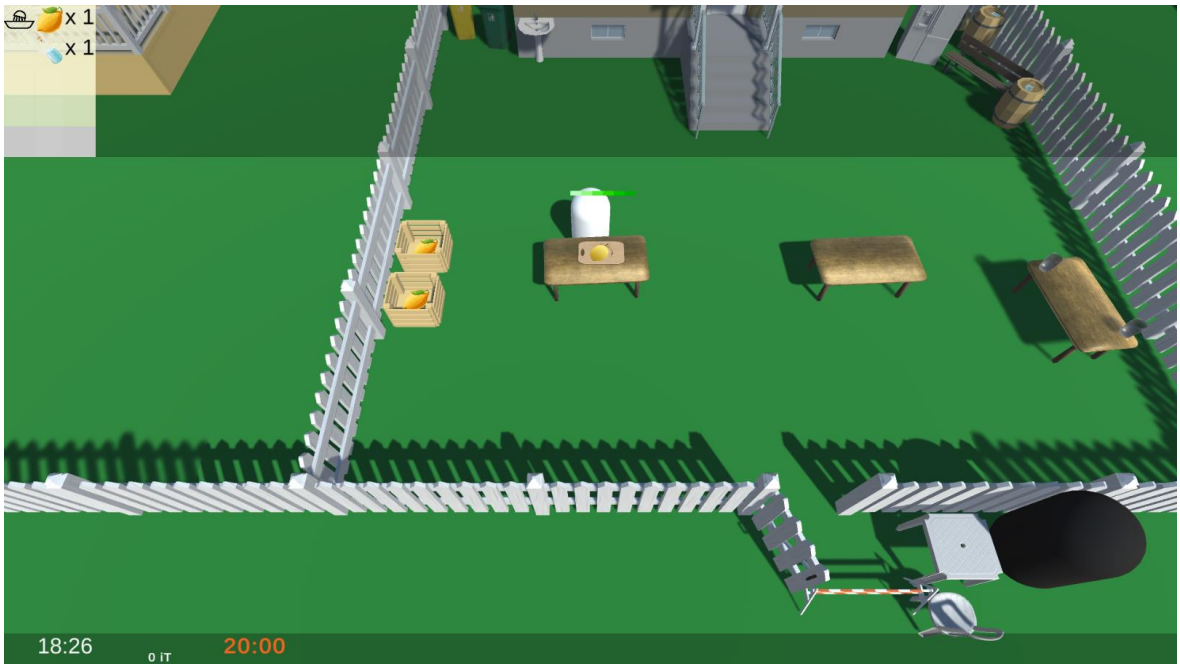


Slika 25: Odabir razina prve kampanje u verziji 0.0.11

Na slici 25 prikazan je izbornik razina u verziji 0.0.11. Trenutno su implementirane dvije razine za prvu kampanju. Ispod svake razine vidljiv je minimalni iznos koji je potrebno ostvariti kako bi se otključala sljedeću razina. U ovom slučaju obje su razine otključane jer je najveći iznos ostvaren u prvoj razini 20 iT-a, a za otključati iduću razinu potrebno je najmanje 15 iT-a. Pritiskom na razinu igrač ju započinje igrati. Na početku će igrač moći odabrati samo prvu razinu kampanje, a ostale se otključavaju postignutim uspjehom na prethodnoj razini.

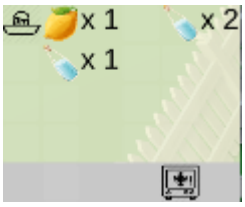
Prva razina prve kampanje zamišljena je da bude jednostavna i ima samo jedan recept, limunadu. Slika 26 prikazuje kako izgleda prva razina u verziji igre 0.0.11. Na slici je vidljiv igrač koji koristi cjedilo za limun. Poviše igrača vidljiva je traka koja se puni s vremenskom interakcijom dok se limun ne ocijedi. Trenutno se za prikaz cjedila koristi

daska jer nije pronađen nijedan *asset* cjedila s odgovarajućom licencom. Igraču je vizualno prikazano kad stavi limun u cjedilo. Promjenom 3D modela limuna, igrač, uz vizualnu traku poviše glave protagonista, vidi kad je uspješno ocijeđen limun. Prije nego što je igrač krenuo cijediti limun, prišao je klijentu za stolom i uzeo njegovu narudžbu koja se može vidjeti u gornjem kutu.



Slika 26: Prva razina prve kampanje u verziji 0.0.11

S lijeve strane narudžbe prikazano je kroz koje kućanske alate igrač mora provući pojedini sastojak. Slika prikazuje da je limun potrebno ocijediti. U istoj liniji na kraju piše koliko sastojaka treba ići u recept – vidljivo je da treba ocijediti jedan limun, a uz to je potrebna i jedna voda. Jednom kad igrač spoji ta dva sastojka, može ih odnijeti klijentu. Ako je narudžba točno napravljena, za istu će dobiti određenu svotu novca. Ovo je najjednostavniji recepata pa ne prikazuje punu moć istih.



Slika 27: Recept za limunadu s ledom

Slika 27 prikazuje malo složeniji recept za limunadu s ledom. Recept se sastoji od dvije nezavisne instrukcije. U prvoj instrukciji potrebno je ocijediti limun, a zatim dodati vodu, kao i u prethodnom receptu. U drugoj instrukciji potrebno je dodati dvije vode u posudu i staviti ih u hladnjak kako bi se dobio led. Nakon toga igrač uzima led iz hladnjaka i dodaje ga u posudu s limunadom. Takvu posudu prosljeđuje klijentu.

U prvoj razini prve kampanje osnovna igriva petlja sastoji se od:

- uzimanja narudžbi od klijenata
- uzimanja sastojaka
- po potrebi obrade na kuhinjskom alatu
- stavljanja sastojaka u jednu posudu
- dostave posude klijentu
- uzimanja novaca od klijenta ako se narudžba napravila točno
- čišćenja stola
- čišćenja posude koja se dala prethodnom klijentu.

U početnoj razini igraču su prezentirani samo kućanski alati, ali ne i kućanski strojevi kako bi se igrača postepeno uvelo u igru. U donjem dijelu slike 26 vidi se trenutno vrijeme, vrijeme prestanka rada i količina novaca koju igrač ima na toj razini. Jednom kad se vrijeme poklopi s vremenom prestanka rada, razina završava. Ovisno o ostvarenoj novčanoj dobiti, igra će igraču ponuditi odlazak na sljedeću razinu ili resetiranje razine kako bi pokušao ponovo.

Na svim sljedećim razinama igriva petlja proširuje se tako da prije početka razine nudi igraču da proširi ili poboljša opremu na toj razini za novac ostvaren u svim prethodnim razinama kampanje. To može uključivati poboljšavanje postojećih kućanskih alata i uređaja, dodavanja novih stolova, posuda i slično. Ostale promjene mehanike ovise o razini, ali mogu uključivati obradu hrane u kućanskim uređajima, popravljanje kućanskih uređaja koji su pokvareni, naručivanje namirnica i slično.

4. 2. Implementacija

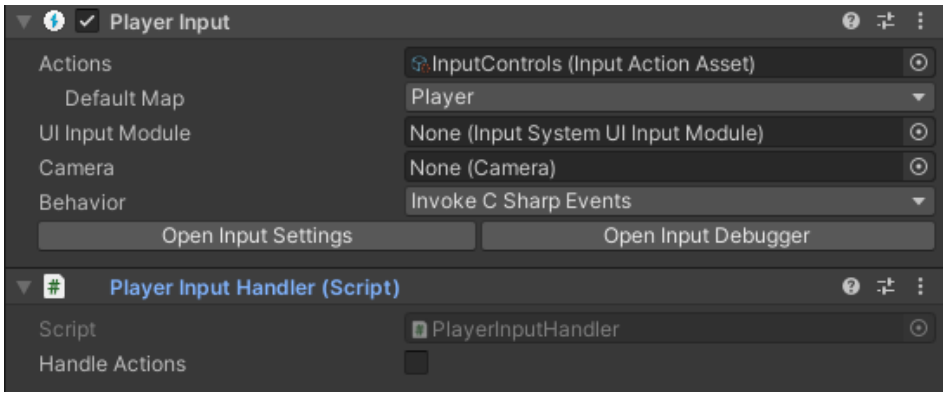
Implementacija je igre kompleksna jer je uz kod potrebno implementirati i prilagoditi zvuk, grafiku, UI, fiziku, AI i slične sustave koje zajedno trebaju činiti jednu homogenu

cjelinu. Svaki od tih sustava ima i svoja ograničenja te rubne slučajeve na koje treba paziti što kreiranje jedne homogene cjeline čini kompleksnim zadatkom. Unity implementira *component pattern* s kojim je predočen svaki objekt na Unity sceni. Svaki objekt na sceni sastoji se najmanje od lokacije u sceni, rotacije u odnosu na scenu i podataka o skaliranju objekta. Ako se želi imati mogućnost dodati neka skripta na Unity objekt, ona mora nasljeđivati od `MonoBehaviour` klase.

4. 2. 1. Konfiguriranje kontrola i igrača

Kod implementacije kontrola za igrača zamišljeno je da igrač jednostavno kontrolira protagonista, tj. da ista kontrola radi različitu stvar, ovisno o kontekstu. Uz to je cilj bio omogućiti da se igra može igrati na PC i Android platformama, neovisno igra li se tipkovnicom, ekranom na dodir ili pak nekim od *gamepad* kontrolera. Kako bi se to postiglo, koristio se novi Unity sustav za ulaz (*Input system*).

Cilj je novog sustava biti modularan te jasno odvojiti kontrole od akcija. Prvo je potrebno dodati na igrača `PlayerInput` klasu koju pruža Unity sustav za ulaz. Tu je potrebno odabrati koja će se datoteka za akcije koristiti kao i na koji će način sustav raditi. Mogući je izbor da radi preko `Send` ili `Broadcast` Unity poruka, preko Unity događaja ili preko čistih C# događaja. `Send` i `Broadcast` Unity poruke procesorski su skupe i stoga nisu preporučljive pa je opcija odabira svedena na Unity ili C# događaje. Unity događaji nešto su sporiji, ali se zato mogu slagati iz editora dok je za C# događaje potrebno nešto malo više programirati i definirati neke dodatne stvari za obradu C# događaja. Odabrani su C# događaji, a dodatna implementacija nalazi se u `PlayerInputHandler` klasi vidljivoj na slici 28. gdje su dodane na objekt igrača zajedno s `PlayerInput` klasom.

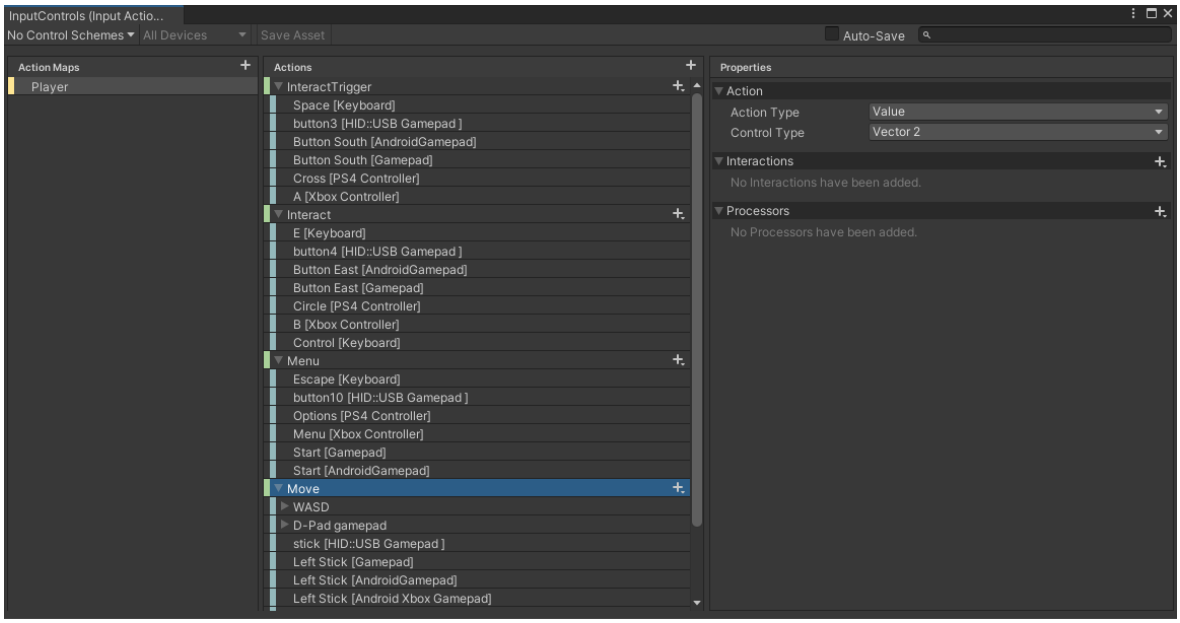


Slika 28: `Player Input` i `Player Input Handler` dodane kao komponente na igrača

`PlayerInputHandler` objašnjen je u daljnjem tekstu. Sljedeće je potrebno kreirati *action* datoteku u kojoj se definiraju sve željene akcije i radi se poveznica s fizičkim uređajima koji pokreću tu akciju. Za ovu igru definirane su ukupno četiri akcije:

- `Interact`
- `InteractTrigger`
- `Menu`
- `Move`.

`Interact`, `InteractTrigger` i `Menu` akcije definirane su kao dugme, a `Move` je akcija definirana kao `Vektor2` što znači da predstavlja dvije dimenzije od -1 do 1, a koristi se za kretanje igrača. `Menu` dugme koristi se za pauzu i ulazak u izbornik s opcijama, a preostale dvije akcije koriste se ovisno o kontekstu. Na slici 29 vidljiv je izbornik u Unityju za konfiguriranje *action* datoteke.



Slika 29: Konfiguriranje *action* datoteke u Unityju

Izbornik pruža mnogo više mogućnosti nego što je potrebno za igru koja je implementirana. Sve mogućnosti koje izbornik pruža neće biti pokrivene u ovom radu. S desne strane na slici 29 vidi se da se za svaku akciju može odabrati tip akcije i tip kontrole. Za kretanje igrača u dvije dimenzije bilo je dovoljno koristiti tip kontrole `Vector2`. Zeleno označena polja na slici predstavljaju akcije, a plave oznake predstavljaju fizičko dugme ili nekakvu drugu gestu s kojom je moguće postići pokretanje akcije. Na primjer, za `Interact` akciju definirano je da se može registrirati sa *space* dugmetom na tipkovnici, X dugmetom na PS4 kontroleru, A na Xbox kontroleru itd. Na ovaj način jednostavno je odvojiti fizičke uređaje od akcije koju izvršava igrač.

