Sveučilište Jurja Dobrile u Puli
Fakultet Informatike u Puli

JOSIP MARIĆ

IMPLEMENTACIJA ARHITEKTONSKOG UZORKA ENTITETSKO-KOMPONENTNOG SUSTAVA ZA RAZVOJ IGARA I WEB APLIKACIJA

Diplomski rad

Pula, rujan, 2021. godine

Sveučilište Jurja Dobrile u Puli
Fakultet Informatike u Puli

JOSIP MARIĆ

IMPLEMENTACIJA ARHITEKTONSKOG UZORKA ENTITETSKO-KOMPONENTNOG SUSTAVA ZA RAZVOJ IGARA I WEB APLIKACIJA

Diplomski rad

JMBAG: 0303068646, redoviti student
Studijski smjer: Diplomski sveučilišni studij Informatika
Kolegij: Napredni algoritmi i strukture podataka
Znanstveno područje: Društvene znanosti
Znanstveno polje: Informacijske i komunikacijske znanosti
Znanstvena grana: Informacijski sustavi i informatologija
Mentor: izv. prof. dr. sc. Tihomir Orehovački

Pula, rujan, 2021. godine

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisan, Josip Marić, kandidat za magistra informatike ovime izjavljujem da je ovaj Diplomski rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Diplomskog rada nije napisan na nedozvoljen način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

Josip Marić

U Puli, rujan, 2021. godine

IZJAVA

o korištenju autorskog djela

Ja, Josip Marić, dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj diplomski rad pod nazivom:

„Implementacija arhitektonskog uzorka entitetsko-komponentnog sustava za razvoj igara i web aplikacija“ koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.

Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, rujan, 2021. godine

Potpis

Josip Marić

Sažetak

Ovaj rad opisuje implementaciju arhitektonskog uzorka entitetsko-komponentnog sustava (ECS). Iskazuje se problematika objektno orijentiranog dizajna te se predlaže implementacija podatkovno orijentiranog dizajna, odnosno njegovog ECS podskupa. Implementacija ECS arhitektonskog uzorka se uspoređuje s ECS standardima te detaljno objašnjava organiziranje i strukturu elemenata entitetsko-komponentnog sustava u razvijanju igara i razvijanju frontend djela web aplikacija.

Ključne riječi: *entitetsko-komponentni sustav, ECS, podatkovno orijentiran dizajn, igra, web aplikacija.*

Abstract

Graduate thesis describes an implementation of the entity-component system (ECS) architectural pattern. The issue of object-oriented design is presented and the implementation of data-oriented design, ie its ECS subset, is proposed as a replacement. Implementation of ECS architectural pattern is compared with ECS standards and explains in detail the organization and structure of elements in entity-component systems for game development and the development of frontend part of web applications.

Keywords: *entity-component system, ECS, data oriented design, game, web application.*

Sadržaj

1. Uvod

Tema ovog diplomskog rada je izrada vlastite implementacije arhitektonskog uzorka entitetsko-komponentnog sustava (eng. entity-component system, ECS). Razvijanje vlastite implementacije je započeto pri opažanju loših performansi pri kreiranju igara za mobilne uređaje koje su izrađene objektno orijentiranim dizajnom. Kako se povećavala kompleksnost logike entiteta pojavljivao se problem koji entitet će biti izvor istine, odnosno kako standardizirati način komunikacije između entiteta bez značajnijeg utjecaja na performanse.

Prvotno je bila zamisao koristiti novi ECS sustav Unity razvojne platforme, ali kako je on u to vrijeme bio tek u alfa verziji, moguće je bilo koristiti jedino njihov hibridni sustav. Naravno, to nije bilo prihvatljivo rješenje kako bi podrška za taj hibridni sustav pri daljnjem razvitku njihovog sustava nestala. Alternativa je bila koristiti neke od paketa trećih strana, ali nedostatak kvalitetne dokumentacije i njihova uska ugrađenost sa samom Unity razvojnom platformom postavljala je opet upit o podršci s novim verzijama te razvojne platforme. Dodatno, preferirala se modularnost same implementacije te mogućnost korištenja iste u drugim razvojnim platformama poput Godot razvojne platforme i pri razvoju web aplikacija.

Pri razvijanju prioritet se dao načinu komunikacije entiteta iz perspektive sustava misija umjesto globalnim sustavima kao što je to slučaj u ostalim ECS implementacijama. Ideja je bila kreirati ECS implementaciju tako da se ne koristi koncept modela relacijskih baza pri upitima koja komponenta pripada kojem entitetu, već se različite veze komponenata-entiteta definiraju pri kompiliranje programa.

Kroz rad se iskazuju napomenuti nedostaci objektno orijentiranog dizajna koji dovode do loših performansi te se predlaže njihova izmjena s podatkovno orijentiranim dizajnom odnosno njegovim ECS podskupom. Objašnjavaju se razlike između baznog dijela vlastite ECS implementacije u usporedbi s predloženim standardom. Nadalje, kroz primjere implementacije za razvoj igara koja je pisana u C# programskom jeziku i Unity razvojnoj platformi (čime je postignut hibridni ECS) i implementacije za razvoj frontend dijela web aplikacija u TypeScript programskom jeziku i Vue progresivnom JavaScript

okviru detaljnije se opisuju dijelovi ECS implementacije, njihove prednosti i nedostaci te predlažu se poboljšanja određenih dijelova. Naposljetku, uspoređuje se implementacija s implementacijom Entitas gdje se jasno iskazuju nedostaci korištenja konceptualnog modela relacijskih baza.

2. Objašnjenje osnova ECS arhitektonskog uzorka

Entitetsko-komponentni sustav (ECS) je arhitektonski obrazac koji se uglavnom koristi u razvoju igara. ECS se bazira na principu kompozicija umjesto nasljeđivanja (eng. composition over inheritance) čime se postiže veća fleksibilnost definiranja entiteta (kod razvijanja igara to bi bili interaktivni objekti na sceni) koji se izgrađuju od individualnih dijelova u raznim kombinacijama. Takvim načinom se uklanjaju problemi dvosmislenosti dugih lanaca nasljeđivanja i promiče se čist dizajn [1].

Odvajanjem podataka i logike, ECS postiže modularni sustav koji omogućuje memorijsko pohranjivanje podataka u susjedna memorijska područja i pojednostavljuje paralelizaciju aplikacijske logike. Također modularnošću izbjegavaju se zamke objektno orijentiranog dizajna (OOD) poput dvosmislenosti kod višestrukih nasljeđivanja [2].

2.1. Povijest

Tim koji je 2007. godine radio na „Operation Flashpoint: Dragon Rising“ eksperimentirao je s ECS dizajnom u kolaboraciji s Adam Martinom koji je kasnije detaljno opisao ECS dizajn uključujući definicije temeljne terminologije i pojmova. Također, Martin je svojim radom popularizirao ideju o sustavu kao prvoklasnom elementu (eng. first-class element), entitetima kao identifikatorima, komponentama kao neobrađenim podacima i logici definiranoj u sustavima [4].

Nadalje Apple Inc. 2015. godine predstavio je GameplayKit (API okvir za razvoj igara za iOS, MacOS, TvOS) koji uključuje implementaciju ECS-a. Uključuje praktičnu podršku za integraciju s Appleovim SpriteKit, SceneKit i Xcode Scene Editor [6].

Nedavno novi korisnici ECS-a su Mojang s izdavanjem „Minecraft Windows 10 Edition“ koji koristi biblioteku EnTT pisanu u C++ i Unity tvrtka koja je 2018. godine prikazala „Megacity Demo“ koji je koristio tehnološki stok baziran na ECS-u. Imao je sto tisuća audio izvora, po jedan za svaki automobil, svaki neonski natpis i još mnogo toga - stvarajući ogromnu, složenu zvučnu kulisu [6, 7].

2.2. Karakteristike

ECS implementacija može se razmatrati kao proširenje kompozicije objektno orijentiranog programiranja, ali njena implementacija zahtjeva drukčiji način razmišljanja. Primjerice, entitete odnosno interaktivne objekte opisuje se kao identifikatore čija je jedina zadaća ujediniti komponente u jednu cjelinu. U ovom smislu komponente su jednostavne reprezentacije podatkovnih svojstava entiteta. Naposljetku, logika se obrađuje s odvojenim sustavima. Ti sustavi ne sadrže nikakve podatke, već služe samo kao crne kutije koje obrade ulazne podatke. Navedena tri elementa (entitet, komponenta i sustav) su pridruženi jedan drugome uz pomoć voditelja konteksta (eng. context manager) koji omogućuje međusobnu komunikaciju elemenata. Voditelj konteksta određuje koja će komponenta pripadati kojem entitetu i koristi se za prosljeđivanje potrebnih svojstava njihovim sustavima.

Temeljni pojmovi koji se pojavljuju kod ECS-a su [4]:

- Entitet (eng. Entity)
    - Jedinstven identifikator koji svaki objekt igre označava kao zasebnu stavku.
    - U tu svrhu često se koristi običan cijeli broj.
- Komponenta (eng. Component)
    - Svi podaci jednog djela entiteta.
    - Koriste se strukture, klase ili asocijativni red.
- Sustav (eng. System)
    - Kod jednog djela objekta koji opisuje logiku.
    - Ideja je da svaki sustav kontinuirano obrađuje sve entitete koji posjeduju komponente koji imaju zajednički dio objekta.

Dodatno kada se spominje komponenta ona može imati dva značenja u kontekstu ECS. Odnosno kao predložak u objektno orijentiranom programiranju i kao individualna instanca tog predloška.

2.3. Entitet ECS arhitekture

Smatraju se temeljnim konceptualnim gradivnim blokovima sustava. Konceptualno svaki entitet reprezentira specifičnu cjelinu koja se opisuje jedinstvenim identifikator i pridruživanjem komponenata. U idealnoj situaciji entitet ne bi trebao sadržavati nikakve podatke ili metode niti biti primjer klase. Jedina zadaća entiteta je povezivanje njemu definiranih komponenata čime se kreira specifična cjelina.

Kako je već navedeno za identifikator entiteta dovoljno je da bude pozitivni cijeli broj. Također, može se koristiti logika pri generiranju identifikatora što može pomoći pri strukturiranju podataka. Primjerice, identifikator se može koristiti kao indeks u redu komponente označavajući koji entiteti sadrže koju komponentu.

2.4. Komponenta ECS arhitekture

Komponente su podatkovni segment ECS-a koji ne sadrže logiku. To su generički tipovi višestruke upotrebe koji se koriste za definiranje svojstava entiteta i njihovu interakciju s drugim entitetima. Svaka komponenta predstavlja različiti segment entiteta koji nadodaje njegovom specifičnom identitetu.

Za primjer entitet zmaj u video igrici mogao bi sadržati sljedeće komponente:

- Leteći
- Neprijatelj
- 3D Model

Niti jedan od ovih komponenata ne sadrži logiku, već je njihova svrha da definiraju entitet svojstvima te izoliraju skupove podataka kako bi se oni mogli koristiti kao identifikatori.

2.5. Sustav ECS arhitekture

Kako bi se osigurala logika za komponente, koriste se sustavi ili procesori. Oni su agenti globalnog opsega, što znači da se za razliku od tradicionalne metode objekta mogu pozivati s bilo kojeg mjesta unutar koda. Umjesto da izravno ciljaju određeni entitet, oni su zainteresirani za aktivne komponente različitih entiteta. Komponente se dodaju u kontekst, koji se zatim prosljeđuje odgovarajućem sustavu kao argument za pružanje opsega operacije. Djelujući odvojeno izvan strukture komponenata entiteta, sustav se upari s nizom komponenata koje imaju iste dijelove kao i taj sustav i izvodi svoje interne metode na komponentama kontinuirano u pozadini, jednu po jednu. Sustavi ne vraćaju vrijednosti, već mijenjaju stanja različitih komponenata izvođenjem transformacija podataka.

2.6. Sastavljanje modela

Programiranje s ECS-om može se usporediti s programiranjem relacijskih baza podataka. Instanca entiteta djeluje slično ključu u bazi podataka kao u bilo kojem sustavu upravljanja relacijskim bazama podataka. S apstraktnog gledišta, pristup komponenti određenog entiteta nije ništa drugo nego upit nad bazom podataka.

Ideja da se komponenti pristupa pretraživanjem se može dodatno promatrati u tablici 1 gdje svaki redak predstavlja instancu entiteta, a svaki stupac različitu komponentu.

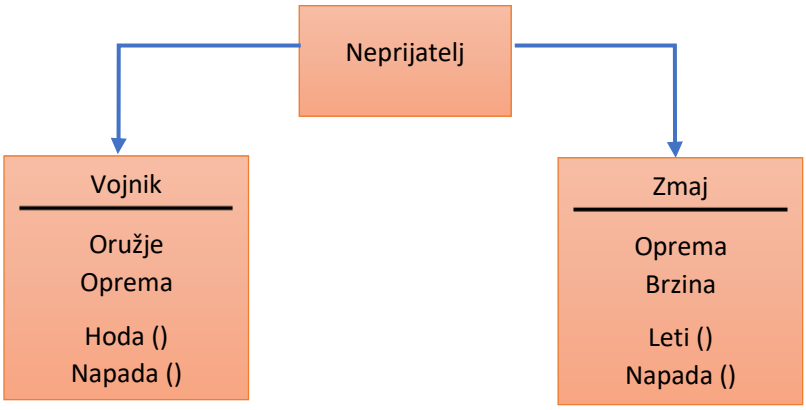
Tablica 1. Apstraktni primjer relacijske baze podataka po instanci entiteta

	Hoda	Leti	Igrač	Neprijatelj	Model3D
Vojnik	X			X	X
Zmaj		X		X	X
Igrač	X		X		X

3. Problem s objektno orijentiranim dizajnom

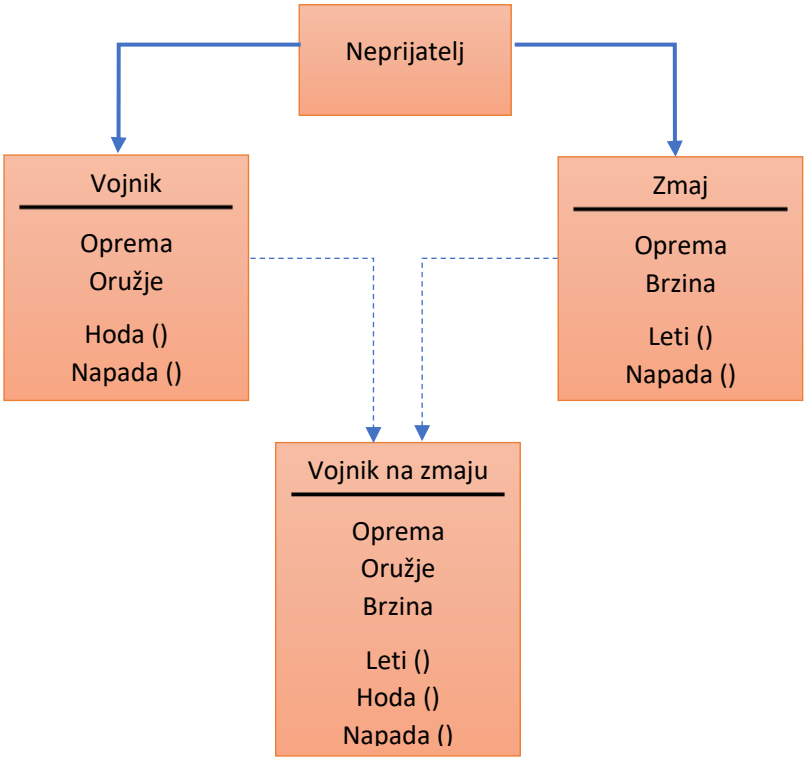
U implementaciji objektno orijentiranog dizajna (OOD), svaki se logički objekt kreira ili kao instanca klase ili kao zbirka (eng. collection) međusobno povezanih instanci klase. Hijerarhija klasa klasificirana je sustavom kriterija poznatim kao taksonomija. Na primjer, biološka taksonomija traži genetske sličnosti na osam razina: domena, carstvo, odjeljak, klasa, red, obitelj, rod i vrsta. Na svakoj razini stabla različiti tipovi živih bića odvojena su u točnije i preciznije skupine prema nekom kriteriju koji pripada toj razini [3].

Detaljnije, u situaciji implementacije OOD-a u svrhu razvijanja igara, definiranje logike i igraćih objekata korištenjem OOD-a dobiva se struktura kao na slici 1. Prikazana je nadklasa neprijatelj i dvije pod klase vojnik i zmaj. Svojstava i logika su zajedno zapakirani u jedan objekt.



Slika 1. Struktura nekih entiteta u objektno orijentiranom dizajnu

Kako se nadodavaju tipovi klasa, hijerarhija postaje sve kompleksnija i time otežava održavanje i dodavanje novih klasa pogotovo ako to nije predviđeno u dizajnu hijerarhije. Za primjer želi se dodati tip klasa vojnik na zmaju što bi izgledalo kao što je prikazano na slici 2.

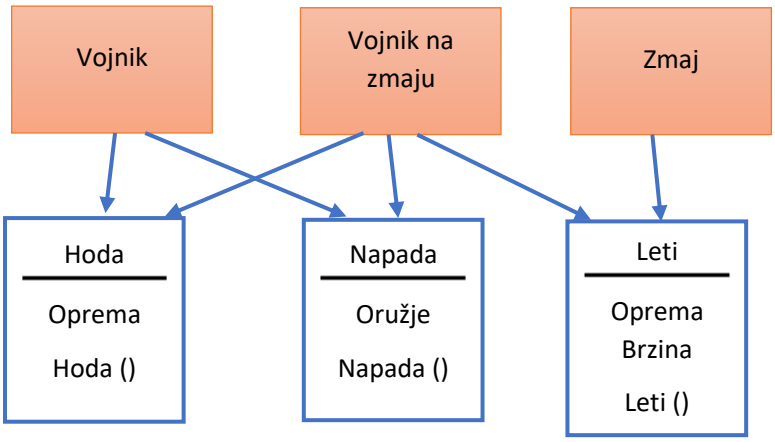


Slika 2. Struktura entiteta nakon dodavanja entiteta vojnik na zmaju

Postoje dvije opcije kako implementirati klasu vojnik na zmaju u OOD-u. Prva bi bila premjestiti sva svojstva i metode u nadklasu neprijatelj. Problem iza toga je u tome da će vojnik imati metodu leti i svojstvo oprema koji će ostati nedefinirani. Drugo rješenje je da klasa vojnik na zmaju naslijedi i zmaja i vojnika. Problem kod ovakvog pristupa je kreiranje takozvanog smrtonosnog dijamanta smrti (eng. deadly diamond of death) što dovodi do dviju kopija neprijatelja te će biti potrebno definirati referira li se na neprijatelja sa zmajeve ili vojničke strane. Ovo je načelno loša praksa te iz toga razloga novi programski jezici ne dozvoljavaju višestruko nasljeđivanje.

3.1. Objektno orijentirana kompozicija

Jedan od pristupa kako izbjeći problem dijamanta je taj da se odvoje metode i svojstva u zasebne klase (komponente) te se vojnik i zmaj tretiraju kao entiteti koji služe kao spremnik za te komponente (slika 3).



Slika 3. Struktura entiteta gdje su metode i svojstava izvučeni u zasebne klase

Ovime komponente pružaju entitetima određenu funkcionalnost te se mogu koristiti kod više entiteta. Dok je objektno orijentirana kompozicija bolje rješenje u svrhu izgradnje kompleksnog programa, još uvijek ima nedostatak kako ne podržava odjeljenje logike i podataka.

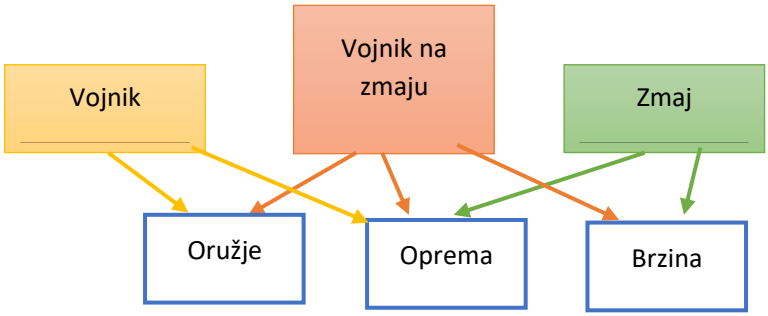
3.2. Podatkovno orijentiran dizajn

Osim samih nedosljednosti kod strukturiranja programa objektno orijentiranim dizajnom, drugi nedostatak su loše performanse izvršavanja programa. Čak i ako je izvršavanje algoritama optimizirano, ostaje problem uskog grla kod neefikasnog upravljanja memorijom. Kako bi se optimiziralo upravljanje memorijom, kao rješenje se nudi podatkovno orijentiran dizajn (eng. data oriented design, DOD), odnosno ECS koji je njegov podskup. Zamisao oko DOD-a je organiziranje metoda da obrađuju podatke na generički način i strukturirati podatke što efikasnije prema zahtjevima hardvera.