Sljedeće što treba konfigurirati jest pretplatiti se na `onActionTriggered` događaj `PlayerInput` klase. To je učinjeno u `PlayerInputHandler` klasi. Uloga je te klase da obrađuje gore spomenuti događaj, kao i da definira specifične događaje koji su važni za ovu igru.

```
void Awake()
{
    GetComponent<PlayerInput>().onActionTriggered += HandleInputAction;
}

2 references | TiTi, 20 days ago | 1 author, 6 changes
private void HandleInputAction(InputAction.CallbackContext context)
{
    if (!handleActions && context.action.name != menu)
        return;

    switch (context.action.name)
    {
        case move:
            OnMove?.Invoke(context.ReadValue<Vector2>());
            break;
        case interact_trigger:
            if(context.phase.Equals(InputActionPhase.Started))
                OnInteractTrigger?.Invoke();
            break;
        case interact:
            if (context.phase.Equals(InputActionPhase.Started))
                OnInteractStarted?.Invoke();
            else if (context.phase.Equals(InputActionPhase.Canceled))
                OnInteractEnded?.Invoke();
            break;
        case menu:
            if (context.phase.Equals(InputActionPhase.Started))
                OnMenu?.Invoke();
            break;
        default:
            Debug.LogWarning("Unhanded Input!");
            break;
    }
}
```

Ispis 36: Obrada `onActionTriggered` događaja

Ispis 36 prikazuje na koji se način obrađuju različite akcije koje su definirane u konfiguracijskoj datoteci. Sama `PlayerInputHandler` klasa ima mogućnost zaustaviti obradu ulaznih akcija, kako se npr. igrač ne bi mogao micati nakon isteka vremena ili u dijelovima igre u kojima se ne bi trebao moći micati. U `switch` uvjetu definirani su slučajevi za četiri različite akcije, a za svaku od akcija definiran je događaj. `Menu` i `interact_trigger` akcije jednostavne su jer se kod njih pozove događaj ako se akcija dogodi. Kod `move` akcije prosljeđuje se dodatan `Vector2` parametar kako je definirano u *action* datoteci ,koji se koristi za kretanje igrača u određenom smjeru. `Interact` se pokreće na dugme, ali nije implementiran kako okidač, već se kod pritiska dugmeta poziva događaj `OnInteractStarted`, a kod puštanja dugmeta poziva se događaj `OnInteractEnded`. Tako se može koristiti ta akcija kao vremenska. Primjer, igrač mora držati dugme stisnuto dok se nešto ne obavi. U ovoj je klasi i jedan zanimljiv rubni slučaj. Budući da se može dogoditi situacija da se igrač pomicao u trenutku kad se isključi obrada ulaznih akcija, mora se poništiti `OnMove` događaj. To je riješeno tako da se, prilikom

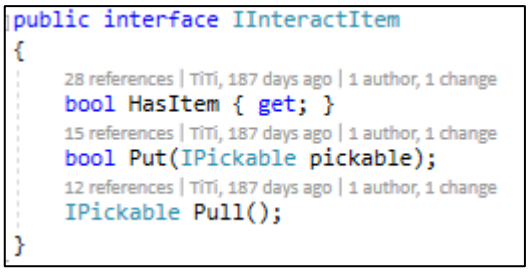
isključivanja obrade događaja, pošalje novi `OnMove` događaj s dvije nule kao vrijednost `Vector2` strukture.

Zadaća je `PlayerController` klase interakcija igrača s okolinom. Na ispisu 37 vidljiva je pretplata na događaje klase `PlayerInputHandler`.

```
Unity Message | 0 references | 0 changes | 0 authors, 0 changes
private void OnEnable()
{
    inputHandler.OnInteractTrigger += InteractTrigger;
    inputHandler.OnInteractStarted += InteractStarted;
    inputHandler.OnInteractEnded += InteractEnded;
    inputHandler.OnMenu += Menu;
    inputHandler.OnMove += Move;
}

Unity Message | 0 references | TiTi, 187 days ago | 1 author, 3 changes
private void OnDisable()
{
    inputHandler.OnInteractTrigger -= InteractTrigger;
    inputHandler.OnInteractStarted -= InteractStarted;
    inputHandler.OnInteractEnded -= InteractEnded;
    inputHandler.OnMenu -= Menu;
    inputHandler.OnMove -= Move;
}
```

Ispis 37: Pretplata na događaje `PlayerInputHandler` klase

Za pretplatu su odabrani Unity poruke `OnEnable`, a `OnDisable` za odjavu pretplate na `PlayerInputHandler` događaje. Razlog zašto nije odabran `Start` ili `Awake` Unity poruka jest taj što se `Start` poziva samo jednom po skripti, a `Awake` se ne bi trebao koristiti za ništa više nego za spremanje referenci ili inicijalizaciju lokalnih varijabli jer se zbog redoslijeda obrade može dogoditi da klasa kojoj se pristupa nije prošla svoju inicijalizaciju. U Unityju je moguće mijenjati redoslijed inicijalizacija, ali to nije preporučeno i najbolje je to izbjegavati. Kako je zamišljeno da igra bude jednostavna po pitanju kontrola, trebalo je implementirati pametan sustav za igračevu interakciju s okolinom. Igrač ima inventar (engl. *inventory*) što mu omogućava da uzme i pomiče objekte po sceni. To je realizirano klasom `HandInventory` koja implementira sučelje `IInteractItem` vidljivo na slici ispisu 38.



Ispis 38: `IInteractItem` sučelje

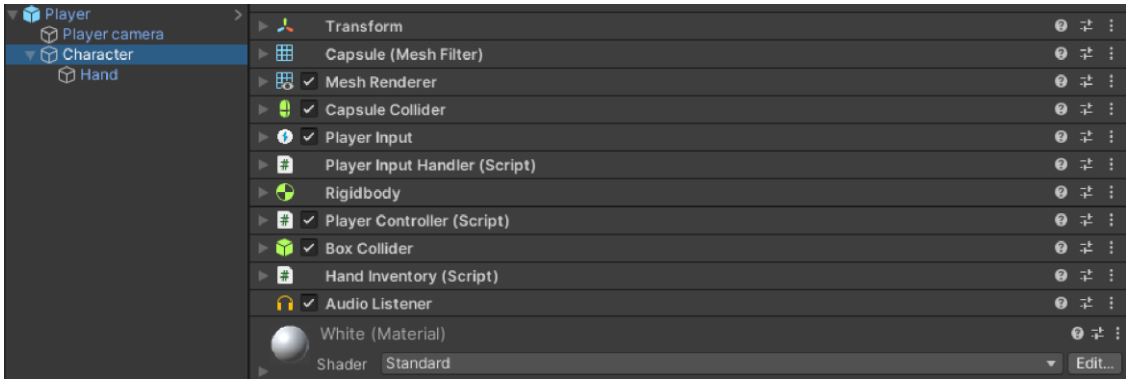
Uloga je `IInteractItem` sučelja omogućiti dohvaćanje i ispuštanje objekta koji implementira `IPickable` sučelje. `IPickable` sučelje ima metodu `GetTag` koju je potrebno implementirati i koje vraća `string`. Cilj je da se uz pomoć oznake (engl. *tag*) može saznati o kojem se tipu objekta radi. Sustav oznaka implementiran je kroz `MonoBehaviour` Unity klasu, a to je iskorišteno za implementaciju `IPickable` sučelja.

Zbog željene pametne interakcije s okolinom potrebno je drugačije postupati ako igrač ima nešto u ruci ili ako je ona prazna. Ispis 39 prikazuje kako je implementiran događaj za akciju `interact_trigger`. Prvo se provjerava ima li igrač išta u inventaru te se, ovisno o tome, poziva neka od metoda za upravljanje. Na slici je prikazana i metoda za rukovanje ako je inventar prazan, tj. ako igrač nema nijedan objekt u ruci. U prvoj `foreach` petlji prolazi se kroz `string` listu oznaka koje imaju prioritet pri obradi. Pomoću te liste može se pametno odabrati kojem će objektu igra dati prioritet u odnosu na kontekst u kojem se nalazi. Primjerice, ako se igrač nalazi u dometu kuhinjskog aparata i nekog sastojka, pokušat će se uzeti objekt iz kuhinjskog aparata jer je veća vjerojatnost da je to ono što igrač želi.

```
private void InteractTrigger()
{
    if (inventory.HasItem)
        HandleFullHand();
    else
        HandleEmptyHand();
}

private void HandleEmptyHand()
{
    foreach (var handle in emptyHandhandlingOrder)
    {
        foreach (var interactObject in objectsInRange[handle])
        {
            var firstGo = objectsInRange[handle].FirstOrDefault();
            var interacter = interactObject.GetComponent<IInteractItem>() ?? firstGo?.GetComponentInParent<IInteractItem>();
            if (interacter != null)
            {
                var item = interacter.Pull();

                if (item == null)
                    continue;

                inventory.Put(item);
                objectsInRange[handle].Remove(firstGo);
                return;
            }
        }
    }
}
```

Ispis 39: Implementacija akcije `interact`

Ako se ni u jednom kuhinjskom aparatu kojem se igrač nalazi u dometu ne nalazi nijedan `IPickable` objekt, tada se obrađuje sljedeća grupa označenih objekata te će tako igrač pokupiti sastojak. Igrač u svakom trenutku zna s kojim se poželjnim objektima nalazi u dometu. To je znanje implementirano u `objectInRange` varijabli koja je tipa `Dictionary<string, LinkedList<GameObject>>`. `String` predstavlja oznaku kojom objekti u vezanoj listi pripadaju. Radi boljeg razumijevanja potrebno je vidjeti na koji se način ta varijabla popunjava. Slika 30 prikazuje implementaciju igrača na sceni.



Slika 30: Player Unity *prefab*

Vidljivo je da je Unity objekt „Player“ spremljen kako *prefab*. *Prefab* je mogućnost da Unity snimi objekt sa svim komponentama i postavkama za svaku pojedinu komponentu objekta. On se kao takav može ponovno upotrebljavati po volji kao novi objekt tog tipa na

istoj ili drugoj sceni. [18]. U slučaju sa slike 30 „Player“ je samo pozicija na sceni koja sadrži ostale objekte. Jedan je objekt kamera koja je opisana niže u tekstu, a drugi je „*Character*“ što predstavlja fizičku i logičku srž igrača. Na slici je označen „*Character*“ objekt i vide se sve komponente dodane na njega. Tu se vide i skripte koje su ranije opisane, a vidi se i `Hand` objekt koji predstavlja lokaciju u odnosu na igrača, tj. pozicija u kojoj će biti prikazan objekt koji igrač ima u ruci. Za popunjavanje `objectInRange` varijable ponajviše je zadužena komponenta *BoxColider* uz samu `PlayerController` klasu. Sam *collider* definiran je kao okidač te kao takav neće onemogućavati prolaz drugih objekata s *colliderom* u njega, već će, ako se dogodi da detektira da neki drugi objekt ulazi u njega, javiti to Unity porukom. Implementacija poruke u `PlayerController` klasi vidi se na ispisu 40. `OnTriggerEnter` Unity poruka implementirana je tako da ovisno o oznaci objekta dodaje isti u odgovarajuću listu kao prvi element. Kod izlaska igrača izvan dosega, jednostavno se pogleda je li objekt ima oznaku koja je zanimljiva, ako da, pronađe se i uklanja iz liste.

```
private void OnTriggerEnter(Collider other)
{
    switch (other.gameObject.tag)
    {
        case table:
            objectsInRange[table].AddFirst(other.gameObject);
            break;
        case kitchen_appliance:
            objectsInRange[kitchen_appliance].AddFirst(other.gameObject);
            break;
        case kitchen_tool:
            objectsInRange[kitchen_tool].AddFirst(other.gameObject);
            break;
        case kitchen_storage:
            objectsInRange[kitchen_storage].AddFirst(other.gameObject);
            break;
        case ingredient:
            objectsInRange[ingredient].AddFirst(other.gameObject);
            break;
        case food_plate:
            objectsInRange[food_plate].AddFirst(other.gameObject);
            break;
        default:
            Debug.LogWarning($"Unhandled trigger on player controller: {other.gameObject.tag}");
            break;
    }
}

Unity Message | 0 references | TiTi, 63 days ago | 1 author, 6 changes
private void OnTriggerExit(Collider other)
{
    if (objectsInRange.ContainsKey(other.gameObject.tag))
        objectsInRange[other.gameObject.tag].Remove(other.gameObject);

    //TODO: Handle interaction lost range -> stop timer for interaction if out of range
}
```

Ispis 40: Implementacija Unity poruka za *collider* kao okidač

U tekstu ispod slijedi objašnjenje glavnih komponenti za igrača:

- *Transform* – komponenta koja sadrži podatke o poziciji, orijentaciji i veličini objekta.
- *Mesh filter* i *renderer* – komponente koje služe za grafički prikaz 3D objekta.
- *Audio listener* – služi kao ulaz zvuka, ako se ima 3D zvuk, služi i za pravilno prenošenje istog u odnosu na trenutnu lokaciju ove komponente.
- *Capsule collider* – služi kako bi se definirale fizičke dimenzije objekta. Sustav za koliziju služi kako bi spriječio *collidere* da ulaze jedan u drugi te tako sprječava da igrač prolazi kroz zidove, pod ili kroz druge 3D objekte na sceni koji imaju *collider*.
- *Box collider* – na igraču služi kao okidač, registrira sve objekte u području koji na sebi imaju neki *collider* te za svakog pozivaju `OnTriggerEnter` i `OnTriggerExit` Unity poruke.
- *Rigidbody* – implementira gravitaciju i fiziku nad igračem i omogućuje da igrač može fizički interaktirati s drugim objektima koji imaju istu komponentu.

Ostale komponente na igraču opisane su ranije.

Kamera je implementirana tako da prati igrača uz pomoć generičko implementirane klase `GameObjectFollow`. Na slici 31 vidi se upotreba te klase u Unity *editoru*. Klasa izlaže dvije varijable, `Item` i `distanceFromItem` koje se mogu popunjavati iz Unity *editora*.

Slika 31: Prikaz `GameObjectFollow` klase u Unity *editoru*

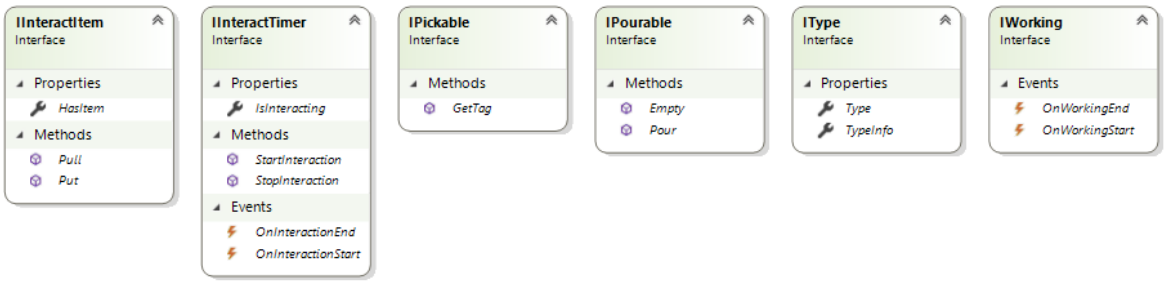
Na slici je vidljivo da je sa slike 30 uzet „*Character*“ Unity objekt kao vrijednost varijable `Item`, a za distancu je određeno da bi kamera trebala biti udaljena od igrača 8 u Y i -7 u Z smjeru. Ista ova skripta upotrebljava se i na drugim mjestima.

Treba napomenuti da će sve javne varijable neke klase biti prikazane u Unity *editoru*. Ako ne odgovara da varijabla bude javna, a želi se ta funkcionalnost, tada treba postaviti atribut `SerializeField` za tu varijablu. Nije moguć prikaz svih tipova varijabli u Unity

editoru, već je to omogućeno samo za neke tipove podataka. Za sve ostale je moguće implementirati određena sučelja i metode kako bi se ta mogućnost dobila.

4. 2. 2. Sučelja

Nakon razumijevanja implementacije igrača potrebno je reći nešto o sučeljima koji su implementirani za ovu igru. Neka od njih spomenuta su ranije, a puna lista implementiranih sučelja vidi se na slici 32.



Slika 32: Sučelja implementirana u igri „*The dream*“

`IInteractItem` i `IPickable` interface objašnjeni su ranije, ali ukratko će biti napisana i njihova uloga.

`IInteractItem` sučelje omogućava fizičku interakciju igrača s okolinom.

`IPickable` omogućava igraču da neke objekte premješta s jednog mjesta na drugo ili ih dodaje u neke druge objekte korištenjem ovog i `IInteractItem` sučelja.

`IInteractTimer` sučelje služi za vremensku interakciju igrača. Samo sučelje definira dvije metode za početak i kraj interakcije, kao i dva događaja za te metode.

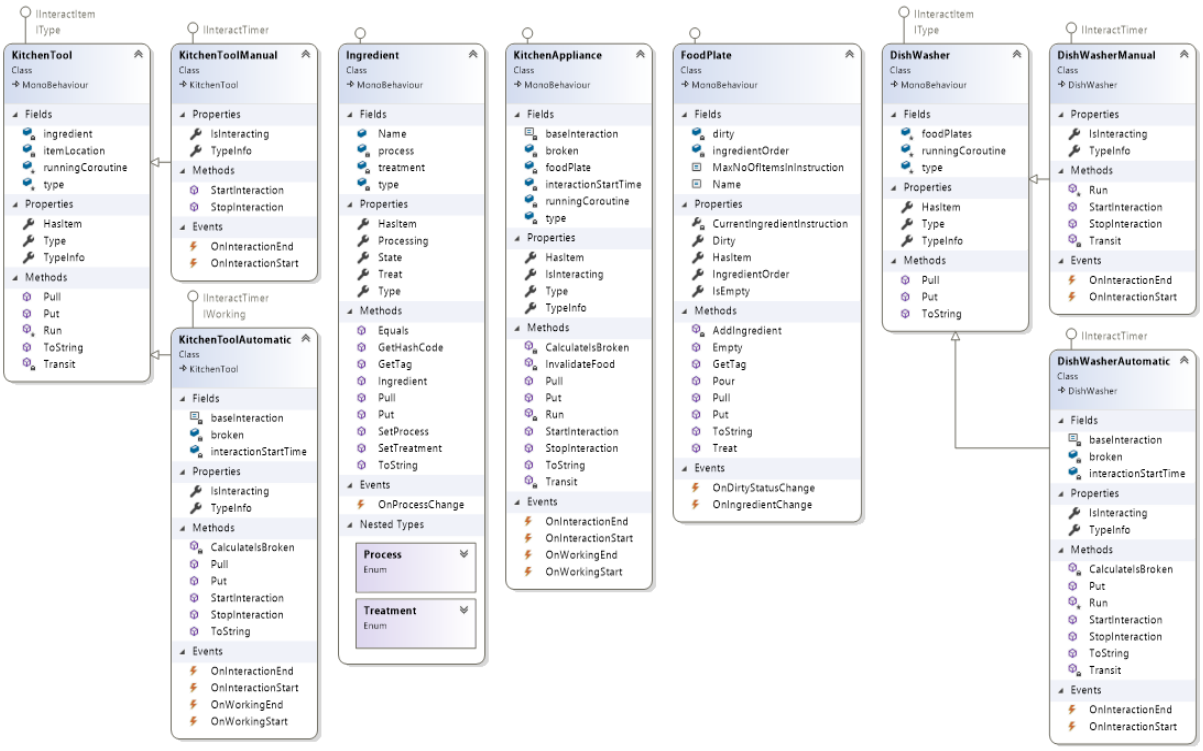
`IPourable` definira metode `Empty` i `Pour`. Služi za sastavljanje kompleksnih recepata koji se sastoje od više instrukcija recepata. Jedan takav prikazan je i objašnjen na slici 27.

`IType` definira tip `ScriptableObject` klase kao i njezinu `string` reprezentaciju koja služi za rukovanje `ScriptableObject` klasama.

`IWorking` sučelje definira dva događaja, a to su početak i kraj rada uređaja.

4. 2. 3. Recepti i interakcija s receptima

Ovo potpoglavlje opisuje kreiranje recepta i interakciju igrača s narudžbama. Na slici 33 vide se glavne klase koje su potrebne kako bi igrač sastavio narudžbu. Na vrhu svake klase prikazano je koja sučelja implementira pojedina klasa. Ispod slijedi ime klase i podatak od koje se klase nasljeđuje. Kako je bitno da se sve ove klase mogu nalaziti na sceni, tj. Unity objektu, tako sve klase moraju nasljeđivati od Unity `MonoBehaviour` klase. Nakon toga izlistane su sve metode, varijable i događaji koji su implementirani za tu klasu. Strelice prikazuju ovisnost nasljeđivanja kod nekih klasa. Treba napomenuti da postoji greška kod iscrtavanja dijagrama klasa te se zbog toga na nekim klasama ne prikazuju koja su sučelja implementirana. Kod daljnjega objašnjenja pojedine klase bit će napomenuto koja sve sučelja ta klasa implementira.



Slika 33: Glavne klase potrebne igraču za sastavljanje recepta

`Ingredient` služi za praćenje trenutnog stanja pojedinog sastojka. Klasa implementira `IPickable` i `IInteractItem` sučelja. `IPickable` sučelje omogućava premještanje objekta s jednog mjesta na drugo, dok `IInteractItem` omogućava

interakciju s tim objektom. `Pull` metoda implementirana je tako da vraća sastojak, a `Put` metoda vraća *`false`*. U budućnosti, ako bi kompleksniji recepti od trenutnih pridonijeli igri, može se implementirati dodavanje začina na sastojak. Unutar klase implementirana su i dva `enuma – Process` i `Treatment`. `Process` opisuje što se napravilo sa sastojkom (npr. usitnjavanje, mljevenje, cijeđenje i slično). `Treatment` opisuje koji se temperaturni tretman odvio nad tim sastojkom. Neke su od mogućih vrijednosti kuhanje, friganje, smrzavanje i sl. `Type` varijabla služi za opisivanje o kojem se točno sastojku radi. `Type` je tipa `SOIngredient` i objašnjen je kasnije u radu. Za sada je dovoljno znati da jednoznačno opisuje neki sastojak kao i njegove karakteristike, npr. limun. To se donekle može vidjeti na ispisu 41. Na slici se vidi da se provjerava, preko tipa sastojaka, može li objekt prijeći u to stanje. Ako je to moguće, javlja se novo stanje svim zainteresiranima putem događaja.

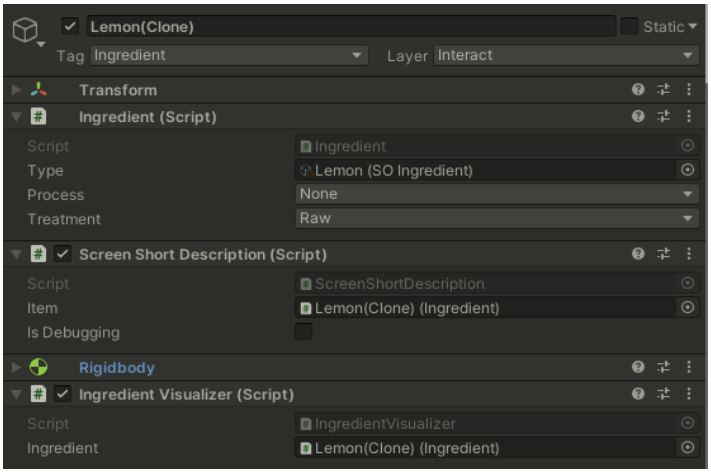
```
public void SetProcess(Process newState)
{
    if (process == newState)
        return;

    process = type.HasTransition(process, newState) ? newState : Process.Invalid;
    OnProcessChange?.Invoke(this, new IngredientProcessChangeEventArgs() { Ingredient = this, NewProcess = process });
}
```

Ispis 41: Metoda za postavljanje procesa klase `Ingredient`

Slika 34 prikazuje komponente od koji se sastoji sastojak. Vidi se `Ingredient` klasa kao komponenta i da je za tip sastojaka odabran limun koji nije procesiran, a nije ni termički obrađen.

Slijedi `ScreenShortDescription` komponenta, pomoćna generička klasa koja korištenjem `ToString` metode bilo kojeg objekta ispisuje bitne podatke tog objekta na ekran. Ta klasa služi za *debugging*. Zatim slijedi Unity *rigidbody* komponenta zaslužna za interakciju s fizikom i drugim objektima koji imaju *rigidbody* komponentu. Ovaj projekt nastoji slijediti SOLID principe i zbog toga su odvojene podatkovne, vizualne i druge razine u igri.

`IngredientVisualizer` je klasa koja se brine za vizualnu reprezentaciju sastojka na sceni. Pretplatom na `OnProcessChange` i poznavanjem tipa sastojka pomoću `IType` sučelja, odlučuje se na koji će se način prezentirati sastojak na sceni. Slika 34 ne prikazuje nijedna vizualna komponenta na samom objektu jer se vizualna komponenta dodaje kao dijete sastojka.



Slika 34: Komponente sastojka na Unity sceni

Za procesuiranje sastojaka koristi se neki tip kućanskih alata. Slika 33 pokazuje da se kućanski alati dijele na ručne i automatske. Obje klase nasljeđuju od bazne klase `KitchenTool` koja implementira `IInteractItem` i `IType` sučelja. `Type` varijabla kuhinjskog alata definirana je kao `SOKitchenTool` koji opisuje koji će se proces vršiti nad sastojkom. `SOKitchenTool` nasljeđuje od `SOKitchenMachine` koje definira karakteristike uređaja, poput:

- vremena potrebnog za obradu
- mogućnosti i vjerojatnosti da se uređaj pokvari
- mogućnosti i vremena da se predmet nad kojim se vrši obrada uništi.

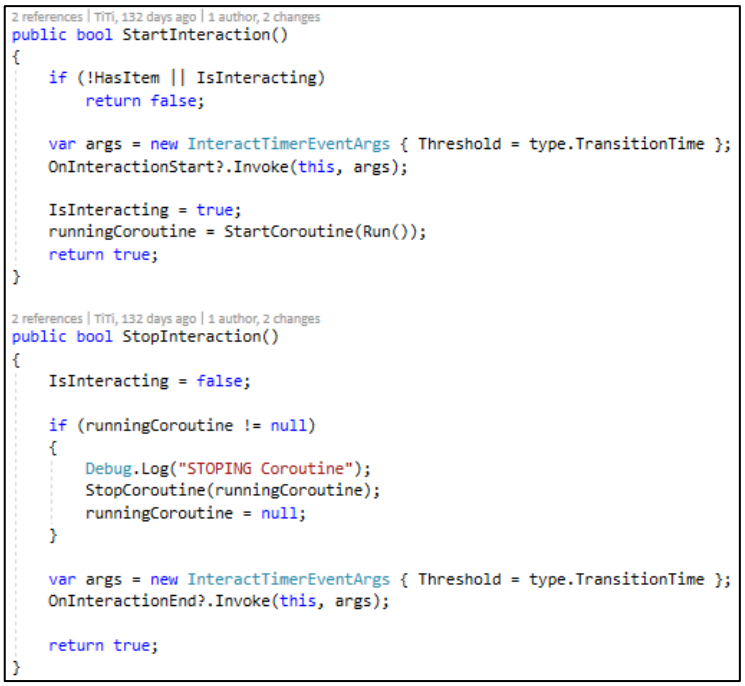
Od važnijih metoda `KitchenTool` implementira zaštićenu `coroutinu Run` koja uz pomoć varijable `Type` određuje nakon koliko će vremena pozvati privatnu metodu `Transit` koja mijenja stanje sastojka. Ispis 42 prikazuje na koji se način vrši tranzicija u `KitchenTool` klasi.

```
3 references | TiTi, 179 days ago | 1 author, 2 changes
protected IEnumerator Run()
{
    Debug.Log("Coroutine running");
    yield return new WaitForSeconds(type.TransitionTime);
    Transit();
}

1 reference | TiTi, 190 days ago | 1 author, 1 change
private void Transit()
{
    Debug.Log($"Ingredient transition {type.Transition}");

    if(!ingredient)
    {
        Debug.LogWarning($"Wrong call of Transit for transition {type.Transition} by {this}");
        return;
    }

    ingredient.SetProcess(type.Transition);
}
```

Ispis 42: Implementacija `coroutine` i tranzicije sastojka u `KitchenTool` klasi

Klasa koja nasljeđuje klasu `KitchenTool` treba se pobrinuti da se `Run` poziva u pravodobno vrijeme i da se poziva samo kad je neki sastojak u kućanskom alatu, međutim, greške se događaju i najlakši je način otkrivanja da ih se očekuje te se s njima pravilno rukuje i zapisuje (engl. *logging*). Zbog toga metoda `Transit` ispisuje poruku upozorenja ako se pokrene bez sastojka. Treba napomenuti da se tipovi metoda `coroutine` mogu zaustaviti. `IInteractItem` sučelje implementirano je tako da kućanski alat može primiti jedan sastojak i vratiti sastojak koji se nalazi u njemu.

`KitchenToolManual` implementira `IInteractTimer` i `IType` sučelja koji su djelomično već implementirani u baznoj klasi. Svrha je ove klase ručna obrada nekog sastojaka, tj. ako igrač dovoljno dugo zadrži definirano dugme, tada će se sastojak obraditi. Ispis 43 prikazuje kako je `IInteractTimer` sučelje implementirano u ovoj klasi.



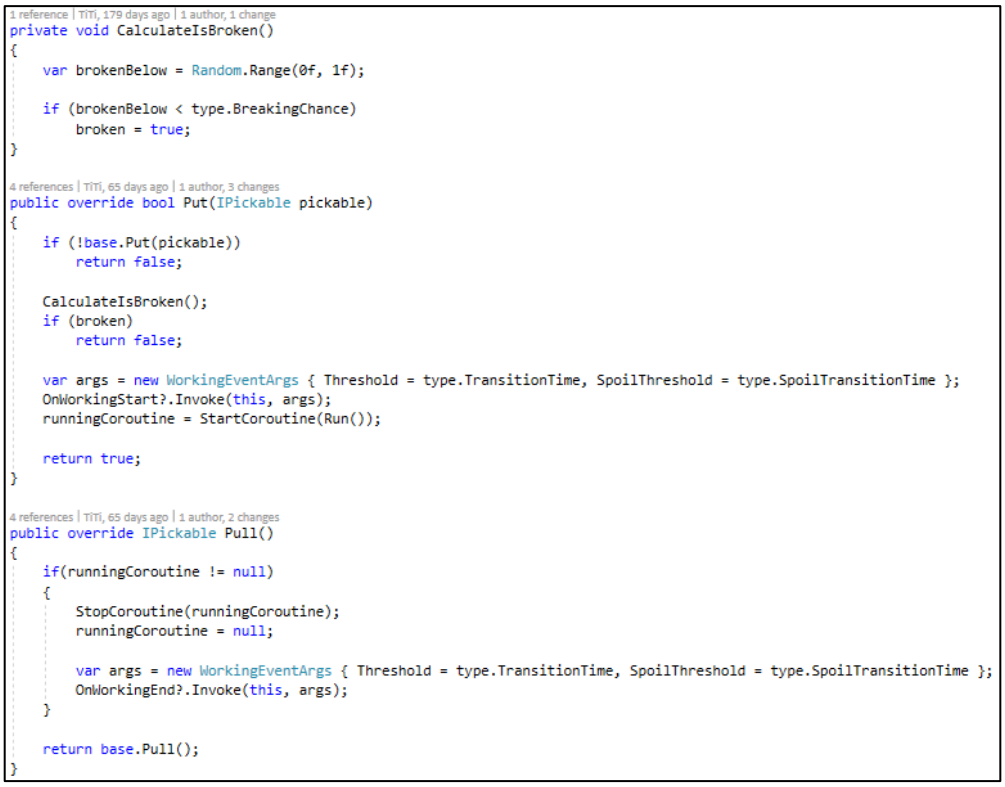
```
2 references | TiTi, 132 days ago | 1 author, 2 changes
public bool StartInteraction()
{
    if (!HasItem || IsInteracting)
        return false;

    var args = new InteractTimerEventArgs { Threshold = type.TransitionTime };
    OnInteractionStart?.Invoke(this, args);

    IsInteracting = true;
    runningCoroutine = StartCoroutine(Run());
    return true;
}

2 references | TiTi, 132 days ago | 1 author, 2 changes
public bool StopInteraction()
{
    IsInteracting = false;

    if (runningCoroutine != null)
    {
        Debug.Log("STOPING Coroutine");
        StopCoroutine(runningCoroutine);
        runningCoroutine = null;
    }

    var args = new InteractTimerEventArgs { Threshold = type.TransitionTime };
    OnInteractionEnd?.Invoke(this, args);

    return true;
}
```

Ispis 43: Implementacija `IInteractTimer` sučelja u `KitchenToolManual` klasi

`StartInteraction` metodu poziva se kad igrač rukuje s tim kućanskim alatom. Ako objekt ima neki sastojak u sebi, tada šalje svim zainteresiranima da se dogodio događaj interakcije s tim uređajem. Potrebni podaci šalju se kao argument događaja i postavlja se svojstvo `IsInteracting` u *`true`*. Potrebno je i spremiti referencu na `coroutinu` koja se pokreće iz bazne klase kako bi se po potrebi mogla zaustaviti.