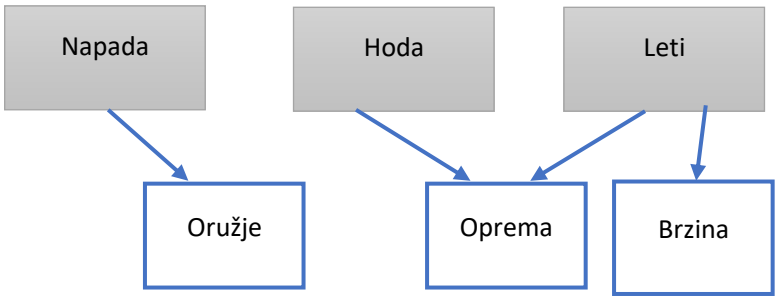
Dok se OOD uglavnom fokusira na pojedini objekt u jedno vrijeme, DOD može obraditi grupe podataka odjednom čime se pri velikim količinama podataka, koji su još i optimizirani u memoriji, postižu drastično bolje performanse. Idealno, podaci u memoriji bi bili složeni tako da se mogu obrađivati uzastopno. Primjerice, za definiranje podataka koristila bi se struktura umjesto klase iz razloga jer kod strukture pristup svim podacima je omogućen preko jednog pokazivača. DOD također omogućuje korištenje više dretveno obrađivanje podataka čime se povećava vjerojatnost pogodaka predmemorije.

Dodatna prednost korištenja DOD-a odnosno ECS-a je u njegovoj modularnosti što dozvoljava fleksibilnu hijerarhiju entiteta i pojednostavljuje serijalizaciju podataka i umrežavanje.

Prijašnji primjer razrađen preko ECS-a na slikama 4. i 5. gdje je logika odijeljena u sustave te svaki sustav i entitet zahtjeva unikatnu kombinaciju komponenata.



Slika 4. Struktura pridruženih entiteta i komponenata



Slika 5. Struktura zahtijevanih komponenata po sustavima

4. Objašnjenje baznog djela vlastite ECS implementacije

Kako trenutno ne postoji standardna implementacija ECS-a te ako korisnik ne želi koristiti neku od postojećih biblioteka zbog njihovih kompleksnosti (primjerice EnTT ima 70 000 linija koda) relativno je lagano implementirati vlastiti ECS koji će biti bolje iskrojen vlastitim potrebama. ECS implementacije se kreću od jednostavnih koje uglavnom dijele logiku i podatke do vrlo optimiziranih koji su prilagođeni samoj memoriji, vrlo efektivno koriste predmemoriju, ali im je nedostatak fragmentiranje tablica podataka.

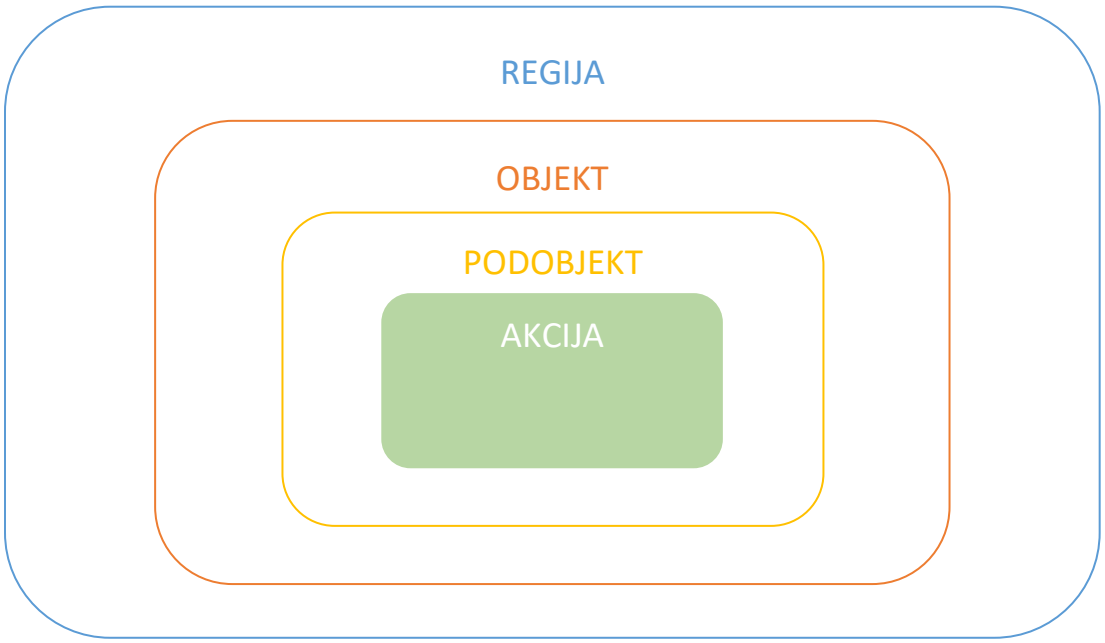
Nadalje će se opisati temeljni dijelovi ECS-a te način na koji su oni implementirani. Važno je napomenuti da je implementacija prototipna te neke od značajki nisu optimizirane, a u nastavku će biti opisano koje nedostatke treba ispraviti.

ECS dijelovi i njihova svrha:

- Entitet
    - Identifikatori entiteta definirani su uz pomoć nekoliko značajki umjesto jednog cijelog broja.
- Komponenta
    - Opisana je kao objekti koji se dodjeljuju instanci entiteta tako da se generiraju pri kreiranju novog entitetom.
- Sustav
    - Može se ih podijeliti na lokalne i globalne kao jedan podskup i na tri sloja logike kao drugi podskup. Razlika je u tome da dok su globalne statične te se pozivaju samo kod interakcije sa sustavom, lokalne mogu imati vlastitu dretvu koja izvršava određeni proces u intervalima. Dodatno, lokalne mogu imati vlastita svojstva te mogu poslužiti kao posrednik između ECS sustava i ostatka aplikacije. Oboje pripadaju prvom sloju logike. Drugi sloj logike je moguće paralelizirati, a treći se koristi kako bi se promjena nad jednom komponentom izrazila na druge komponente.

4.1. Entitet ECS arhitekture

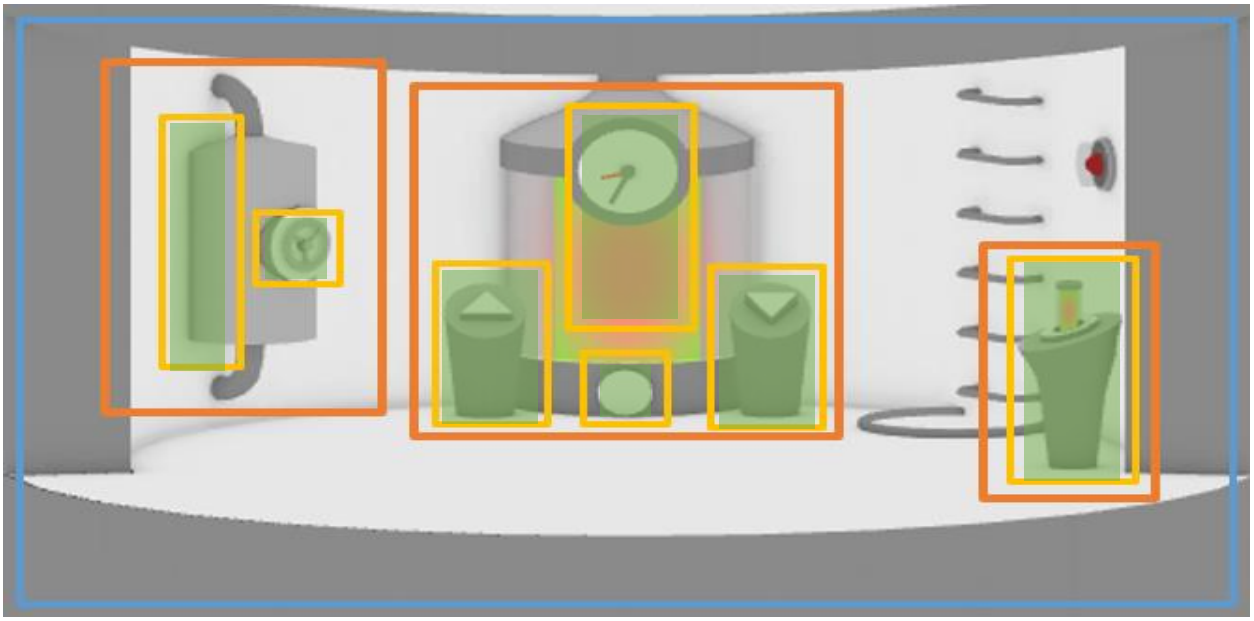
Razlog iza više identifikatora identiteta je taj da se mogu definirati bez pokretanja koda čime se olakšava njihovo promatranje u sučelju. Ti identifikatori su ugniježđeni (slika 6) počevši od gore s tipom *akcije*, *podobjektom*, *objektom* te posljednje *regijom*. Naravno, svi napomenuti identifikatori su neki cijeli brojevi koji se definiraju s enum (definirane konstante koje poprimaju brojčanu vrijednost). Na slici 7. apstraktno je vizualizirana disekcija identifikatora po svakom od interaktivnih entiteta kako se to pojavljuje u igri.