Kod `StopInteraction` metode zaustavlja se `coroutinu`, ako postoji referenca na nju, a zatim se šalje obavijest o događaju i postavlja `IsInteracting` svojstvo u *`false`*.

`KitchenToolAutomatic` klasa implementira `IInteractTimer`, `IType` i `IWorking` sučelja. Automatski kućanski alati rade drugačije od ručnih. Dok je kod ručnih potrebno zadržati dugme da sastojak prijeđe iz jednog svojstva u drugo, u automatskim kućanskim alatima to se događa kad se neki sastojak doda u automatski alat. Sastojak neće biti obrađen ako je uređaj pokvaren. `IInteractTimer` sučelje implementirano je za popravak uređaja, na sličan način kao što je za obradu sastojka implementirano u `KitchenToolManual` klasi. Ispis 44 prikazuje način na koji je implementiran prijelaz sastojka u ovoj klasi.



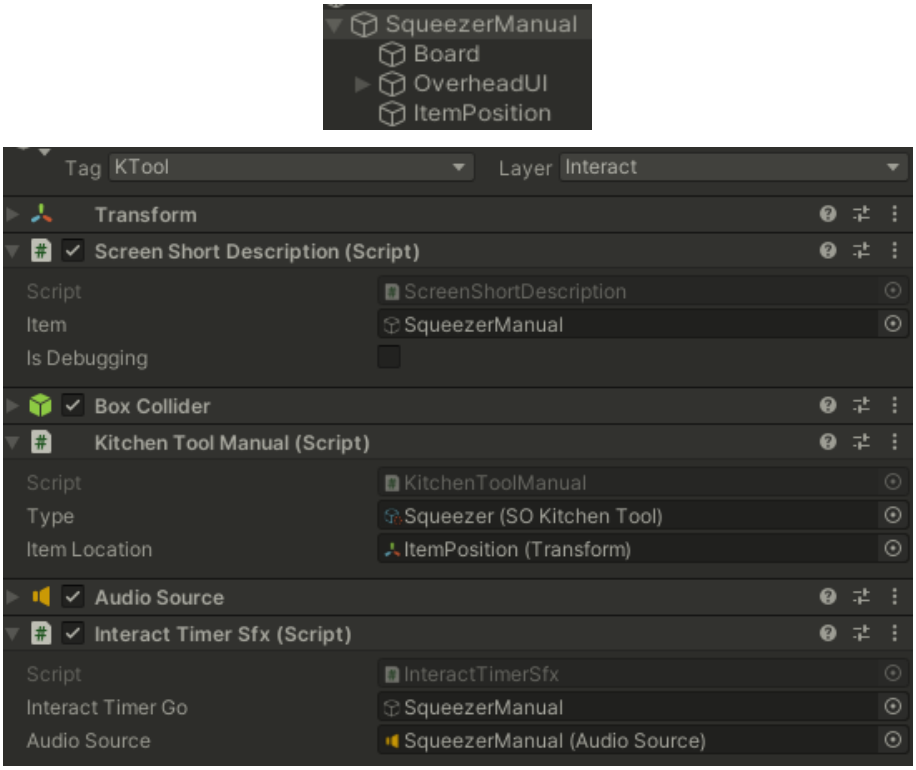
```
1 reference | TiTi, 179 days ago | 1 author, 1 change
private void CalculateIsBroken()
{
    var brokenBelow = Random.Range(0f, 1f);

    if (brokenBelow < type.BreakingChance)
        broken = true;
}

4 references | TiTi, 65 days ago | 1 author, 3 changes
public override bool Put(IPickable pickable)
{
    if (!base.Put(pickable))
        return false;

    CalculateIsBroken();
    if (broken)
        return false;

    var args = new WorkingEventArgs { Threshold = type.TransitionTime, SpoilThreshold = type.SpoilTransitionTime };
    OnWorkingStart?.Invoke(this, args);
    runningCoroutine = StartCoroutine(Run());

    return true;
}

4 references | TiTi, 65 days ago | 1 author, 2 changes
public override IPickable Pull()
{
    if(runningCoroutine != null)
    {
        StopCoroutine(runningCoroutine);
        runningCoroutine = null;

        var args = new WorkingEventArgs { Threshold = type.TransitionTime, SpoilThreshold = type.SpoilTransitionTime };
        OnWorkingEnd?.Invoke(this, args);
    }

    return base.Pull();
}
```

Ispis 44: Implementacija tranzicije svojstva sastojka u `KitchenToolAutomatic` klasi

Na slici je vidljivo da je bazna implementacija metode `Put` pregažena. Bazna implementacija metode poziva se za dodavanje sastojka u uređaj. Ako ta metoda vrati *`false`*, to znači da objekt koji implementira `IPickable` nije zadovoljavajućeg tipa za kućanski alat. Kućanski alat može primiti samo sastojke te ima saznanje o tome je li nešto sastojak korištenjem metoda `IPickable` sučelja. Kad je to zadovoljeno, računa se je li uređaj pokvaren koristeći Unity metodu `Random.Range` i `BreakingChance` svojstvo definirano u tipu automatskog kućanskog alata. Ako alat nije pokvaren, obavještava se preko `IWorking` sučelja da je rad alata započeo i pokreće se bazna `coroutina`. Ako igrač odluči uzeti sastojak prije tranzicije sastojka, `coroutina` se zaustavlja i igraču se daje sastojak u nepromijenjenom stanju.

Slika 35 prikazuje kako u Unity hijerarhiji na sceni izgleda ručno cjedilo za limune, kao i koje komponente sadrži objekt kojem je dodijeljena `KitchenToolManual` komponenta. Vidljivo je da se objekt cijedila za limune sastoji od više objekata djece, a neki od njih imaju i svoju djecu objekte. Slika prikazuje i od kojih je komponenta sastavljen `SqueezerManual` objekt. `InteractTimerSfx` je generička klasa za procesuiranje audio sadržaja bazirana na `IInteractTimer` sučelju.

`SquezzerManual` ima troje djece vezanih za sebe. `Board` dijete vizualna je reprezentacija. `OverheadUI` je UI komponenta za prikaz UI elemenata vezanih za alat, poput interakcije i stanja objekta. `ItemPosition` je pozicija u Unity sceni koja se prosljeđuje `KitchenToolManual` klasi koja prilikom dodavanja sastojaka u alat premješta sastojak na navedenu lokaciju na sceni. Na vrhu slike vidi se i Unity tag sustav te je ovaj objekt tagiran kao `KTool`.



Slika 35: Komponente ručnog kuhinjskog alata

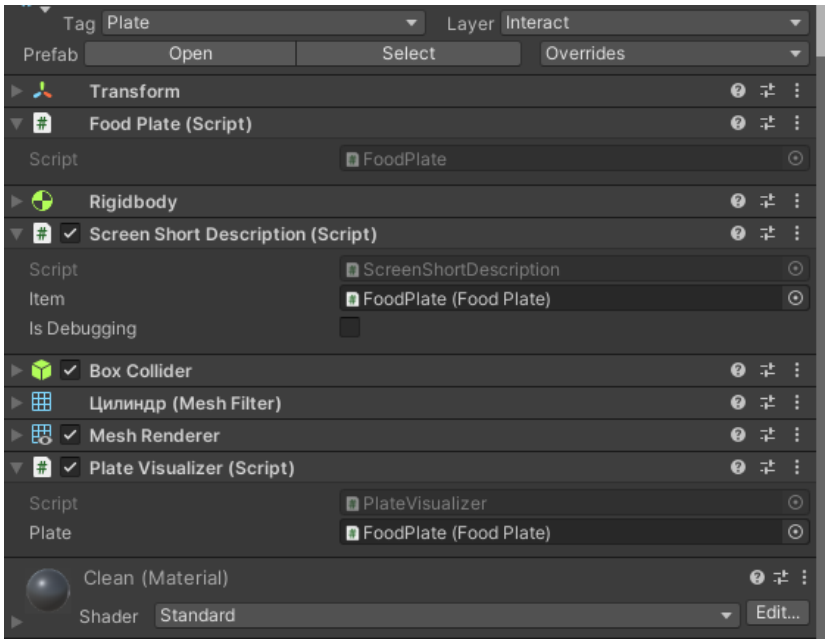
`FoodPlate` je klasa kojoj je uloga kombiniranje jednog ili više sastojaka. Koristi se i kao završni proizvod koji se kao takav odnosi agentu, tj. naručitelju. Klasa implementira sučelja `IPickable`, `IIntearactItem` i `IPourable`. Pomoću sučelja `IPourable` moguće je dodavanje grupe sastojaka ili pražnjenje između objekata `FoodPlate` klase. Tako je omogućeno sastavljanje kompleksnih recepta koji se sastoje od više neovisnih instrukcijskih cjelina.

Varijabla `ingredientOrder` definira je kao lista liste sastojaka. Unutarnja lista sastojaka predstavlja jednu neovisnu instrukcijsku cjelinu. Preko metode `Pour` iz `IPourable` sučelja dodaju se završene instrukcijske cjeline unutar `FoodPlate` klase. U trenutni `FoodPlate` objekt dodavanje pojedinih sastojaka uvijek ide na prvo mjesto u nizu.

Prvotno je bilo na zadnjem mjestu, ali to nije bilo intuitivno igračima koji su isprobali ovu igru. Na isti su način promijenjeni i neki drugi implementacijski detalji. Na ovaj je način moguće neovisno graditi jednu instrukciju recepta u jednom `FoodPlate` objektu, dok se paralelno može dodavati u taj isti objekt druge instrukcije recepta sastavljene na nekom drugom `FoodPlate` tipu objekta. Radi jednostavnosti implementacije verifikacije recepta `MaxNoOfItemsInInstruction` je postavljen na 64, što je više nego dovoljno jer je zamišljeno da instrukcija recepta ne bi trebala biti kompliciranija od četiri različita sastojka.

`IIntearactItem` sučelje implementirano je tako da metoda `Pull` vraća `FoodPlate` objekt, dok metoda `Put` prihvaća samo `Ingredient` objekte. Dodavanje novog sastojka rezultira `OnIngredientChange` događajem koji obavještava sve zainteresirane s parametrima trenutnih instrukcija i sastojaka na `FoodPlate` objektu, kao i parametrom promjena na objektu. Isti se događaj poziva i kod poziva `Pour` metode.

Slika 36 prikazuje komponente `FoodPlate` Unity objekta. Sve komponente objašnjene su ranije pa se neće ovdje posebno napominjati. Nova je komponenta klasa `PlateVisualizer` koja uz pomoć `SOPlateVisualizer` klase mapira vizualni izgled `FoodPlate` objekta u odnosu na to koji se sastojci nalaze u `FoodPlate` objektu, u kojem su stanju i je li `FoodPlate` svojstvo `Dirty` u *`true`*.



Slika 36: Komponente posude za sastojke

`KitchenAppliance` implementira sučelja `IInteractItem`, `IInteractTimer`, `IWorking` i `IType`. Uloga je ove klase termička obrada sastojaka sadržanih u `FoodPlate` klasi. Slično kao i `KitchenTool` klase `Treatment` svojstvo je definirano u `SoKitchenAppliance` klasi koje se postavlja u Unity editoru. On nasljeđuje od `SOKitchenMachine`, koja sadrži ostale statičke karakteristike uređaja, šanse za kvar, vrijeme potrebno za tranziciju i vrijeme nakon tranzicije da `FoodPlate` postane nevaljan.

Ispis 45 prikazuje termičku obradu hrane iz `KitchenAppliance` klase. `Coroutina Run` radi na isti način kao `KitchenToolAutomatic` klasa s razlikom da se ovdje ubacuje kao `IPickable FoodPlate` objekt, a ne `Ingredient`. Tranziciju svakog `Ingredient` objekta vrši `FoodPlate` klasa pozivom `SetTreatment` metode `Ingredient` klase.

```
private IEnumerator Run()
{
    Debug.Log("Coroutine running");

    yield return new WaitForSeconds(type.TransitionTime);
    Transit();

    yield return new WaitForSeconds(type.SpoilTransitionTime);
    InvalidateFood();
}

1 reference | TiTi, 66 days ago | 1 author, 4 changes
private void Transit()
{
    Debug.Log($"Ingredient treatment {type.Transition}");

    foodPlate.Treat(type.Transition);
```

FoodPlate.cs

```
            return true;
        }

        public void Treat(Ingredient.Treatment treatment)
        {
            foreach (var ingredientOrder in ingredientOrder)
                foreach (var ingredient in ingredientOrder)
                    ingredient.SetTreatment(treatment);

            var newIngredientList = new List<List<Ingredient>>();
            newIngredientList.Add(new List<Ingredient>());
            var args = new PlateChangeEventArgs { Treatment = treatment, CurrentIngredients = ingredientOrder, NewIngredients = newIngredientList };
            OnIngredientChange?.Invoke(this, args);
        }
```

Ln: 86 Ch: 55 SPC CRLF

```
}

1 reference | TiTi, 66 days ago | 1 author, 2 changes
private void InvalidateFood()
{
    Debug.Log($"Ingredient invalidated");

    foodPlate.Treat(Ingredient.Treatment.Invalid);
}
```

Ispis 45: Termička obrada sastojaka u `KitchenAppliance` klasi

Kao i ostali Unity elementi, i `KitchenAppliance` se sastoji od niza komponenti, najsličnijim komponentama s ispisa 44, samo što implementira `WorkingSfx` klasu za reprodukciju zvuka. Oznaka je za ovaj element `KAppliance`.

Sučelje `IInteractItem` implementirano je kao i kod `KitchenTool` klase, s razlikom da ne prima sastojak, već instancu klase `FoodPlate`. Na isti način implementirano je sučelje i u `DishWasher` klasi.

Uloga je `DishWasher` klase očistiti posudu u kojoj se nalazila hrana, tj. postaviti svojstvo `Dirty` objekta tipa `FoodPlate` u *`false`* te maknuti sve sastojke i instrukcije iz objekta klase. `DishWasher` je bazna klasa `DishWasherManual` i `DishWasherAutomatic` klasi. Implementira sučelja `IInteractItem` i `IType`. `DishWasher` može primiti n `FoodPlate` objekata te zbog toga drugačije funkcionira rad `DishWasherManual` i `DishWasherAutomatic` klase.

`DishWasherManual` očisti jednu po jednu posudu interakcijom s igračem, dok `DishWasherAutomatic` očisti sve posude dokle god se ubace prije nego istekne vrijeme potrebno da se pokrene metoda za čišćenje. Zbog tih je razlika `coroutina Run` implementirana na način da prima delegat bez dodatnih parametara. Implementacija je prikazana na ispisu 46.