Slika 6. Hijerarhija identifikatora

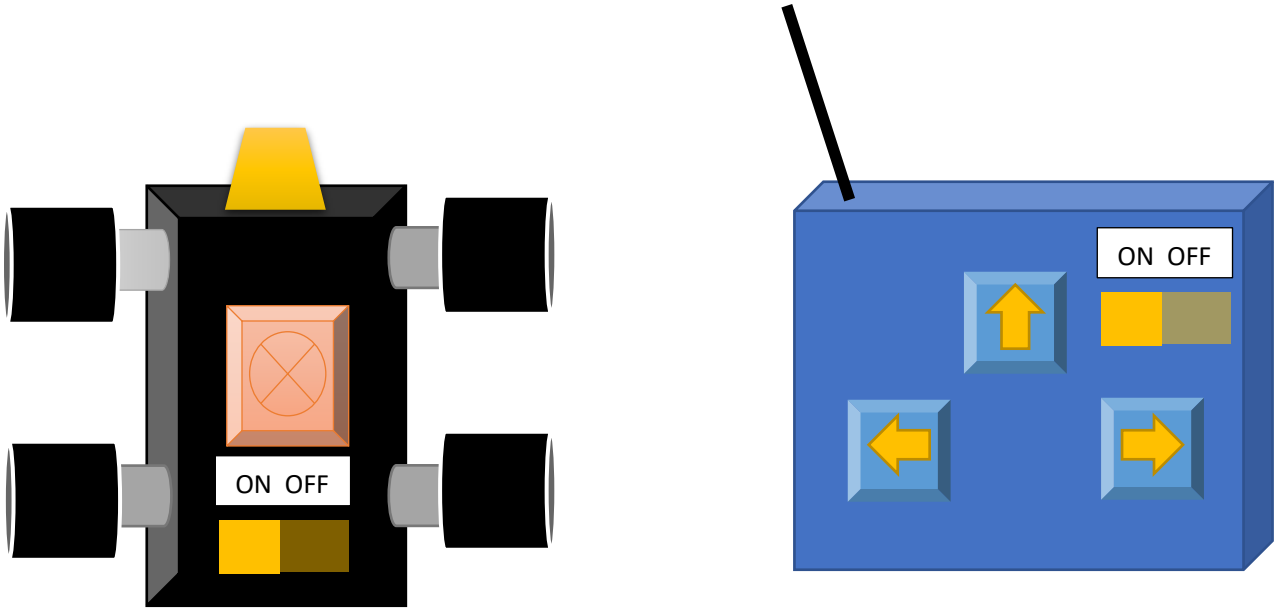
Identifikatori koji se koriste kod opisivanja entiteta su sljedeći:

- *Akcija*
    - Definira koji će se globalni sustav pokrenuti pri interakciji s njegovim entitetom.
- *Podobjekt*
    - Uz pomoć *podobjekta* moguće je rastaviti objekt na više entiteta (kao što je prikazano na slici 7.) čime se kreira entitet roditelj i njemu pripadajuća djeca. Time se proširuje broj *akcija* koji se mogu izvršiti nad roditeljem te dodatno ako ti rastavljeni entiteti imaju jednaku *akciju* moguće je raspoznati iz kojeg je entiteta inicirana *akcija* u opsegu rastavljenih entiteta.
- *Objekt*
    - Služi kao reprezentacija realnog objekta u sučelju ili stvarnosti te kao grupacija *podobjekata*. Također moguće je definirati koje od *akcija* i *podobjekata* su kompatibilni s određenim objektom.
- *Regija*
    - Kako je nemoguće raspoznati između istih tipova *objekata* u opsegu globalnih sustava bez neke interakcije nad njima, izolira ih se u *regije*.



Slika 7. Isječak iz igre na kojem se vidi približna disekcija entiteta po identifikatorima (po bojama iz slike 6)

Kroz sljedeći primjer objasnit će se prednosti ovakvog rasporeda, trenutno se zanemaruju komponente. Za primjer, koristit će se RC auto na daljinski s time da RC auto i daljinski reprezentiraju dva entiteta s identifikatorima (slika 8.). U oba slučaja *regija* entiteta bi se koristila kako bi se mogli upariti daljinski i auto i kako bi se izbjegla situacija da se jednim daljinskim upravlja s dva RC auta u slučaju da su u dometu dva RC auta. Sljedeće, definira se tip *objekta* koji bi bio RC auto s *podobjektima* motor i roditelj. Motor bi imao *akciju* vozi dok bi roditelj imao *akciju* upali/ugasi. U slučaju daljinskog tip *objekta* bi bio daljinski, a *podobjekti* bi bili tipka naprijed, tipka desno, tipka lijevo i roditelj. Za tipke *akcija* bi bila vozi, a za roditelja *akcija* bi bila upali/ugasi.



Slika 8. Prikaz objekata čiji se dijelovi promatraju kao *podobjekti*.

Trenutno ovo su samo definirani entiteti bez sustava ili komponenata, ali moguće je deducirati kako bi se odvijala komunikacija između dva objekta. Ovime su se kreirala šest entiteta, onoliko koliko ima *podobjekata*.

4.2. Komponenta ECS arhitekture

Za razliku od entiteta i sustava, komponente su ostale u načelu kao i u standardu. Pridružuju se entitetima ovisno o tipu *podobjekta*. Drugim riječima, moguće je svakom podtipu omogućiti da pridruži komponente entitetu, no to je često nepotrebno. Primjerice, u slučaju gdje postoji jedan tip *objekta* i nekoliko *podobjekata* dovoljno je te čak preporučljivo da se pridruže komponente samo na tip roditelja *podobjekta*. Nadalje, kako bi komponente komunicirale sa sustavima koristi se „obrazac promatrača“ (eng. observer pattern).

Vraćanjem na prošli primjer sada se mogu implementirati komponente. Sve komponente se dodaju na roditelje (RC auto i daljinski). Prvo će se dodati komponenta aktivan koja će se označavati sa 0 ili 1 i njih će zasada mijenjati *akcija* upali/ugasi, dok će se kasnije to odvijati preko sustava. Nakon toga se dodaje komponenta smjer sa -1 što će biti lijevo, 0 ravno i 1 desno, a još se dodaje komponenta gas sa 0 ili 1. Komponente smjer i gas će se pozivati *akcijom* vozi.

4.3. Sustav ECS arhitekture

Dok kod standardnog pristupa gdje svaki sustav konstantno u intervalima provjerava ima li aktivnih komponenata, u ovoj implementaciji svi globalni sustavi su isključivo na obrascu promatrača. To je izvedeno tako da pri generiranju entiteta oni se pretplate (kod korištenja *podobjekata* to čini samo roditelj) na događaj (eng. event) koji će signalizirati komponentama promjenu pri nekoj od interaktivnih radnji korisnika ili sustava. Pod interaktivne radnje se misli primjerice klikom na miš u slučaju korisnika ili u slučaju lokalnih sustava izlazak dretve iz suspenzije (simulira tikove).

Sustavi se dijele na lokalne i globalne čime se misli na sam pristup sustavima koji je u oba slučaja indirektan, sami sustavi su uvijek globalno dostupni, ali pristup njima je isključivo preko *akcija* ili mehanike gdje je *akcija* globalna, a mehanika lokalna. *Akciju* se definira identifikatorom i ona se poziva pri interakciji nad entitetom dok je moguće i mehaniku pozvati pri interakciji nad entitetom (to se postiže tako da se kao i u slučaju komponente tip *objekta* pretplati na mehaniku), njena korist je u tome što može izvršavati određenu logiku u intervalima.

4.4. Konceptualna razlika od predloženog standarda

Dok se predložena standardna implementacija koristi modelom relacijske tablice (u konceptualnom smislu) nad kojom se vrši upit za pronalaženje entiteta preko njegovih komponenti, u ovoj implementaciji to je kao i kod identifikatora definirano unaprijed te nije potrebno pretraživati parove entiteta-komponenata tijekom izvođenja programa.

To je izvedeno tako da se definiraju grupe. U slučaju prošlog primjera mogla bi se definirati grupa upali/ugasi s definiranim identifikatorima tipova *objekata* (daljinski i RC auto) koja bi se pozvala pri interakciji sa sklopkom odnosno pri pozivanju *akcije* upali/ugasi te bi se tada izvršila promjena nad komponentom aktivan. Analogno za *akciju* vozi definirat će se grupa smjer i grupa gas.

Implementacijom na ovakav način moguće je bezbrižno dodavati entitete RC auto i daljinski. Jedino treba pripaziti da se u slučaju premještanja entiteta u drugu *regiju* ili brisanja otkažu pretplate (što nije toliko nužno osim u slučaju masovnog premještanja).

4.5. Nedostaci

Glavni nedostatak ove implementacije je njena neistestiranost na velikim kompleksnim projektima. Također, trebalo bi ispraviti nekoliko početnih grešaka poput spremanja instance komponenata u instanci entiteta tako da se kreira polje (u nekom statičnom obliku) u koje će se spremati sve komponente entiteta te će se u instanci entiteta dodati još jedan identifikator koji će označavati poziciju u tom polju. Nadalje, kako se ne koristi manager konteksta, nego su se upiti za pronalazak para entiteta komponenata zamijenili s apstrakcijama odnosno polimorfizmom. Srodno tome, kako je u nekim slučajevima potrebno implementirati različite tipove samog entiteta što otvara potrebu za kastingom (eng. casting) tijekom izvođenja programa što utječe na performanse te bi iz tog razloga bilo poželjno pronaći rješenje koje eliminira tu potrebu.

Ostale nedostatke teško je definirati s obzirom da se ova implementacija nastavlja razvijati te nije došla u doticaj sa svim mogućim zahtjevima koji se mogu pojaviti.

5. Implementacija ECS arhitekture za programiranje igara

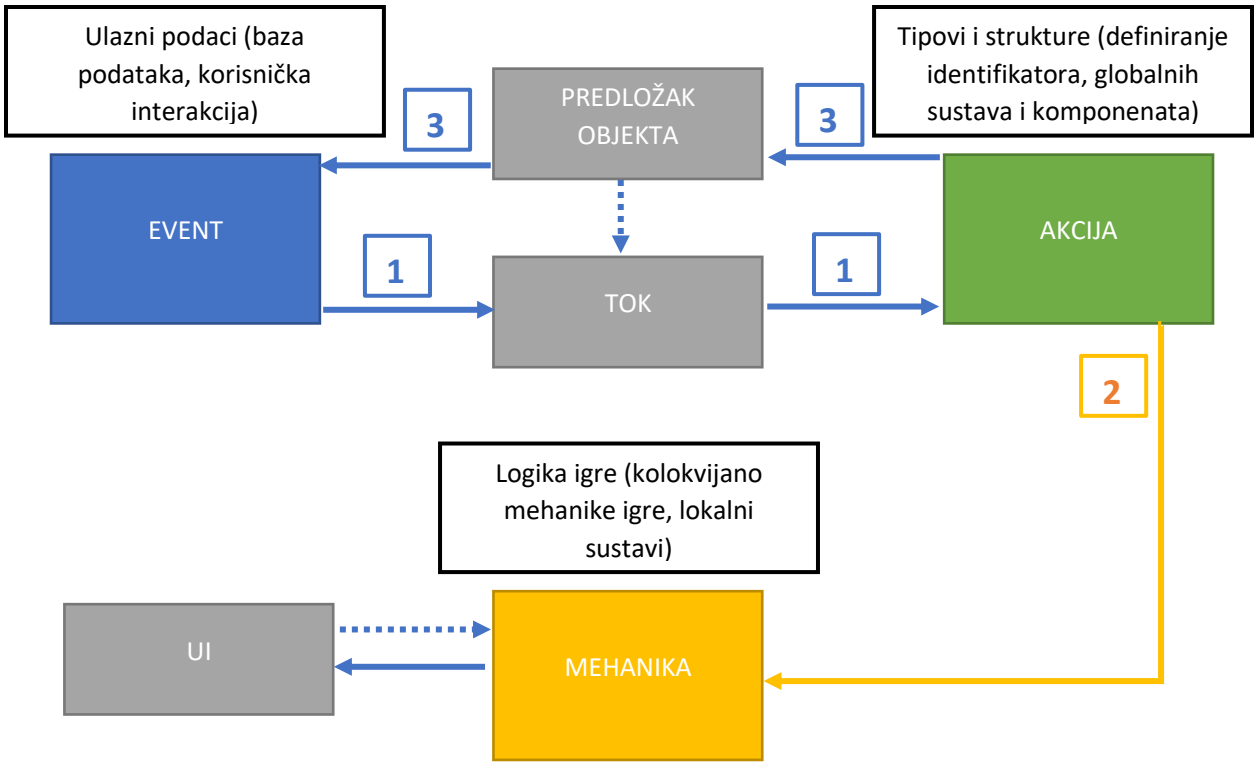
U ovom poglavlju opisat će se implementacija ECS-a koja je složena u programskom jeziku C# te se koristi u Unity razvojnoj platformi. Na bazni dio implementacije dodane su dodatne komponente iz razloga jer u opseg ove implementacije ulaze isključivo dijelovi koji nisu dio razvojne platforme te se koriste dijelovi razvojne platforme koji su pisani nasljeđivanjem (MonoBehavior) od kojih će se koristiti osnovne komponente poput transform[1] i collider[2] čime se dobiva neki tip hibridnog ECS-a. Ukratko, za te komponente nisu pisani sustavi, već se koriste tradicionalne metode.

5.1. Opširno objašnjenje izvođenja implementacije

Na slici 9. prikazano je apstraktno izvođenje igre (prikazano je samo izvođenje pri interakciji korisnika s igrom), odnosno ECS-a i njemu ovisnih dijelova programa. Cijeli ovaj dio pripada menadžer sloju programa (isključujući Object Template modul). Takozvani sustav misija (eng. quest system) se nalazi pod modulom Event, on služi kao sloj između SQLite baze i ECS-a. Kad Event modul primi signal pri pokretanju scene, na određeni objekt u igri (definiran *akcijom*) postavlja se delegat koji će se pozvati pri izvršenju *akcije* nad tim objektom, koji može biti interakcija korisnika ili poziv iz nekog sustava.

Po koracima prikazanih na slici 9. gdje se prikazuje put izvođenja pri interakciji korisnika s interaktivnim objektima, ovisno o identifikatorima entiteta (objekta) izvršava se neka od *akcija*. *Akcije* se pune u *stream*[3] modul koji se sastoji od jednostruke vezane liste gdje se *akcije* izvršavaju jedna nakon druge. U drugom koraku, koji je opcionalan, moguće je utjecati na izvršavanje mehanike preko identifikatora *podobjekt* što bi se koristilo za interakciju korisnika s vanjskim sustavima (poput korisničkog sučelja). Izvršenjem *akcije* pokreće se treći korak koji kod postavljenog delegata (ako se izvršila definirana *akcija*) signalizira Event modulu učitavanje novog cilja (u sklopu sustava misija) i signalizira *stream* modulu nastavak rada ako ima *akcija* u njegovom redu čekanja.

[1] hrv. transformacija, sadrži poziciju, rotaciju i skalu svakog objekta u igri [8]
[2] hrv. kolizija, definiraju oblik za korištenje kod fizičkih sudara [8]
[3] hrv. tok, služi kao privremeni spremnik korisničkih unosa



Slika 9. Apstraktni prikaz izvođenja igre pri interakciji korisnika

5.2. Opis implementacije entiteta ECS-a

Temeljeno na baznom dijelu kreiranje entiteta započinje se s definiranjem identifikatora. Predlošci identifikatora se u principu sastoje od enum-a (hrv. pobrojane vrijednosti), uzorak kreativnog dizajna (eng. creational design pattern) od kojih se koristi tvornička metoda (eng. factory method) ili tvornica apstrakcije (eng. abstract factory) te trenutno u prototipu i jedinstveni objekt (eng. singleton).

Ukratko o identifikatorima, njihovom korištenju u sklopu ove implementacije:

- *Regija*
  - Izolira entitete s istim identifikatorima tipa *objekta*.
- *Objekt*
  - Opisuje grupu kojoj *podobjekti* pripadaju.
- *Podobjekt*
  - Omogućuje dodavanje dodatnih *akcija* entitetu.
- *Akcija*
  - Definira koji globalni sustav će se izvršiti.

5.2.1. Identifikator tipa *regija*

Kako što je to navedeno prvo se definira enum (slika 10.). U ovom primjeru postoji samo jedna *regija*. Sljedeće se otvara apstraktna klasa s rječnikom (eng. dictionary) i javnom metodom koja ga postavlja (slika 11.) te metoda pretplati (eng. subscribe). Svi dijelovi apstraktne klase su također apstraktni.

```
15 references
public enum RegionEnum { InsideRocket };
4 references
public abstract class RegionAbstract
{
    1 reference
    protected abstract Dictionary<object, object> SetObjects();
    public abstract Dictionary<InteractableObjectTypeEnum,
        3 references
        InteractableObjectTypeAbstract> Objects { get; protected set; }

    2 references
    internal abstract void Subscribe(InteractableObjectTypeEnum objectType,
        SubObjectTypeEnum subObjectType, ConditionDel conditionDel,
        StatChangeDel damageDel);
}
```

Slika 10. Definiranje tvorničke metode za definiranje *regija*

Za pristupanje instanci *regije* koristi se statični rječnik. Jednaka primjena će se primijetiti kod svih identifikatora. Razlog iza korištenja statičnih svojstava će biti objašnjen pri kreiranju entiteta u editoru.

```
36 references
public sealed partial class ObjectTypes
{
    4 references
    public static Dictionary<RegionEnum, RegionAbstract> RegionTypes
    { get; private set; }
        = new Dictionary<RegionEnum, RegionAbstract>()
        {
            {RegionEnum.InsideRocket, new InsideRocket()}
        };

    1 reference
    public static void InitializeRegionTypes()
    {
        RegionTypes = new Dictionary<RegionEnum, RegionAbstract>()
        {
            {RegionEnum.InsideRocket, new InsideRocket()}
        };
    }
```

Slika 11. Definiranje rječnika koji grupira enum *regije* i definiciju *regije*

Nakon toga se kreira nova klasa koja nasljeđuje od regionalne apstraktne klase te se definiraju apstraktni elementi (slika 12.). Treba napomenuti kako su *regije* dodane kasnije u razvijanju implementacije te kako se koristi samo jedna *regija,* njihova utilizacija ne dolazi do izražaja u ovom primjeru.

```
2 references
public sealed class InsideRocket : RegionAbstract
{
    1 reference
    protected override Dictionary<object, object> SetObjects()
    {
        return new Dictionary<object, object>(){
            { System.Enum.GetName(typeof(InteractableObjectTypeEnum),
            InteractableObjectTypeEnum.Computer),InteractableObjectTypeEnum.Computer },
            { System.Enum.GetName(typeof(InteractableObjectTypeEnum),
            InteractableObjectTypeEnum.Stasis),InteractableObjectTypeEnum.Stasis },
            { System.Enum.GetName(typeof(InteractableObjectTypeEnum),
            InteractableObjectTypeEnum.Gas),InteractableObjectTypeEnum.Gas },
            { System.Enum.GetName(typeof(InteractableObjectTypeEnum),
            InteractableObjectTypeEnum.Charger),InteractableObjectTypeEnum.Charger },
            { System.Enum.GetName(typeof(InteractableObjectTypeEnum),
            InteractableObjectTypeEnum.Clock),InteractableObjectTypeEnum.Clock },
            { System.Enum.GetName(typeof(InteractableObjectTypeEnum),
            InteractableObjectTypeEnum.Reactor),InteractableObjectTypeEnum.Reactor }
        };
    }

    2 references
    internal override void Subscribe(InteractableObjectTypeEnum objectType,
        SubObjectTypeEnum subObjectType, ConditionDel conditionDel, StatChangeDel damageDel)
    {
        Objects[objectType].Subscribe(subObjectType, conditionDel, damageDel);
    }

    //Not meant to be used this way, but since Regions were added
    //later and this game uses only one scene it is fine.
    public override Dictionary<InteractableObjectTypeEnum,
        3 references
        InteractableObjectTypeAbstract> Objects
    { get; protected set; } =
        new Dictionary<InteractableObjectTypeEnum, InteractableObjectTypeAbstract>()
    {
        {InteractableObjectTypeEnum.Computer,
                InteractableObjectTypes[InteractableObjectTypeEnum.Computer]},
        {InteractableObjectTypeEnum.Stasis,
                InteractableObjectTypes[InteractableObjectTypeEnum.Stasis]},
        {InteractableObjectTypeEnum.Gas,
                InteractableObjectTypes[InteractableObjectTypeEnum.Gas]},
        {InteractableObjectTypeEnum.Charger,
                InteractableObjectTypes[InteractableObjectTypeEnum.Charger]},
        {InteractableObjectTypeEnum.Clock,
                InteractableObjectTypes[InteractableObjectTypeEnum.Clock]},
        {InteractableObjectTypeEnum.Reactor,
                InteractableObjectTypes[InteractableObjectTypeEnum.Reactor]}
    };
}
```

Slika 12. Definiranje *regije* inside rocket (hrv. unutar rakete)

5.2.2. Identifikator tipa *objekt*

Ponovno, prvo se definiraju enum-i (slika 14.), u ovom slučaju ima ih nekoliko, ali u primjeru će se prikazati implementacija samo jednoga. Novitet je delegat, odnosno event u apstraktnoj definiciji. Taj event nam služi kao mogućnost komuniciranja entiteta s mehanikom. Sljedeće je pretplati uvjete (eng. subscribe conditions) koja je jedna od elemenata same igre, to je dio sustava misija. Nadalje, za razliku od identifikatora *regija*, ovdje se definiraju svi ostali dijelovi identifikatora koji se mogu pridružiti određenom tipu *objekta* entiteta. Poslije toga nastavlja se put pretplate. Ta pretplata (odnosno event) služi sustavu misije kao potvrda obavljene misije, ali njome putuje i delegat na metodu koja će transformirati komponente. Također, isti princip puta ima metoda odaberi tip *podobjekta* koja će se pojaviti kasnije. Ostale metode služe kao pomoćne funkcije pri korištenju prije opisanog eventa. Kao što je napomenuto ranije, tipovima *objekata* se pristupa na jednak način (slika 13.).

```
public sealed partial class ObjectTypes
{

    public static Dictionary<InteractableObjectTypeEnum, InteractableObjectTypeAbstract>
        38 references
        InteractableObjectTypes { get; private set; } =
        new Dictionary<InteractableObjectTypeEnum, InteractableObjectTypeAbstract>()
        {
            {InteractableObjectTypeEnum.Reactor, new Reactor()},
            {InteractableObjectTypeEnum.Computer, new Computer()},
            {InteractableObjectTypeEnum.Stasis, new Stasis()},
            {InteractableObjectTypeEnum.Gas, new Gas()},
            {InteractableObjectTypeEnum.Charger, new Charger()},
            {InteractableObjectTypeEnum.Clock, new Clock()}
        };

    2 references
    public static void InitializeObjectTypes()
    {
        MakeNull();
        InteractableObjectTypes = new Dictionary<InteractableObjectTypeEnum,
            InteractableObjectTypeAbstract>()
        {
            {InteractableObjectTypeEnum.Reactor, new Reactor()},
            {InteractableObjectTypeEnum.Computer, new Computer()},
            {InteractableObjectTypeEnum.Stasis, new Stasis()},
            {InteractableObjectTypeEnum.Gas, new Gas()},
            {InteractableObjectTypeEnum.Charger, new Charger()},
            {InteractableObjectTypeEnum.Clock, new Clock()}
        };
    }
```