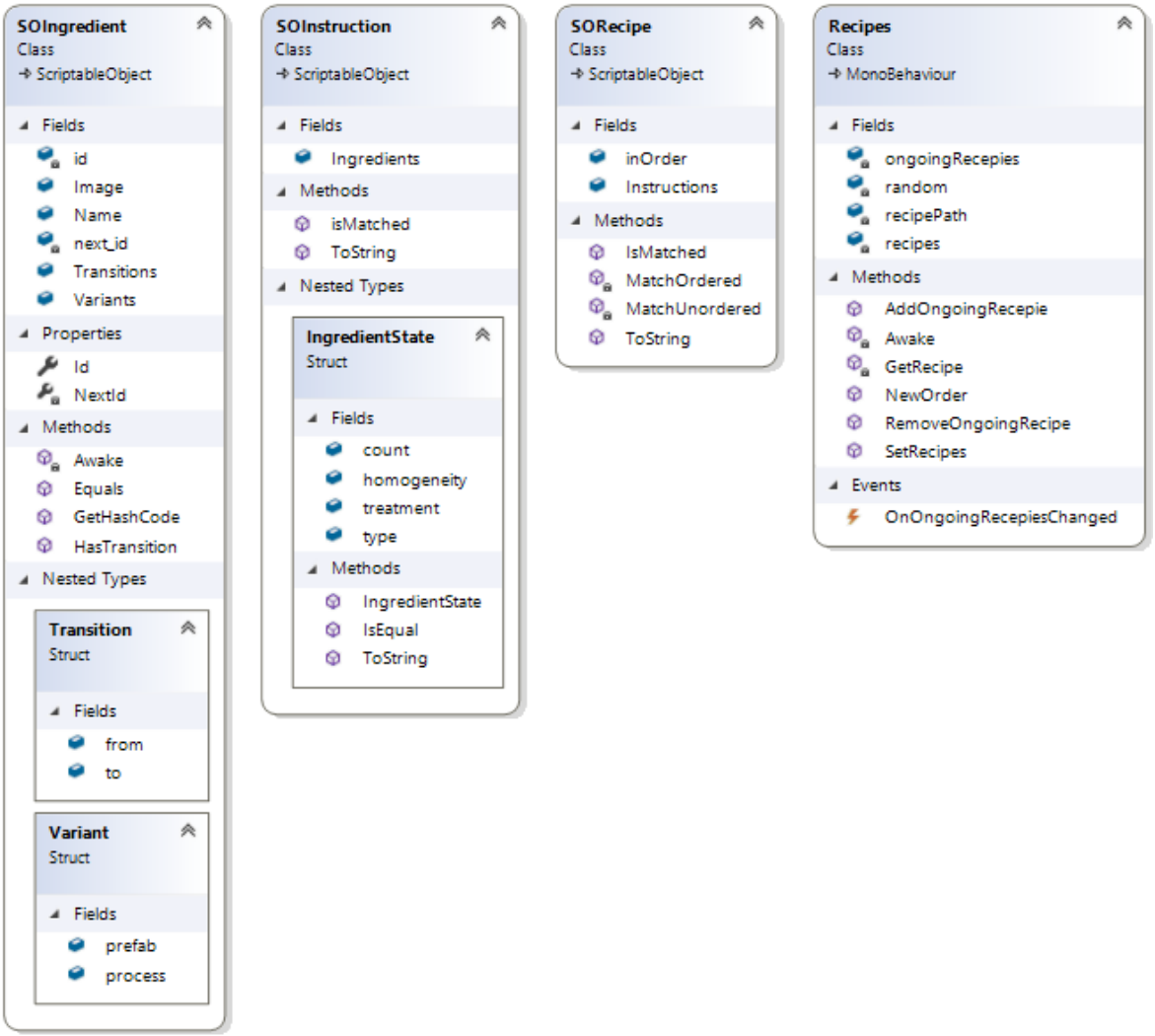
```
protected IEnumerator Run(System.Action transit)
{
    Debug.Log("Coroutine running");
    yield return new WaitForSeconds(type.TransitionTime);
    transit();
}
```

Ispis 46: `Coroutina Run` s delegatom kao parametrom u `DishWasher` baznoj klasi

Kod ručnog pranja ne može doći do kvara, a kod automatskog, npr. u perilici za posuđe, može. Popravlja se na isti način kao i `KitchenToolAutomatic` klasa pa je i sama implementacija slična.

Kreiranje recepata i definiranje svih potrebnih parametara nasljeđuje se od `ScriptableObject` klase. To je Unity klasa koja omogućava jednostavnu implementaciju *Flyweight desing patterna* – način strukturiranja koda koji omogućuje dijeljenje zajedničkih dijelova podataka s objektima istog tipa. U memoriji se pojavljuje

samo jednom bez obzira na broj takvih objekata na sceni. Primjerice, tip neprijatelja koji ima neka svoja statička svojstva, poput maksimalne brzine, razine zdravlja, tipa napada i slično. Te podatke dovoljno je jednom učitati u memoriju i oni se mogu dijeliti među svim instancama tog tipa neprijatelja. [18, 19] `ScriptableObject` koriste se još i kao podatkovni kontejneri. Zgodna je stvar što se mogu implementirati tako da se kreiraju iz Unity *editora*. Ako bi na projektu radilo više ljudi, dizajner recepata ne treba ulaziti u kod ni funkcioniranje istog. On treba kreirati objekt za recept i popuniti parametre koje se od njega traže. Za kreiranje recepta na taj način implementirane su klase čija su osnovna svojstva prikazana na slici 37. Svaka klasa sa SO prefiksom nasljeđuje od `ScriptableObject` klase.

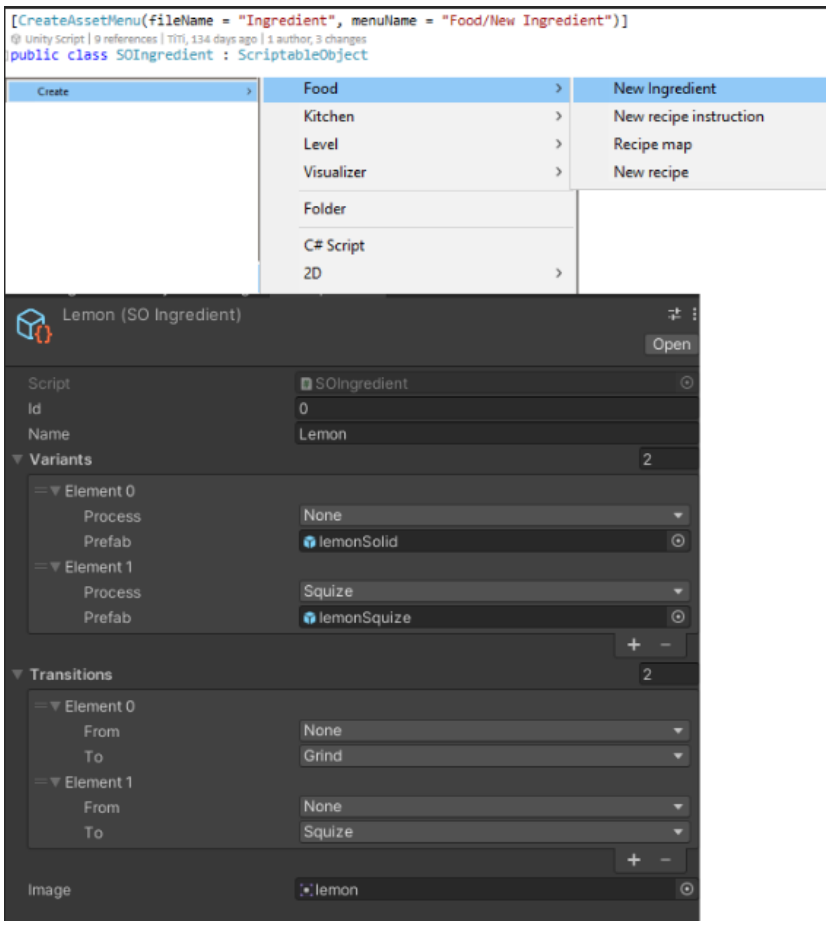


Slika 37: Glavne klase potrebne za sastavljanje recepta

Klasa `SOIngredient` predstavlja svojstva jednog sastojka. Za definiciju sastojka potrebno je:

- popuniti ime sastojka
- postaviti 2D sliku sastojka koja će se koristiti kao UI element u prikazu recepta
- postaviti popis mogućih prijelaza iz prijelaza X u prijelaz Y
- za svaku moguću varijantu postaviti 3D objekt i naznačiti o kojoj se varijanti radi.

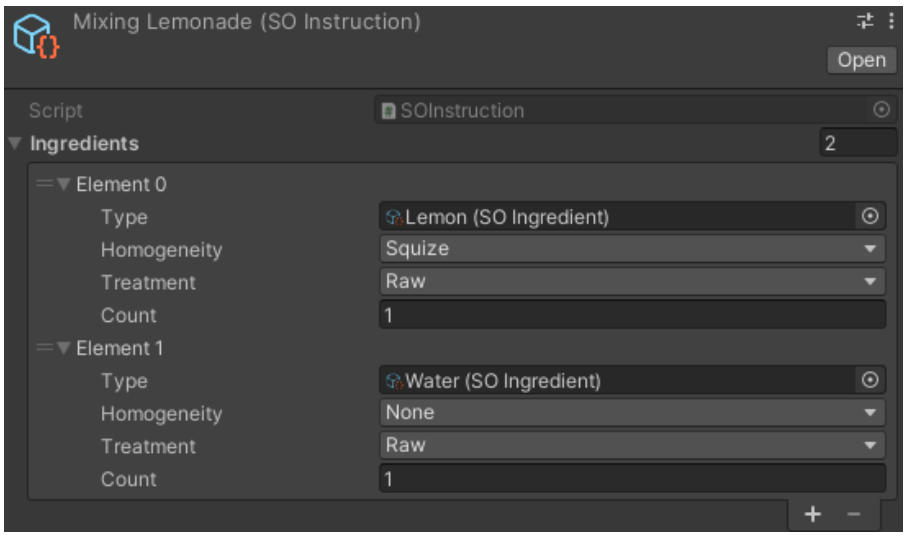
`Id` se popunjava automatski i uloga mu je da kod provjere recepta ista bude brza i jednostavna. Razlog tome je što je dovoljno provjeriti `Id` kako bi se zaključilo da se radi o točnom sastojku. Slika 38 prikazuje s kojim je atributom definirana putanja za kreiranje instance klase iz Unity *editora*. Sredina slike prikazuje odabir te putanje, a donji dio slike prikazuje prazan izbornik za popuniti s gore navedenim svojstvima. Svaka instanca `ScriptableObject` klase snima se kao datoteka.



Slika 38: Definiranje i kreiranje `SOIngredient` klase

Na slici je definiran sastojak limun. Definirani su prijelazi iz `None` u `Grind` i iz `None` u `Squize`. Svojstvu `Variants` dodijeljena su dva 3D objekta varijanata za svaki prijelaz. Klase zadužene za vizualnu reprezentaciju sastojka, koje su ranije spomenute, koriste `Variants` javnu listu kako bi mijenjali vizualni izgled sastojka, dok `KitchenTool` klasa koristi `HasTransaction` za verifikaciju prijelaza.

Klasa `SOInstruction` sadrži listu `IngredientState` objekata. Slika 39 prikazuje kako izgleda Unity izbornik kod kreiranja instance te klase. Svaki je element tipa `IngredientState`. Na slici je prikazana instrukcija miješanja limunade. Za svaki element definira se referenca ranije kreiranog `SOIngredient` sastojka, uz definiciju u kojem je stanju, termalnom procesu i koliko takvih sastojaka očekujemo. Primjećuje se da se termalni proces ovdje definira po sastojku iako igra trenutno ne podržava više različitih termalnih vrijednosti po jednoj instrukciji. Razlog za to je što je u ranijim verzijama igre sustav recepata bio drugačije zamišljen, ali je izmijenjen jer je igranjem utvrđeno da je takav sustav teško razumjeti. Budući da je izrada igre iterativan proces, odlučeno je da je najbolje ostaviti to svojstvo po sastojku jer je moguće da će se u nekoj od sljedećih iteracija to ponovo mijenjati.



Slika 39: Prikaz instrukcije za limunadu

Klasa `SORecipe` predstavlja jedan recept. Sastoji se od jednog ili više instanci `SOInstruction` klase. Osim toga ima samo jednu varijablu `inOrder` koja definira trebaju li instrukcije biti u ispravnom redoslijedu ili ne. Implementacija klasa kompleksnija je jer definira metodu `IsMatched` koja provjerava je li `FoodPlate` primljen kao

parametar. Ispis 47 prikazuje na koji se način radi provjera ako se radi o tipu recepta kod kojeg redoslijed nije bitan. U prvom dijelu provjerava se da broj instrukcija objekta `plate` odgovara broju instrukcija recepta. Nakon toga prolazi se kroz svaku instrukciju recepta. Ranije je spomenuto da je u `FoodPlate` klasi najveći broj instrukcija 64 jer `matchedFlag` varijabla popisuje koje su instrukcije `FoodPlate` objekta već verificirane. Nad svakom instrukcijom `plate` objekta poziva se `IsMatched` metoda koja je definirana u `SOInstruction` klasi. Definicija metode vidljiva je na ispisu 47. U metodi za svaki `IngredientState` zbraja se koliko ima sastojaka, kojeg su istog tipa, procesa i termalne obrade. Ako se očekivan broj poklapa s navedenim brojem `IngredientState` instance, nastavlja se provjera. Na kraju se mora provjeriti da je ukupan broj elemenata primljen kao parametar, jednak zbroju svih `count` varijabla svih `IngredientState` instanci u toj instrukciji. Ako ta metoda prođe, postavlja se zastava u 1 na mjestu na kojem se nalazi ta instrukcija u `IngredientOrder` varijabli `plate` objekta. Tako je moguće imati i više istih instrukcija u receptu. Ovaj način provjere pokriva rubne slučajeve kojih ima mnogo.

```
1 reference | TiTi, 68 days ago | 1 author, 2 changes
private bool MatchUnordered(FoodPlate plate)
{
    if (Instructions.Count != plate.IngredientOrder.Count())
        return false;

    long matchedFlag = 0;

    foreach (var instruction in Instructions)
    {
        bool instructionMatchFound = false;

        for (int i = 0; i < plate.IngredientOrder.Count(); i++)
        {
            if (((matchedFlag & (1L << i)) == 0) && instruction.IsMatched(plate.IngredientOrder.ElementAt(i)))
```

SOInstruction.cs

```
41
42      public bool IsMatched(IEnumerable<Ingredient> plateIngredients)
43      {
44          foreach (var instructionIngredient in Ingredients)
45          {
46              var nPlateIngredients = plateIngredients.Count(x => instructionIngredient.IsEqual(x));
47              if (nPlateIngredients != instructionIngredient.count)
48                  return false;
49          }
50          long nIngredientsTotal = Ingredients.Sum(x => x.count);
51          return nIngredientsTotal == plateIngredients.Count();
52      }
```

Ln: 42 Ch: 17 SPC CRLF

```
            {
                matchedFlag |= 1L << i;
                instructionMatchFound = true;
                break;
            }
        }

        if(!instructionMatchFound)
        return false;
    }

    return true;
}
```

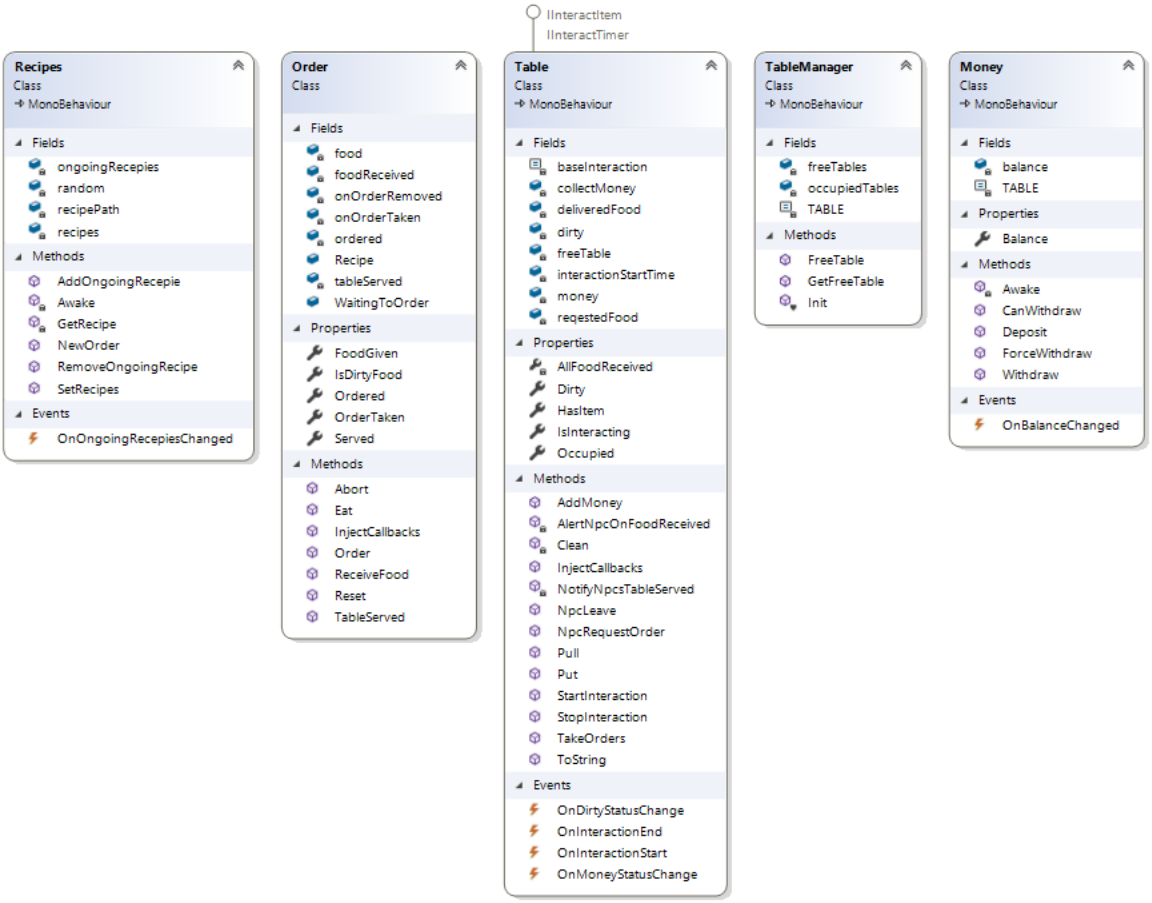
Ispis 47: Verifikacija `FoodPlate` instance klase