Slika 13. Definiranje rječnika koji grupira enum tipa *objekta* i definiciju tipa *objekta*

```
95 references
public enum InteractableObjectTypeEnum { Reactor, Computer, Stasis, Gas, Charger, Clock };
public delegate void LogicDelegate(SubObjectTypeEnum subObjectType);
8 references
public abstract class InteractableObjectTypeAbstract
{
    private event LogicDelegate LogicInvoked;
    9 references
    protected abstract void SubscribeConditions(System.Delegate sender);
    7 references
    protected abstract Dictionary<object, object> SetSubComponents();
    7 references
    protected abstract Dictionary<object, object> SetPossibleActions();
    7 references
    protected abstract Dictionary<object, object> SetStats();
    3 references
    public abstract void Subscribe(SubObjectTypeEnum subObjectType,
        ConditionDel conditionDel, StatChangeDel damageDel = null);
    16 references
    public virtual void InvokeStatChange(InteractableObjectStatEnum statType, int amount)
    {
        throw new System.NotImplementedException();
    }
    2 references
    public List<Dictionary<object, object>> GetComponentParts()
    {
        return new List<Dictionary<object, object>>(){SetSubComponents(),
            SetPossibleActions(), SetStats()};
    }
    1 reference
    public virtual bool ChooseSubType(InteractableObject @object)
    {
        return ObjectTypes.SubObjectTypes[@object.SubObjectType].ChooseAction(@object,
            InvokeLogic);
    }
    3 references
    public virtual bool ChooseSubType(IActiveObject @object)
    {
        return ObjectTypes.SubObjectTypes[@object.SubObjectType].ChooseAction(@object);
    }
    5 references
    public void SubscribeLogic(LogicDelegate logicDel)
    {
        LogicInvoked += logicDel;
    }
    2 references
    public void UnSubscribeLogic(LogicDelegate logicDel)
    {
        LogicInvoked -= logicDel;
    }
    4 references
    public void NullifyLogic()
    {
        LogicInvoked = null;
    }
    1 reference
    protected void InvokeLogic(StackData stackData)
    {
        LogicInvoked?.Invoke(stackData.InteractableObject.SubObjectType);
        stackData.InteractableObject.InvokeEventHolder.Pop().Invoke(stackData);
    }
```

Slika 14. Definiranje tvornice apstrakcija za definiranje *objekata* (prvi dio)

Još jedna dodatna kompleksnost, kako se kod nekih ECS sustava želi limitirati globalni pristup, apstraktna klasa tipova *objekata* još se jednom dijeli na dvije apstraktne klase. Glavna razlika između tih interaktivnih entiteta i entiteta koji nisu interaktivni je u tome da je za interaktivne entitete zamišljeno da bar jedan pod tip (roditelj) u njihovoj skupini sadrži ECS komponente (slika 15.).

```
6 references
public class InteractableObjectInterface
{
    2 references
    public abstract class Default : InteractableObjectTypeAbstract
    {

        3 references
        public sealed override bool ChooseSubType(IActiveObject @object)
        {
            return SubObjectTypes[@object.SubObjectType].ChooseAction(@object);
        }
        2 references
        public sealed override void Subscribe(SubObjectTypeEnum subObjectType,
            ConditionDel conditionDel, StatChangeDel statChangeDel = null)
        {
            SubObjectTypes[subObjectType].Subscribe(conditionDel, SubscribeConditions);
        }
    }

    4 references
    public abstract class IChangeStat : InteractableObjectTypeAbstract
    {
        private event StatChangeDel StatChangeEvent;
        16 references
        public sealed override void InvokeStatChange(InteractableObjectStatEnum statType,
            int amount)
        {
            StatChangeEvent.Invoke(new StatChangeEventArgs(statType, amount));
        }
        2 references
        public sealed override void Subscribe(SubObjectTypeEnum subObjectType,
            ConditionDel conditionDel, StatChangeDel statChangeDel = null)
        {
            if(statChangeDel == null)
                SubObjectTypes[subObjectType].Subscribe(conditionDel,
                    SubscribeConditions, statChangeDel);
            else
                StatChangeEvent += SubObjectTypes[subObjectType].Subscribe(conditionDel,
                    SubscribeConditions, statChangeDel);
```

Slika 15. Definiranje tvornice apstrakcija za definiranje *objekata* (drugi dio)

Nakon definiranja apstrakcije treba definirati klasu samog *objektnog* tipa što je prikazano na slici 16. To je jednostavno, pod „subscribe conditions“ definiraju se oni uvjeti koji obavijeste sustav misije o njihovom statusu. Ostale tri metode služe kako bi se definirali kompatibilni identifikatori i komponente koji se kasnije odabiru u editoru.

```
public sealed class Gas : InteractableObjectInterface.IChangeStat
{
    4 references
    protected override void SubscribeConditions(System.Delegate sender)
    {
        Condition.Conditions[ConditionEnum.Maintain].Subscribe(sender as ConditionDel);
        Condition.Conditions[ConditionEnum.RepairGas].Subscribe(sender as ConditionDel);
        Condition.Conditions[ConditionEnum.Vent].Subscribe(sender as ConditionDel);
    }

    2 references
    protected override Dictionary<object, object> SetSubComponents()
    {
        return new Dictionary<object, object>(){
            { System.Enum.GetName(typeof(SubObjectTypeEnum),
            SubObjectTypeEnum.ParentObject),SubObjectTypeEnum.ParentObject },
            { System.Enum.GetName(typeof(SubObjectTypeEnum),
            SubObjectTypeEnum.MiddleButton),SubObjectTypeEnum.MiddleButton }
        };
    }

    2 references
    protected override Dictionary<object, object> SetPossibleActions()
    {
        return new Dictionary<object, object>()
            {
                {  System.Enum.GetName(typeof(ConditionEnum),
                ConditionEnum.RepairGas),ConditionEnum.RepairGas },
                {  System.Enum.GetName(typeof(ConditionEnum),
                ConditionEnum.Maintain),ConditionEnum.Maintain },
                {  System.Enum.GetName(typeof(ConditionEnum),
                ConditionEnum.Vent),ConditionEnum.Vent }
            };
    }

    2 references
    protected override Dictionary<object, object> SetStats()
    {
        return new Dictionary<object, object>() {
            {InteractableObjectStatEnum.Health, new ObjectHealth()},
            {InteractableObjectStatEnum.EnergyConsumption, new ObjectEnergyConsumption()}
        };
    }
```

Slika 16. Definiranje tipa *objekta* gas

5.2.3. Identifikator tipa *podobjekt*

Tipovi *podobjekata* služe kao kraj puta pretplate i kao nastavak puta do *akcije* (slika 17.). Ostalo je jednako kao i u slučaju *regija* ili tipova *objekata*. Ima jedan poseban slučaj, kada se želi koristiti više istih tipova *objekata* u jednoj *regiji*, ali u istom trenutku koristiti modularni način tipa *objekta* (roditelj, tipka), u tu svrhu koriste se lažni roditelj (eng. dummy parent). Tada će roditelj biti tipa entiteta *default* (više o tome kasnije) i interakcija ostalih tipova *podobjekata* i lažnih roditelja će se morati definirati u mehanici.