`Recipes` je `MonoBehaviour` klasa zadužena za vođenje liste svih tekućih recepata na sceni i dodjeljivanja novih recepata i novih narudžbi agentima. Može ju se inicijalizirati pozivom metode `SetRecipes` koja prima listu recepata dostupnih za tu razinu. Ako to nije napravljeno, instanca klase `Recipes` će u Unity `Awake` poruci učitati sve dostupne recepte iz zadane putanje za recepte. Kako je klasa zadužena i za davanje instanca `Order` klase agentima, tako je zadužena i za dodavanje (engl. *inject*) dvije metode `Order` klasi. Dvije metode koje klasa `Recipes` prosljeđuje klasi `Order` jesu `Add` i `Remove` `OngoingReceipe` jer se kod kreiranja novog agenta recept ne dodaje odmah na listu recepata koju igrač treba izvršiti. Recept se dodaje jednom kad igrač upita agenta koju narudžbu želi. Kako je cilj projekta bio i uredan kod vođen SOLID principima, tako je to izvedeno na način da `Order` klasa ne zna ništa o implementaciji metoda `OnOrderTaken`, `OnOrderRemoved`, već ih je samo dužna pozvati ovisno o situaciji `Order` instance. Način dodavanja i korištenja jednog od *callback* metoda može se vidjeti na ispisu 48. *Callback* su metode one koje nisu implementirane u klasi već su dobivene kao parametar.

```
private static Action<SORecipe> onOrderTaken;
private static Action<SORecipe> onOrderRemoved;

1 reference | TiTi, 182 days ago | 1 author, 1 change
public static void InjectCallbacks(Action<SORecipe> ongoingRecepieAdded, Action<SORecipe> ongoingRecepieRemoved)
{
    onOrderTaken = ongoingRecepieAdded;
    onOrderRemoved = ongoingRecepieRemoved;
}

1 reference | TiTi, 193 days ago | 1 author, 1 change
public void Abort() => onOrderRemoved(Recipe);
```

Ispis 48: Postavljanje i korištenje statičkih *callback* metoda klase `Order`

Callback su metode statične jer su iste za svaku instancu klase `Order`, stoga kod `Awake` Unity poruke klasa `Recipes` poziva statičku metodu `InjectCallbacks` za postavljanje *callback* metoda. Na slici 40 prikazane su ostale klase zadužene za interakciju igrača s agentom.



Slika 40: Klase za interakciju između agenta i igrača

Uloga je `Order` klase pratiti podatke vezane uz narudžbu, a podaci su sljedeći:

- recept koji je naručen
- vrijeme otkad agent ček za prihvat narudžbe
- vrijeme kad je igrač zaprimio narudžbu
- vrijeme kad je hrana primljena
- hrana koja je primljena
- podatak o tome je li hrana primljena u čistoj ili šporkoj posudi
- obavijest o tome da su svi agenti za istim stolom dobili svoje jelo.

Ostale metode u ovoj klasi koriste se najčešće iz `Table` klase koja je obrađena u daljnjem tekstu. Uloga je `Table` klase služiti kao interakcija između korisnika i agenta, tj. kao stol. Klasa je implementirana tako da više agenta može doći za isti stol. Jednom kad agent dođe do stola, pozvat će `NpcRequestOrder` metodu koja će registrirati u `Order` klasi vrijeme kad je agent spreman naručiti hranu i dodati ga u mapu agent – narudžba. `NpcLeave` se

poziva ako agent odluči napustiti stol iz bilo kojeg razloga. Pozivom metode `AddMoney` agent ostavlja novac na stolu.

Klasa `Table` implementira `IinteractItem` i `InteractTimer` za interakciju s igračem. `IInteractItem` je implementiran tako da igrač može dodavati Unity objekte koji imaju `FoodPlate` komponentu na sebi. Prilikom uzimanja objekta sa stola, uzimaju se objekti u FIFO stilu. `IinteractItem` radi ovisno o tome u kojoj se fazi stol nalazi. Ako je stol čisti i postoje agenti za stolom, igrač interakcijom pita agente narudžbu koji će onda, putem *callback* metoda i događaja, obavijestiti sve zainteresirane stranke. Ako je stol prljav, igrač čisti stol i prikuplja novac, ako se isti nalazi na stolu. Tada će se pozvati određeni događaji i *callback* metode s kojima će igra registrirati i prikazati na UI elementu da je igrač pokupio novac. Vizualno se stol mijenja i prikazuje se čistim. Pozivom `FreeTable` *callback* metode iz `TableManager` klase, stol se vraća kao čist. Na slici 40 vidi se koje sve ostale metode i događaje implementira klasa `Table`.

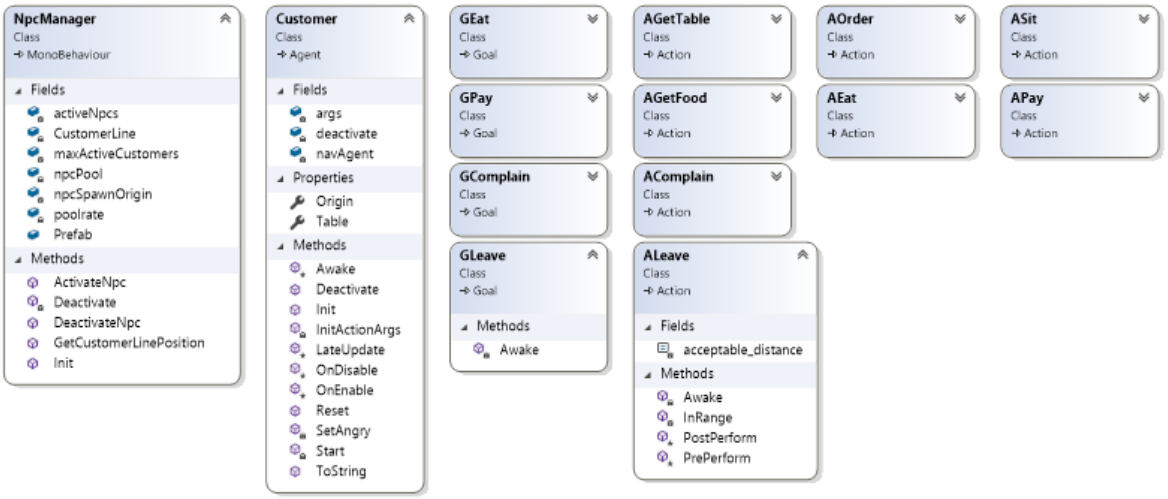
Uloga je `TableManager` klase upravljanje stolovima. Pozivom `Init` metode klasa pronalazi sve Unity objekte na trenutnoj sceni s tagom „Table“. Od svakog objekta dohvatit će komponentu `Table` i dodati ga u listu slobodnih stolova. Preostale dvije metode `GetFreeTable` i `FreeTable` služe za oslobađanje očišćenog stola ili dohvaćanje čistog stola koji nije zauzet.

`Money` predstavlja novčani račun za plaćanje u igri. Definira sljedeće metode i svojstva za interakciju:

- `Balance` – daje podatak o trenutnom stanju novca.
- `CanWithdraw` – ispituje može li se željena količina novca podignuti. Ako je željena količina veća od `Balance` svojstva, novac nije moguće podignuti.
- `Deposit` – dodavanje novca na račun.
- `Withdraw` – uzimanje novca s računa, samo ako je to moguće, tj. ako je `Balance` veći ili jednak od zatraženog iznosa.
- `ForceWithdraw` – prisilno uzimanje novca, mogućnost da `Balance` bude negativan.

4. 2. 4. Implementacija agenata

Slika 41 predstavlja klase za implementaciju agenata pomoću GOAP paketa. U projektu je implementirana jedna vrsta agenta, a to je potrošač. Svi definirani ciljevi na slici 41 počinju s prefiksom G, a sve akcije počinju s prefiksom A.



Slika 41: Klase za implementaciju i upravljanje agentima

`NPCManager` zadužen je za upravljanje umjetnom inteligencijom, tj. agentima na sceni. Metodom `Init` određuje se maksimalni broj agenata na sceni u danom trenutku kao i svako koliko će vremena `NPCManager` instanca klase provjeravati treba li dodati novog agenta na scenu. Klasa implementira *object pool pattern* s automatskim proširenjem. To je način kodiranja koji ne uništava instance objekta kad ih više ne koristi, nego ih sprema kao neaktivne u nekakav kontejner. Jednom kad opet budu trebali, umjesto kreiranja novih, koristit će se oni neaktivni koji su ranije bili spremljeni u kontejner. Automatsko proširenje znači da duljina kontejnera nije ograničena već će se, ako ne postoji nijedan slobodni agent u kontejneru, kreirati novi. Ispis 49 prikazuje kako su implementirane metode te klase. Metode su kratke i jednostavne za čitati, pa nije potrebno dodatno elaborirati implementaciju. Zbog vizualnih i optimizacijskih razloga `DeactivateNpc` je implementirana kao `coroutina` koja gasi agenta nakon jedne sekunde.

```
0 references | TiTi, 182 days ago | 1 author, 3 changes
public void ActivateNpc()
{
    if (npcPool.Any())
    {
        var npc = npcPool.Dequeue();
        activeNpcs.Add(npc);

        npc.Reset();
    }
    else if ( activeNpcs.Count < maxActiveCustomers)
    {
        var npcGo = Instantiate(Prefab, npcSpawnOrigin.transform);
        var npc = npcGo.GetComponent<Customer>();

        npc.Init(DeactivateNpc);
        activeNpcs.Add(npc);
    }
}
1 reference | TiTi, 182 days ago | 1 author, 1 change
public void DeactivateNpc(Customer npc)
{
    StartCoroutine("Deactivate", npc);
}

0 references | TiTi, 182 days ago | 1 author, 1 change
private IEnumerator Deactivate(Customer npc)
{
    yield return new WaitForSeconds(1f);
    npc.gameObject.SetActive(false);
    activeNpcs.Remove(npc);
    npcPool.Enqueue(npc);
}

1 reference | TiTi, 182 days ago | 1 author, 1 change
public void Init(uint maxActiveCustomers, float poolrate)
{
    this.maxActiveCustomers = maxActiveCustomers;
    this.poolrate = poolrate;
    InvokeRepeating("ActivateNpc", 0.1f, poolrate);
}
```

Ispis 49: Implementacija klase `NPCManager`

`Customer` klasa predstavlja naručitelja i koristi implementirani GOAP paket. Paket je implementiran na način kako je opisano u poglavlju o korištenju GOAP paketa. Argumenti koje `Customer` šalje kroz metodu `Execute` GOAP paketa su:

- instanca naručitelja
- instanca `NavMeshAgent` klase koja pokreće naručitelja
- prazna mjesta za narudžbu i stol.

`NavMeshAgent` je Unity klasa koju je potrebno dodati kao komponentu Unity objektu kako bi se mogao iskoristiti Unity paket za navigaciju na sceni. Unity navigacijski sustav omogućuje izgradnju sustava za pametnu navigaciju agenta ovisno o svojstvima agenta. [20] Na slici 42 plavom bojom označena je putanja kojom agenti mogu hodati. Putanja se kreira nakon definiranja parametara područja, agenata i prepreka na sceni. Svaki Unity objekt koji

je statičan i prepreka na sceni moraju se definirati kao navigacijski statični objekti. Dok god se agent kreira na lokaciji plave putanje i dok se točka do koje ide nalazi na putanji, navigacijski će sustav doći do te točke, pritom izbjegavajući statičke i dinamičke prepreke, poput drugih agenata.

Slika 42: Unity navigacijski sustav

Ispis 50 u metodi `PrePerform` prikazuje kako se koristi `NavMeshAgent` za dolazak agenta do mjesta gdje agent čeka slobodan stol. Jednom kad agent zadovolji minimalnu udaljenost od zadanog objekta, prelazi u sljedeću fazu internog FSM-a akcije.

```
public static System.Func<Table> GetTable;
public static Vector3 LinePosition;
private const float acceptable_distance = 2f;

Unity Message | 0 references | TiTi, 182 days ago | 1 author, 1 change
private void Awake()
{
    Abortable = true;
    Effects.Add(Conditions.HasTable);
}

1 reference | TiTi, 182 days ago | 1 author, 1 change
private bool InRange(float currentDistance) => currentDistance <= acceptable_distance;

0 references | TiTi, 182 days ago | 1 author, 1 change
protected override void PrePerform(ActionArgs args)
{
    var cArgs = (CustomerActionArgs)args;

    var distanceToExit = Vector3.Distance(cArgs.Customer.transform.position, LinePosition);

    if (InRange(distanceToExit))
        Transit();

    cArgs.NavAgent.destination = LinePosition;
}

0 references | TiTi, 182 days ago | 1 author, 1 change
protected override void Perform(ActionArgs args)
{
    var table = GetTable();
    if (table)
    {
        Abortable = false;
        var cArgs = args as CustomerActionArgs;
        cArgs.Table = table;
        Transit();
    }
}
```

Ispis 50: Implementacija `AGetTable` GOAP akcije

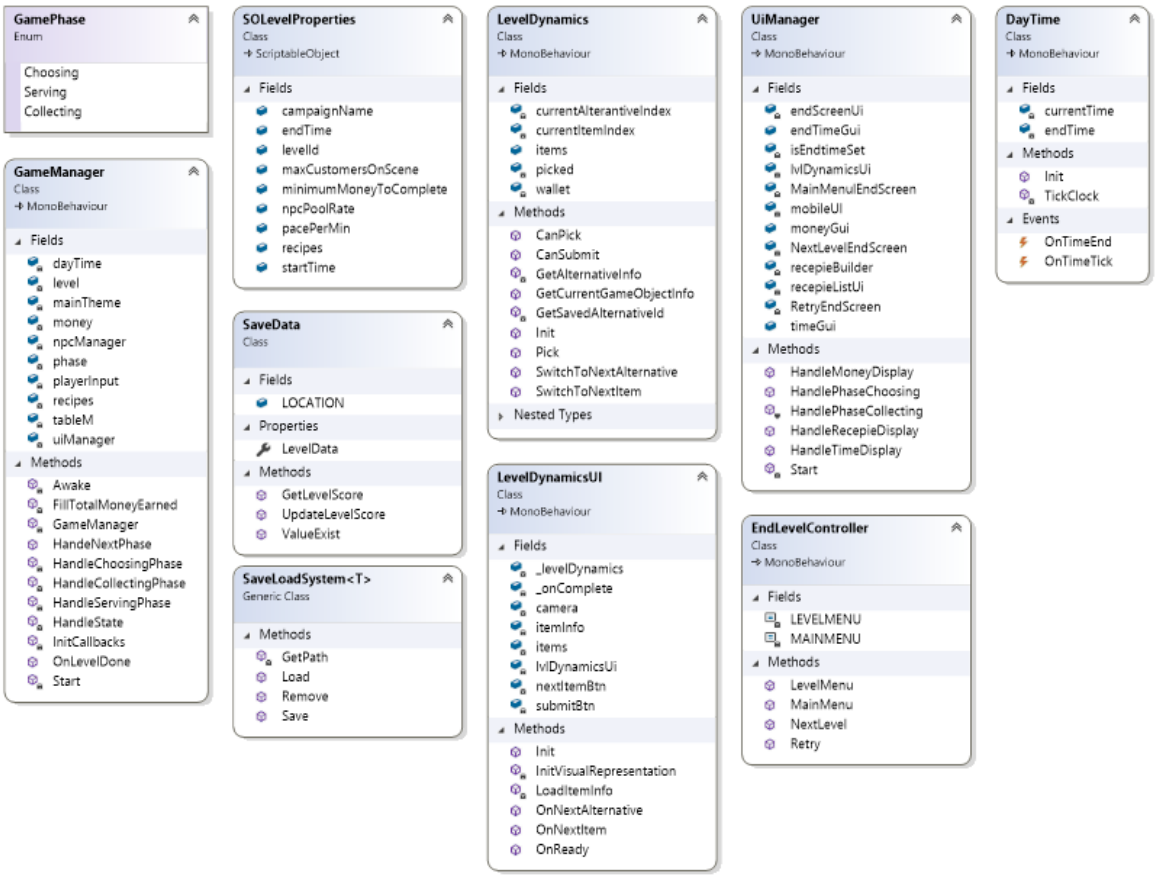
U `Perform` fazi poziva se *callback* metoda `GetTable` koja je definirana u `TableManageru`. U tom stanju agent čeka dok ne dobije stol ili dok ne prekine čekanje. Vidljivo je da je u Unity `Awake` poruci postavljen efekt za ovu akciju kao i svojstvo Abortable u `true`. Ostale akcije koje su prikazane na slici 41 neće biti detaljno objašnjenje jer je konceptualno kroz ovu akciju naveden stvaran primjer implementacije. Za definirane akcije moguće je jasno iz imena klase razaznati koja je svrha te akcije. Međutim, na ispisu 51 prikazana je jedna zanimljiva implementacija `Perform` metode u klasi `AEat`. Implementirano je da, ako igrač dostavi agentu hranu u posudi koja je prljava, on neće biti zadovoljan dodavanjem tog svojstva agentu. Ako je dostavljena posuda čista, simulira se da agent jede neko vrijeme nakon čega je on zadovoljan.

```
0 references | TiTi, 26 days ago | 1 author, 4 changes
protected override void Perform(ActionArgs args)
{
    if (cArgs.Order.IsDirtyFood)
    {
        cArgs.Customer.AddCondition(Conditions.IsUnsatisfied);
        Completed = true;
        return;
    }

    if (startTime == null)
    {
        startTime = Time.time;
    }
    else if (Time.time - startTime > waitTime)
    {
        cArgs.Customer.AddCondition(Conditions.IsSatisfied);
        cArgs.Order.Eat();
        Transit();
    }
}
```

Ispis 51: Implementacija metode `Perform` `AEat` GOAP akcije

4. 2. 5. Upravljanje

Slika 43 prikazuje na koji se način upravlja igrom unutar scene. Najvažnija je klasa `GameManager` koja je zadužena upravljati ostalim upravljačkim klasama kao što je `UiManager`, `NPCManager` i druge. Također joj je uloga da prati i pokreće različite faze igre na pojedinoj razini. Faze igre po razini definirane su `GamePhase enumom`, a `Choosing` faza ne postoji kod prve razine kampanje.



Slika 43: Upravljačke klase pojedine scene

Ispis 52 prikazuje dohvaćanje neke od komponenti koje inicira `GameManager` klasa. Vidljivo je da se u `Awake` Unity poruci dohvaćaju potrebne komponente s Unity objekta.

U metodi `Start` dodjeljuju se potrebne *callback* metode, tako se, npr. klasi AGetTable dodjeljuje *callback* metoda za dohvat slobodnog stola. Nakon toga iz instance klase `SOLevelProperties` dohvaćaju se recepti dostupni za trenutnu razinu i prosljeđuju se `Recepies` klasi. Iza slijedi pretplata na razne događaje između klasa i, prije nego što se započne upravljati stanjem razine, dodjeljuje se pravilni `PlateVisualizer` koji se brine za vizualnu reprezentaciju `FoodPlate` Unity objekta. Tako je moguće imati drugačiju vizualnu reprezentaciju za iste recepte u odnosu na kampanju i razinu.

```
Unity Message | 0 references | TiTi, 69 days ago | 1 author, 5 changes
private void Awake()
{
    recipes = GetComponent<Recipes>();
    npcManager = GetComponent<NpcManager>();
    money = GetComponent<Money>();
    dayTime = GetComponent<DayTime>();
    tableM = GetComponent<TableManager>();
    mainTheme = GetComponent<AudioSource>();
}

Unity Message | 0 references | TiTi, 69 days ago | 1 author, 5 changes
private void Start()
{
    Debug.Log("GAME MANAGER START");
    InitCallbacks();
    recipes.SetRecipes(level.recipes);
    recipes.OnOngoingRecepiesChanged += uiManager.HandleRecepieDisplay;
    money.OnBalanceChanged += uiManager.HandleMoneyDisplay;
    dayTime.OnTimeTick += uiManager.HandleTimeDisplay;
    dayTime.OnTimeEnd += OnLevelDone;

    PlateVisualizer.visualMapper = Resources.Load<SOPlateVisualizer>($"{PlateVisualizer.baseFolder}/{level.campaignName}");

    HandleState();
}

1 reference | TiTi, 182 days ago | 1 author, 1 change
private void InitCallbacks()
{
    Table.InjectCallbacks(tableM.FreeTable, money.Deposit);
    AGetTable.GetTable = tableM.GetFreeTable;
    AGetTable.LinePosition = npcManager.GetCustomerLinePosition();
    AOrder.GetOrder = recipes.NewOrder;
}
```

Ispis 52: Implementacija `Awake` i `Start` Unity metode u `GameManager` klasi

Sljedeći ispis 53 prikazuje na koji način `GameManager` upravlja pojedinom razinom. Na slici se vidi da će se faza odabira preskočiti ako komponenta `LevelDynamics` nije dodijeljena Unity objektu na kojem se nalazi `GameManager` klasa. Faza je odabira kad igrač, novcem zarađenim u prethodnim razinama kampanje, kupuje dodatnu ili unaprjeđuje postojeću opremu. Ako postoji takav objekt, onemogućuje se kretanje igrača i, koristeći implementiranu generičku klasu za snimanje i učitavanje podataka iz datoteke, dohvaćaju se `SaveData` podaci.

`SaveData` je klasa za snimanje maksimalno zarađenog novca po razini. Implementira tri metode: `GetLevelScore`, `UpdateLevelScore` i `ValueExist` za lakšu interakciju sa snimljenim podacima. Ako datoteka postoji, poziva se lokalna metoda `FillTotalMoneyEarned` koja će zbrojiti sve novce zarađene u toj kampanji do trenutne razine i dodati iznos u novčanik. Novčanik je instanca `Money` klase koja se potom prosljeđuje `LevelDynamics` klasi. Na kraju `GameManager` naređuje `UiManager` klasi da započne s pokretanjem faze odabira pomoću parametra `onComplete` koji je tipa metode bez parametara, tj. prosljeđuje se anonimna metoda u kojoj se pokreće sljedeća faza igre i postavlja se vrijednost u novčaniku na 0.

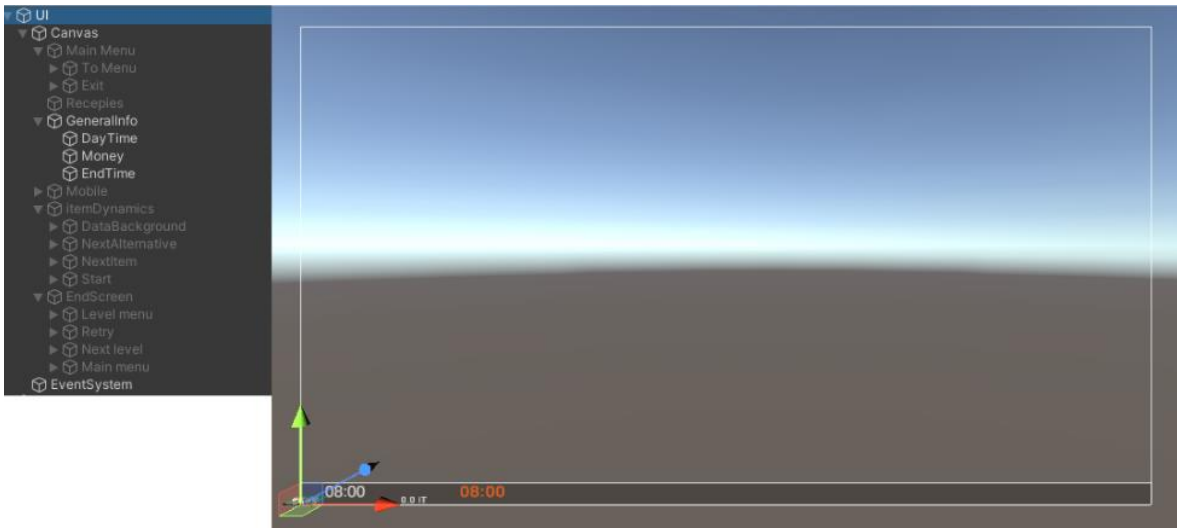
Sljedeća je faza serviranje. U ovoj fazi pokreće se pozadinska glazba, omogućava se igraču kretanje i iniciraju se instance klasa `NPCManager`, `TableManager` i `DayTime` kojima se prosljeđuju potrebni podaci iz `SOLevelProperties` instance klase.
Zadnja je faza skupljanja u kojoj se snima postignuti uspjeh ako je to prvi put da se ta razina igrala ili ako je ostvareni rezultat bolji od prethodno ostvarenog rezultata. Nakon toga `GameManager` naređuje `UiManager` instanci klase da započne s pokretanjem faze skupljanja koja će, u odnosu na postignuti uspjeh, prikazati drugačiji UI igraču.

```
private void HandleChoosingPhase()
{
    var lvlDynamics = GetComponent<LevelDynamics>();

    if (!lvlDynamics)
    {
        Debug.LogWarning("LevelDynamics not found on GameManager, handling next phase.");
        uiManager.HandlePhaseChoosing(default,
            () =>
            {
                HandeNextPhase();
            });
        return;
    }


    SaveLoadSystem<SaveData> saveLoadSys = new SaveLoadSystem<SaveData>();
    var saveData = saveLoadSys.Load(SaveData.LOCATION);

    if (saveData != null)
        FillTotalMoneyEarned(saveData);

    lvlDynamics.Init(money);

    playerInput.HandleActions = false;

    uiManager.HandlePhaseChoosing(lvlDynamics,
        () =>
        {
            HandeNextPhase();
            money.ForceWithdraw(money.Balance);
        });
}
```

```
private void HandleServingPhase()
{
    if (mainTheme)
        mainTheme.Play();

    playerInput.HandleActions = true;

    tableM.Init();
    dayTime.Init(level.startTime, level.endTime, level.pacePerMin);
    npcManager.Init(level.maxCustomersOnScene, level.npcPoolRate);
}

1 reference | TiTi, 4 days ago | 1 author, 4 changes
private void HandleCollectingPhase()
{
    playerInput.HandleActions = false;

    SaveLoadSystem<SaveData> saveLoadSys = new SaveLoadSystem<SaveData>();
    var saveData = saveLoadSys.Load(SaveData.LOCATION);

    if (saveData == null)
        saveData = new SaveData();

    var highestScore = saveData.GetLevelScore(level.campaignName, level.levelId);

    if (money.Balance > highestScore)
    {
        highestScore = money.Balance;
        saveData.UpdateLevelScore(level.campaignName, level.levelId, money.Balance);
        saveLoadSys.Save(saveData, SaveData.LOCATION);
    }

    uiManager.HandlePhaseCollecting(highestScore, money.Balance, level.minimumMoneyToComplete);
}
```

Ispis 53: Implementacija faza za razinu

Klasa `UiManager` glavna je klasa za prikaz UI elemenata igraču prilikom igranja neke razine. Na slici 26 za svaki UI element na sceni, osim za elemente koje implementiraju `IInteractTimer` sučelja, `UiManager` je direktno ili indirektno odgovoran. U poglavlju implementacije GOAP sustava spomenuto je da Unity trenutno podržava tri različita UI sustava i da je za implementaciju igre odabran Unity UI (uGUI) sustav jer ostali sustavi nisu preporučljivi ili dovršeni da bi ih se koristilo kao UI za igru. Slika 44 prikazuje izgled UI sustava na sceni u Unity *editoru* i prikazuje neke od elemenata od kojih se UI sastoji.



Slika 44: *head-up* zaslon igrača s UI Unity elementima koji ga implementiraju

`UiManager` upravlja elementima sa slike i mijenja im svojstva kako bi igraču prikazao potrebne podatke na zaslonu. Većina je elemenata isključena te ih `UiManager` uključuje i popunjava na određeni događaj. UI Unity element „Mobile“ sa slike prikazuje se samo kod faze posluživanja i ako je igra izgrađena (engl. *build*) za mobilne uređaje. Za prikaz elementa

`UiManager` koristi dvije pomoćne klase `RecepieBuilderUI` i `LevelDynamicsUI`. `RecepiesBuilderUI` zadužen je za dinamičko kreiranje recepata koji se mogu vidjeti na slikama 26 i 27. `UiManager` poziva javnu metodu `HandleReceipes` `RecepieBuilderUI` klase kojoj prosljeđuje argumente dobivene od događaja `OnOngoingRecepiesChanged` Recepies klase. Ispis 54 prikazuje na koji se način kreiraju novi Unity UI elementi i popunjavaju kako bi prikazali recept na zaslonu igraču.