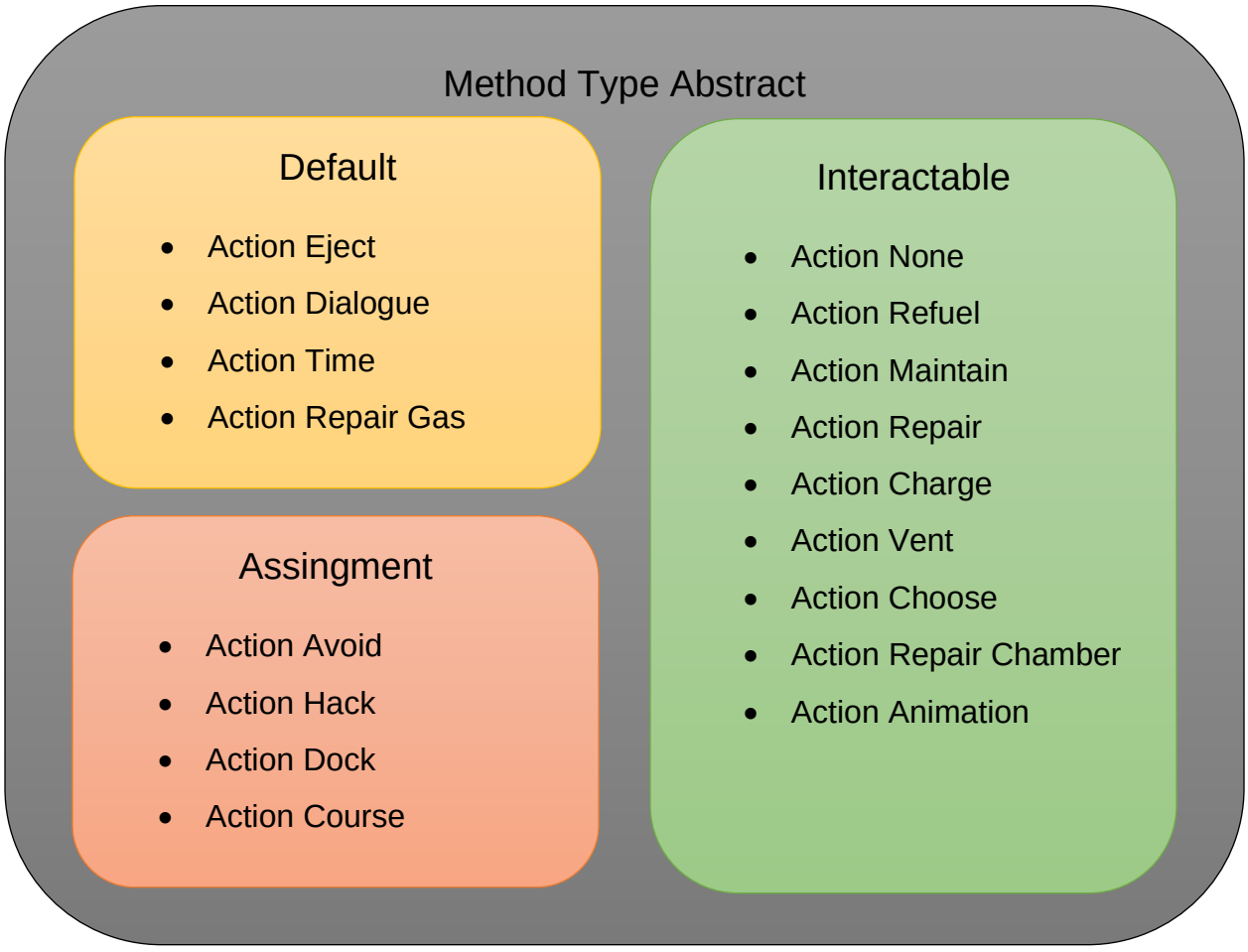
```
public enum SubObjectTypeEnum { ParentObject, DummyParentObject,
    LeftButton, RightButton, MiddleButton, Indicator };
9 references
public abstract class SubObjectTypeAbstract
{
    7 references
    public virtual bool ChooseAction(InteractableObject @object,
        StackDataDel invokeLogic)
    {
        return ((IActed<InteractableObject>)Condition.
            Conditions[@object.ActionType]).Act(@object);
    }

    2 references
    public bool ChooseAction(IActiveObject @object)
    {
        return ((IActed<IActiveObject>)Condition.
            Conditions[@object.ActionType]).Act(@object);
    }
    5 references
    public virtual StatChangeDel Subscribe(ConditionDel conditionDel,
        System.Action<System.Delegate> SubToConditions, StatChangeDel damageDel = null)
    {
        return null;
        //throw new System.Exception();
    }
}
```

Slika 17. Definiranje tvorničke metode za definiranje *podobjekata*

5.2.4. Identifikator tipa *akcija*

Akcije služe kao identifikator i kao globalni pristup sustavima. Naime, svi inputi (klik miša) igrača završavaju u *akcijama*. Što se tiče apstraktne nadklase *akcija,* ona je u usporedbi s tipovima *objekata* relativno jednostavna, ali se kompleksnost stvara pri modeliranju različitih tipova *akcija* (slika 18.). Dodatno, *akcije* imaju tri metode koje se pozivaju sekvencijalno, jedna nakon druge i označuju početak *akcije*, izvršenje *akcije* i kraj *akcije*. Razlog tome je što kod interaktivnih entiteta, kada igrač generira input započne *akcija*, tada se čeka da primjerice čovječuljak izvrši svoju animaciju te kraj animacije obavijesti *akciju* da se izvrši *akcijska* logika. Izvršenje te logike je prvi sloj ECS sustava. Nakon toga pozove se kraj *akcije* koji obavijesti „stream handler" na izvršenje sljedeće *akcije*, a ako ju nema taj dio programa završava s izvršenjem. Time je postignuto da program miruje ako nema inputa.



Slika 18. Sve definirane *akcije* razvrstane po vrsti entiteta

Nadapstraktna klasa *akcija* definira elemente za komunikaciju sa sustavom misija i metodu za pozivanje kontrolera koja uglavnom služi za komuniciranje *akcije* s korisničkim sučeljem pri izvršavanju igre (slika 19.). Ostaje metoda „enact“ (hrv. odigrati) koja je objašnjena prije, ona izvršava logiku prvog sloja ECS sustava.

```
public abstract class MethodTypeAbstract
{
    protected event ConditionDel CondEvent;
    15 references
    protected virtual void Enact(StackData stackData)
    {
        throw new System.NotImplementedException();
    }
    15 references
    public void Subscribe(ConditionDel sender)
    {
        CondEvent += sender;
    }
    2 references
    public virtual void InvokeController(IActiveObject enact)
    {
        throw new System.NotImplementedException();
    }
    1 reference
    public void InvokeCondition(ConditionEnum conditionDel,
        EndPhaseDel endPhase, string dialogue)
    {
        CondEvent.Invoke(new TaskLoadedEventArgs(conditionDel, endPhase, dialogue));
    }
}
```

Slika 19. Definiranje tvornice apstrakcija za definiranje akcija (prvi dio)

Za primjer jedne od *akcijske* apstraktne klase koja nasljeđuje nadapstraktnu klasu promotrit će se apstraktna klasa *interactable*. Kao što je vidljivo na slici 20., uz samo nasljeđivanje nadklase, nasljeđuju se i dodatna sučelja (eng. interface), pa tako *IActed* sučelje implementira metodu „act“ koja se poziva pri pokretanju *akcije*. Ostala sučelja *ISubController* i *ICheckAnswer* služe pri korištenju sustava misija. Također, kao što je prije objašnjeno, dodaje se i metoda „end act“ (hrv. kraj akcije) čijim pozivom završava *akcija*.

```
public abstract class Interactable : MethodTypeAbstract, IActed<InteractableObject>,
    ISubController<ActedInteractableDel>, ICheckAnswer<bool, InteractableObject>
{
    protected event ActedInteractableDel ActedInteractableDel;
    21 references
    public void SubscribeController(ActedInteractableDel sender)
    {
        ActedInteractableDel += sender;
    }
    13 references
    public void UnSubscribeController(ActedInteractableDel sender)
    {
        ActedInteractableDel -= sender;
    }
    2 references
    public sealed override void InvokeController(IActiveObject enact)
    {
        try
        {
            ActedInteractableDel.Invoke(new ActedUponEventArgs((InteractableObject)enact));
        }
        catch (System.NullReferenceException e)
        {
            UnityEngine.Debug.LogError(e);
            UnityEngine.Debug.LogError("Action is probably not subscribed in Tracker, info " +
                "about the object: "+enact.Region +" -> "+ enact.ObjectType +" -> "+
                enact.SubObjectType +" -> "+ enact.ActionType);
        }
    }
    12 references
    public virtual bool Act(InteractableObject _object, StackDataDel action = null)
    {
        _object.InvokeEventHolder.Pop().Invoke(new StackData(_object));
        _object.InvokeEventHolder.Push(Enact);
        return true;
    }
    9 references
    public void EndAct(Stack<StackDataDel> stack)
    {
        stack.Pop().Invoke(new StackData());
    }

    7 references
    public virtual bool Check(InteractableObject _object)
    {
        throw new System.NotImplementedException();
    }
```

Slika 20. Definiranje tvornice apstrakcija za definiranje *akcija* (drugi dio)

Time preostaje definiranje klase same *akcije*. Za primjer uzet će se „repair“ (hrv. popravak) *akcija* koja je prikazana na slici 21. Kao što se može iščitati iz sljedećeg koda, *akcija* „repair“ će pri izvršenju promijeniti komponentu „health“ (hrv. zdravlje) za plus 100 bodova. Tip *objekta* nad kojim će se to izvršiti može se iščitati iz identifikatora entiteta te nakon toga izvršit će se ostale naredbe vezano uz sustave misija. Treba napomenuti da „PlayerStats“ nije ECS komponenta. Uz to, metoda „check“ (hrv. provjeri) provjerava ispunjava li entitet uvjete za otvaranje objektiva sustava misija.

```
public sealed class Repair : ActionInterface.Interactable
{
    2 references
    protected override void Enact(StackData stackData)
    {
        if (stackData.PlayerStats.CanRepair)
        {
            ObjectTypes.InteractableObjectTypes[stackData.InteractableObject.ObjectType].
                InvokeStatChange(InteractableObjectStatEnum.Health, 100);
            stackData.InteractableObject.ConfirmAction(ConditionEnum.Repair, new EndActionData());
            stackData.PlayerStats.CanRepair = false;
        }
        EndAct(stackData.InteractableObject.InvokeEventHolder);
    }

    4 references
    public override bool Check(InteractableObject _object)
    {
        if (_object.Stats[InteractableObjectStatEnum.Health].Points >= 75)
            return true;
        else
            return false;
    }
}
```

Slika 21. Definiranje *akcije* repair (hrv. popravak)

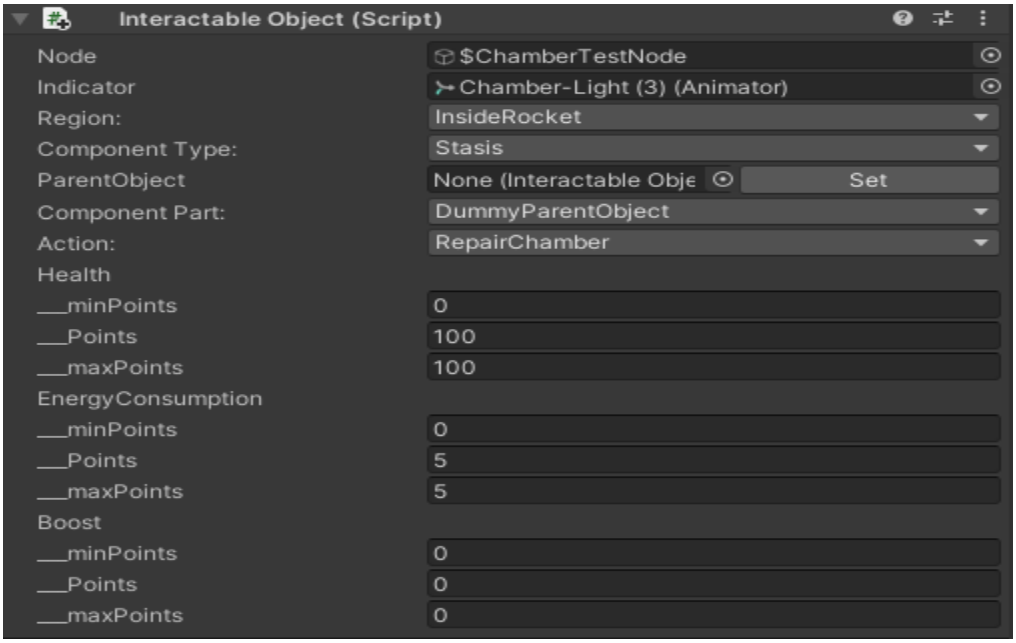
5.2.5. Kreiranje entiteta

Nakon definiranja svih entiteta preostaje njihovo dodavanje u scenu igre. Kako se koristi Unity, njihovo postavljanje se olakšava tako da svi entiteti nasljeđuju od *MonoBehavior*.