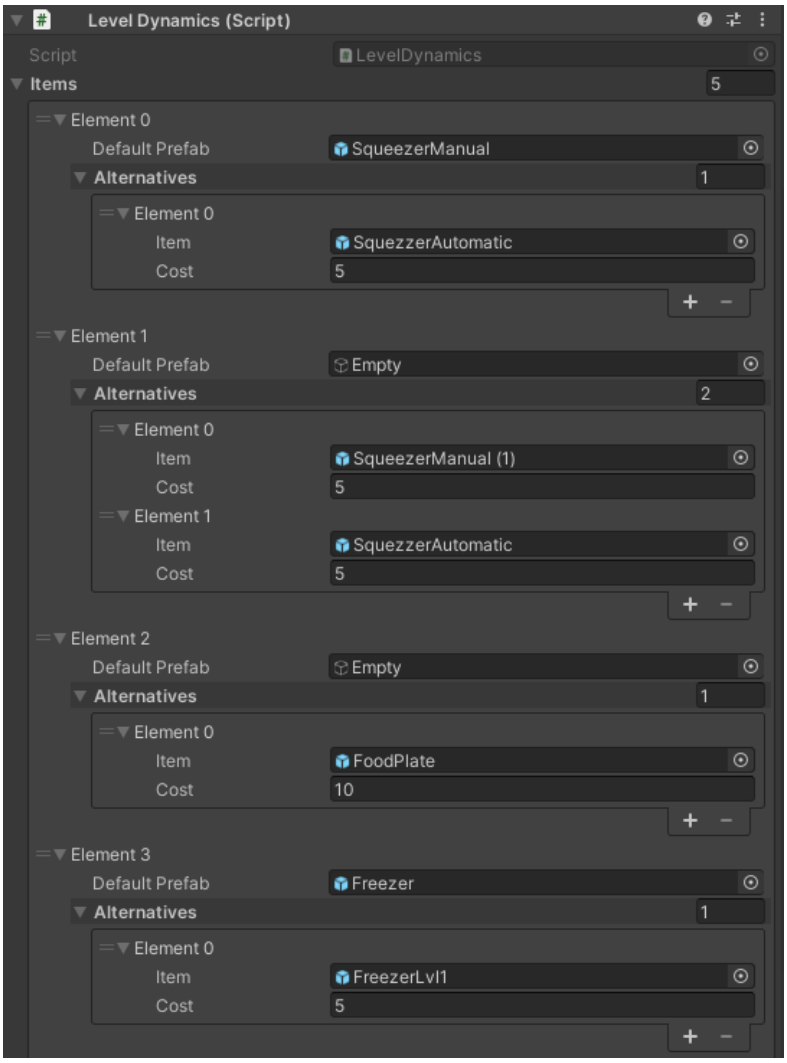
```
public void HandleReceipes(RecepiesUiEventArgs args)
{
    args.RemovedRecepies.ForEach(recipe =>
    {
        var goToHide = recipeConnections.Where(y => y.recipe == recipe && y.visual.activeSelf).FirstOrDefault();
        if (!goToHide.Equals(default))
            goToHide.visual.SetActive(false);
    });

    args.AddedRecepies.ForEach(recipe => {
        var goToPopulate = recipeConnections.Where(y => y.recipe == recipe && !y.visual.activeSelf).FirstOrDefault();
        if (goToPopulate.Equals(default))
        {
            var go = Instantiate(recepieTempate, ongoingRecepiesList);
            goToPopulate = (recipe, go);
            recipeConnections.Add(goToPopulate);
            Populate(goToPopulate);
        }

        goToPopulate.visual.SetActive(true);
        goToPopulate.visual.transform.SetAsLastSibling();
    });
}
public void Populate((SORecipe recipe, GameObject visual) recipeConnection)
{
    SetReceipeWidth(recipeConnection.visual, recipeConnection.recipe);

  for (int i = 0; i < recipeConnection.recipe.Instructions.Count; i++)
    {
        var recipeInstruction = GetRecipeInstruction(recipeConnection.visual, i);
        recipeInstruction.SetActive(true);

        var foodProcessParent = GetFoodProcessing(recipeInstruction, iFoodProcess);
        var foodTypeParent = GetFoodProcessing(recipeInstruction, iFoodType);
        var foodCountParent = GetFoodProcessing(recipeInstruction, iFoodCount);

        var ingredients = recipeConnection.recipe.Instructions[i].Ingredients;

        for (int j = 0; j < ingredients.Count; j++)
        {
            var foodProcessImage = foodProcessParent.transform.GetChild(j).GetComponent<Image>();

            foodProcessImage.sprite = processVariants
                .Where(x => x.process == ingredients[j].homogeneity)
                .FirstOrDefault()?.image;

            var foodTypeImage = foodTypeParent.transform.GetChild(j).GetComponent<Image>();
            foodTypeImage.sprite = ingredients[j].type.Image;

            var foodCountText = foodCountParent.transform.GetChild(j).GetComponent<TextMeshProUGUI>();
            foodCountText.text = $"x {ingredients[j].count}";
        }

        GetFoodTreatment(recipeInstruction).sprite = treatmentVariants
                .Where(x => x.treatment == ingredients[0].treatment)
                .FirstOrDefault()?.image;
    }
}
```

Ispis 54: Dinamičko kreiranje prikaza Unity UI recepta

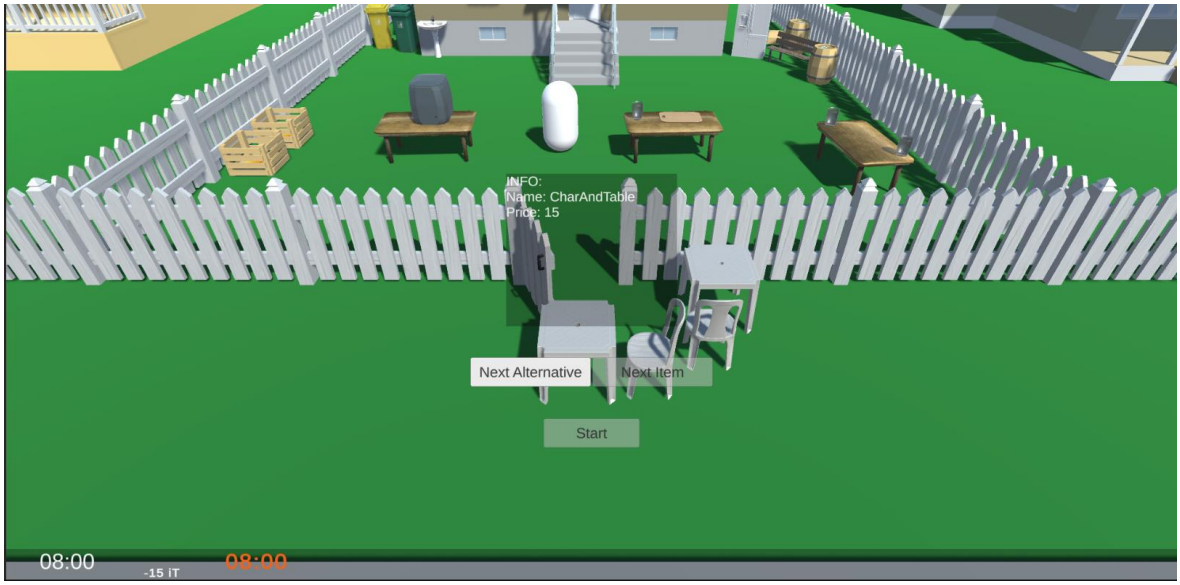
U metodi `HandleReceipes` može se vidjeti da se stari kreirani recepti ne brišu, već ih se deaktivira kako bi se smanjio utjecaj GC-a (*garbage collector*). Kod aktiviranja recepta pretražuje se ta ista mapa u potrazi za identičnim neaktivnim receptom. Ako je pronađen, aktivira ga se i dodaje na kraj liste. Ako je potrebno kreirati novi recept, tada se poziva metoda `Populate` koja na osnovu `SoRecipe` i `GameObject` klase popunjava `GameObject` tako da dodaje potrebnu djecu i elemente te spaja potrebne slike i tekst kako

je prikazano na slici 44. Zbog moguće kompleksnosti recepata i ovaj je sustav sastavljanja ponešto složeniji kako bi sustav mogao podržati sastavljanje svih podržanih tipova recepata.

Klase `LevelDynamics` i `LevelDynamicsUI` zadužene su za upravljanje fazom odabira. `LevelDynamics` zadužen je za podatkovni dio upravljanja, a on se upravlja i poziva iz `GameManager` klase, dok je `LevelDynamicsUI` zadužen za *head-up* zaslon, tj. grafički prikaz informacija igraču kao i za upravljanje kamerom u fazi odabira. `LevelDynamicsUI` upravljan je od `UiManager` klase. Korisnik `LevelDynamics` klase treba je u Unity *editoru* popuniti broj elemenata za koje želi imati alternative, zatim je za svaki element potrebno popuniti koji je osnovni element, koji može biti i prazan, a koji su alternativni. Za svaki alternativni element navodi se i cijena. Slika 45 prikazuje popunjavanje klase u Unity editoru, a slika 46 prikazuje kako igrač vidi fazu odabira. Na slici 46 vidi se da nije omogućen početak igre jer je podatkovni dio obavijestio grafički da je igrač ušao u minus s novcem. Nije moguće ići dalje dok igrač ne odabere neku alternativu za koju ima dovoljno novca.



Slika 45: `LevelDynamics` u Unity *editoru*



Slika 46: Prikaz faze odabira kako ju vidi igrač

Ulazak u detalje svake implementirane klase bio bi podugačak i nepotreban. Stoga će ostale klase biti navedene u tablici 10. Klase čija je svrha audio-vizualna reprezentacija uz pomoć pretplate na neki tip sučelja ili događaja, kao i klase koje nasljeđuju od `EventArgs` i `ActionArgs`, neće biti prikazane u tablici jer je njihova svrha očita.

Tablica 10: Kratak opis svrhe preostalih bitnijih klasa

KLASA	OPIS
`Conditions`	Statička klasa sa svim definiranim stanjima.
`DayTime`	Simulacija vremena u igri s mogućnošću odabira početka i kraja vremena te brzine protoka vremena. Implementira metode i događaje prikazane na slici 43.
`EndLevelController`	Klasa koja implementira metode za rukovanje scenama. Koristi se kao `Click` događaj na Unity UI kontrolama za dugmad (engl. *button*).
`IngredientStorage`	Simulira skladište sastojaka s ograničenom količinom.
`LevelMenuController`	Služi za upravljanje scenom za odabir kampanje i razine.
`PeriodicInstantiator`	Pomoćna klasa za periodično stvaranje zadanog objekta u zadanom vremenu na zadanoj lokaciji.
`MainMenuController`	Upravlja izbornikom za pauzu.
`SaveLoadSystem`	Generička klasa za snimanje bilo koje klase koja se može *serializirati*. Za sada se koristi samo za snimanje `SaveData` objekta koji sprema podatke o kampanji, razini i maksimalnoj količini zarađenog novca po razini.
`ScreenShortDescription`	Klasa koja koristi IMGUI Unity UI sustav za prikaz osnovnih informacija o proizvoljnom objektu. Koristi se za *debugging* svrhe.
`SOLevelProperties`	Konfiguracijska klasa za pojedinu razinu.
`Toilet`	Klasa implementirana za sljedeću kampanju s kafićima gdje će se igrač morati brinuti i o zahodima. Nije dio ovog rada.

`TrafficSimulation`	Generička simulacija prometa s velikim izborom parametara.
`CloudMovementSimulator`	Simulacija oblaka na početnom zaslonu.

Implementacija novih razina uz pomoć implementiranih klasa s pogleda dizajnera znatno je olakšana. Primjerice, za definiranje novog kuhinjskog uređaja potrebno je definirati novu instancu klase `SOKitchenAppliance` ako se želi da kuhinjski uređaj ima drugačije karakteristike od prethodno implementiranih. Nakon toga potrebno je u Unity sceni stvoriti novi Unity objekt te nadodati na njega potrebne komponente i za kraj ga snimiti kao Unity *prefab* kako bi se mogao koristiti i izvan scene na kojoj je kreiran. Na sličan se način definiraju i novi kuhinjski ručni alati, automatski alati, uređaji i slično.

Za recepte, ovisno o receptu, potrebno je kreirati nove instance pripadnih SO tipova klasa za sastojke, instrukcije i na kraju sve zajedno spojiti u recept. Na ovaj je način moguće kreiranje novih razina i kampanja bez ulaženja u kod. Naravno, ako se žele drugačiji agenti, potrebno je definirati nove akcije i ciljeve koristeći GOAP paket, ili ako se žele nove funkcionalnosti kao spomenuta klasa `Toilet`, potrebno je to isprogramirati.

Iako je mnogo vremena uloženo, dizajn razine, razne prilagodbe i konfiguracije u Unityju, neće biti detaljno opisane u ovom radu jer to nije fokus ovog rada.Ipak, potrebno je naglasiti da je većina fontova, zvukova i najviše 3D modela i tekstura preuzeto s Unity *asset* trgovine i konfigurirano za potrebe projekta. Unity *asset store* jest trgovina s koje se mogu kupovati ili skidati besplatno, ovisno o licenci, razni dodaci za Unity *editor*, 3D modeli, zvukovi, teksture, materijali. animacije, fontovi i sve potrebno za izradu igre. [21] Neka moguća proširenja razmotrena su u zaključku.

5. Zaključak

U ovom je radu detaljno opisana implementacija algoritma ponašanja GOAP i implementacija prototipa igre koristeći GOAP algoritam ponašanja kao Unity paket. Cilj je bio prikazati prepreke na koje nailazi razvojni inženjer prilikom razvoja igre ili proširenja za njih.

Implementirani Unity paket može se povući i iskoristiti za druge Unity projekte u kojima GOAP, kao algoritam ponašanja, odgovara. Paket je dobro pokriven testovima te time pruža sigurnost potencijalnim korisnicima. Proširivanje paketa novim mogućnostima znatno je olakšano zbog pokrivenosti testovima. U budućnosti bi se svi dijelovi algoritma, osim agenta, mogli odvojiti u dodatnu biblioteku. Tako bi se algoritam mogao koristiti u bilo kojem projektu. Zatim bi današnji paket trebao povlačiti tu biblioteku i služiti kao omotač biblioteka specijalizirana za Unity.

Igra je već testirana od nekolicine ljudi i neke su se promjene implementirale u igru. Razvoj je igre iterativan i dugotrajan proces, a trenutna je igra daleko od gotovog proizvoda. Sljedeći koraci bili bi vizualno poboljšavanje u smislu da se igraču jasnije intuitivno da do znanja što i kako raditi. Tu su i razna zamišljena proširenja, od jednostavnijih, poput novih tipova agenata, recepta i uređaja, do složenijih poput implementiranja *multiplayer* mehanike, mehanike pomoćnih radnika, mehanike za prestiž restorana, *editora* razina i mnoga druga proširenja.

Literatura

1. About Unity, https://unity.com/our-company (pristupljeno 28. 6. 2021.)
2. Unity wiki https://en.wikipedia.org/wiki/Unity_(game_engine) (pristupljeno 29. 6. 2021.)
3. Git wiki https://en.wikipedia.org/wiki/Git (pristupljeno 30. 6. 2021.)
4. What is git https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F (pristupljeno 30. 6. 2021.)
5. F.E.A.R. Review https://www.gamespot.com/reviews/fear-review/1900-6135744/ (pristupljeno 1. 7. 2021.)
6. A.I. of F.E.A.R. https://alumni.media.mit.edu/~jorkin/gdc2006_orkin_jeff_fear.pdf (pristupljeno 11. 7. 2021)
7. Unity packages https://docs.unity3d.com/Manual/CustomPackages.html (pristupljeno 10. 7. 2021.)
8. Unity UI sustavi https://docs.unity3d.com/2020.3/Documentation/Manual/UI-system-compare.html (pristupljeno 18. 7. 2021.)
9. UI Toolkit https://docs.unity3d.com/2021.2/Documentation/Manual/UIElements.html (pristupljeno 18. 7. 2021.)
10. UXML https://docs.unity3d.com/2021.2/Documentation/Manual/UIE-UXML.html (pristupljeno 18. 7. 2021.)
11. USS https://docs.unity3d.com/2021.2/Documentation/Manual/UIE-USS.html (pristupljeno 18. 7. 2021.)
12. UQuery https://docs.unity3d.com/2021.2/Documentation/Manual/UIE-UQuery.html (pristupljeno 18. 7. 2021.)
13. Git submodul https://git-scm.com/book/en/v2/Git-Tools-Submodules (pristupljeno 18. 7. 2021.)
14. Unity Unit test framework https://docs.unity3d.com/Packages/com.unity.test-framework@1.1/manual/index.html (pristupljeno 18. 7. 2021.)
15. Unit test, Editor vs play mode https://docs.unity3d.com/Packages/com.unity.test-framework@1.1/manual/edit-mode-vs-play-mode-tests.html (pristupljeno 18. 7. 2021.)
16. Keep a changelog https://keepachangelog.com/en/1.0.0/ (pristupljeno 20. 7. 2021.)

17. Unity prefab https://docs.unity3d.com/Manual/Prefabs.html (pristupljeno 18. 8. 2021.)
18. Scriptable object https://docs.unity3d.com/Manual/class-ScriptableObject.html (pristupljeno 22. 8. 2021.)
19. Flyweight desing pattern https://refactoring.guru/design-patterns/flyweight (pristupljeno 22. 8. 2021.)
20. Unity navigation and pathfinding https://docs.unity3d.com/Manual/Navigation.html (pristupljeno 23. 8. 2021.)
21. Unity asset store https://assetstore.unity.com/ (pristupljeno 25. 8. 2021.)