Postoje tri različita tipa entiteta:

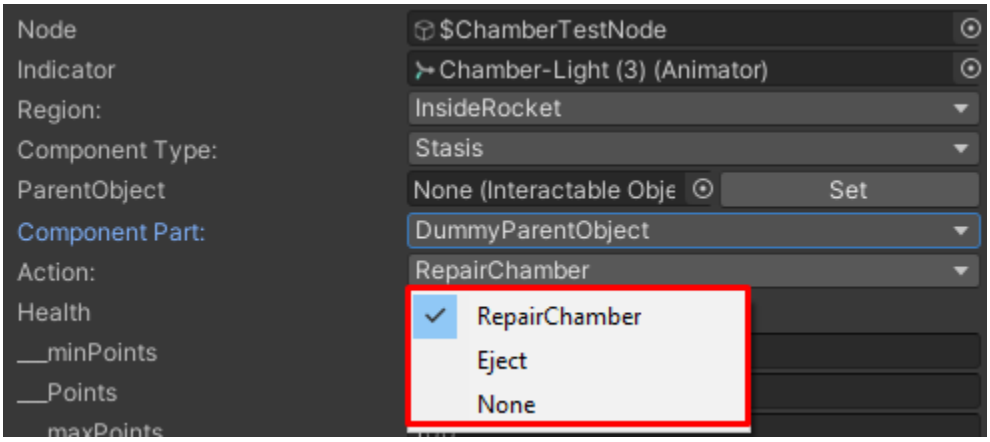
- Interactive (hrv. interaktivan)
    - Entitet nad kojim igrač vrši direktne akcije. Preporučljivo je da se u ovom slučaju koriste *akcije* apstraktne klase *interactable*.
- Default (hrv. standardni)
    - Entitet neovisan o igraču, ali za njegovo korištenje treba dodati mehaniku odnosno lokalni pristup sustavima.
- Dependent (hrv. ovisni)
    - Specijalni tip entiteta koji se ne dodaje u sceni, nego je vezan za određenu *akciju* (početak akcije) i nestaje pri izvršenju te *akcije*.

Njihovo postavljanje je osnovno dodavanje skripte u Unity igraćem objektu (eng. game object) i postavljanja parametara (slika 22.). *Node* i *indicator* služe pri komuniciranju s ostalim ugrađenim sustavima razvojne platforme, a ostalim se koristi ECS.



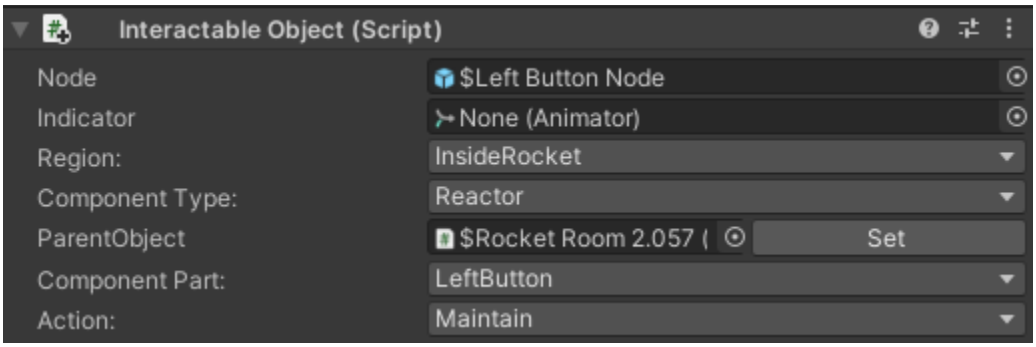
Slika 22. Primjer definiranih identifikatora i komponenata jednog entiteta

Kako su za „stasis“ (hrv. komora za zastoj) definirane samo *akcije* „repair chamber“ (hrv. popravi komoru), „eject“ (hrv. izbaci) i „none“ (hrv. ništa), tako su nam one na raspolaganju (slika 23.). Isto vrijedi i za *podobjekte* (koji su u sučelju nazvani component part).



Slika 23. Prikaz padajućeg imenika koji pokazuje moguće opcije *akcija*

Ako bi se nadalje dodao entitet s *podobjektom* koji nije roditelj ili lažni roditelj, ali njegov roditelj (entitet) bi bio tipa *interactable* tada bi unos parametara izgledao kao na slici 24. Dodatna prednost više identifikatora je u tome što ako se prvo doda entitet roditelj, moguće je klikom na gumb „set“ automatski postaviti referencu na entitet roditelja tog tipa *objekta*.



Slika 24. Prikaz entiteta kod kojeg *podobjekt* nije roditelj

5.3. Opis implementacije komponente ECS-a

Komponente se definira na sličan način kao i identifikatore uz važnu razliku. Kod identifikatora se nisu koristila svojstva (isključujući regije i evente), dok se kod komponenata koriste tri svojstva. Min i max označavaju minimalni i maksimalni mogući iznos bodova (eng. points) dok, bodovi označavaju trenutno stanje bodova komponente. Druga zanimljivost je u tome što u trenutnoj implementaciji treći sloj sustava se definira u komponenti. Isto tako, komponente nisu singleton, već se kreira instanca za svaki tip *objekta,* što se može vidjeti na slici 25. pod metodom „Create Stat“.

```
public enum InteractableObjectStatEnum { Health, EnergyConsumption,
    EnergyGeneration, Fuel, Pressure, EnergyDemand, Boost };
12 references
public enum StatTypeEnum { minPoints, Points, maxPoints };
public delegate InteractableObjectStatAbstract CreateStatDel();
15 references
public abstract class InteractableObjectStatAbstract
{
    /* private int index; ...
    7 references
    public virtual int minPoints { get; set; }
    52 references
    public virtual int Points { get; set; }
    15 references
    public virtual int maxPoints { get; set; }
    8 references
    public abstract void CheckRequirements(InteractableObject @object);
    7 references
    public InteractableObjectStatAbstract CreateStat()
    {
        return Activator.CreateInstance(this.GetType())
            as InteractableObjectStatAbstract;
    }
    4 references
    public void ReverseIfBorderPassed()
    {
        if (this.Points > this.maxPoints)
            this.Points = this.maxPoints;
        else if (this.Points < this.minPoints)
            this.Points = this.minPoints;
    }
}
```

Slika 25. Definiranje tvorničke metode za definiranje *objekta*

Definiranje komponente je vrlo jednostavno. Samo treba definirati u metodu što će se dogoditi nakon promjene broja bodova. Primjerice, prikazano u sljedećem kodu (na slici 26.) na komponenti „tlak objekta“ (eng. object pressure), ako je pritisak prešao maksimalnu dozvoljenu vrijednost, svim tipovima *objekata* koji su definirani u ECS sustavu pod nazivom „električni“ (eng. electrical) oduzet će se sto bodova s komponente „potrošnja energije" (eng. energy consumption). Sljedeće, proizvodnja energije (eng. energy generation) će se postaviti na nulu. Kako je gornja granica sto, ako se oduzme sto, pokrenut će se metoda koja će izjednačiti broj bodova s donjom granicom i na svaki entitet s identifikatorom *objekta* „stasis“ komponenta „pojačanje“ (eng. boost) će se smanjiti za minus dva.

```
sealed class ObjectPressure : InteractableObjectStatAbstract
{

    2 references
    public override void CheckRequirements(InteractableObject @object)
    {
        if (this.Points > this.maxPoints)
        {
            Damage.DamageTypes[DamageTypeEnum.Electrical].ChangeGroupStat(-100);
            InteractableObjectTypes[InteractableObjectTypeEnum.Reactor].
                InvokeStatChange(InteractableObjectStatEnum.EnergyGeneration, -100);
            InteractableObjectTypes[InteractableObjectTypeEnum.Stasis].
                InvokeStatChange(InteractableObjectStatEnum.Boost, -2);
        }
    }
}
```

Slika 26. Definiranje komponente tlak objekta (eng. object pressure)

5.4. Opis implementacije sustava ECS-a

Uz globalni i lokalni pristup sustavima, sustave bi se još moglo podijeliti na tri sloja. Prvi sloj bi bila logika u akciji ili mehanici, drugi sloj bi bila logika kod grupiranja tipova *objekata* i treći sloj bi bila logika kod komponenata.

Detaljnije o pojedinačnoj podijeli sustava:

- Globalni pristup
    - U pitanju su *akcije* koje kako se pokreću iz interakcije igrača s igrom imaju pristup podacima entiteta. Dio je prvog sloja logike.
- Lokalni pristup
    - To su mehanike čiji je smisao da mogu neovisno o interakciji igrača s igrom izvršavati kod. Trenutno, kako Unity razvojna platforma zahtjeva da se sve njene komponente izvršavaju u glavnoj dretvi, potrebno je nakon svakog tika vratiti mehaniku na glavnu dretvu kako bi se mogla izvršiti logika, u suprotnom izvršavala bi se na nekoj sporednoj dretvi (pri spominjanju dretvi, misli se na C# dretve). U slučaju interakcije s njima, jedina informacija entiteta koja do njih dolazi je tip *podobjekta*. Imaju mogućnost definiranja svojstava u koje se može pohraniti neka informacija izvan komponenata. Također su dio prvog sloja logike i služe kao veza s korisničkim sučeljem.
- Drugi sloj logike
    - Detaljnije u kasnijem primjeru, ali važno je napomenuti kako se njihova obrada paralelizira preko Unity jobs biblioteke.
- Treći sloj logike
    - Glavna korist trećeg sloja logike je kako bi se promjena nad jednom komponentom mogla odraziti na promjene ostalih komponenata.

Kroz sljedeći primjer mehanike ukratko će se objasniti od čega se mehanike sastoje. Sa slike 27. se vidi da klasa nije statična niti nema statični pristup. Naime, mehanike se inicijaliziraju u upravitelju igre. Mehanika na slici 27. ima vezu s korisničkim sučeljem kojeg se pokreće sa *ActivateUI* metodom pri pokretanju *akcije* „dijalog“ (eng. dialogue). U ovom slučaju pretplaćivanje na logiku gdje je moguće akcijom utjecati na mehaniku se događa u *ActivateUI* metodi. Kako je već napomenuto, upravitelj igre ima definirane metode za pauzu, nastavak i kraj igre u sklopu scene gdje je definiran. Analogno tome, stop metoda se poziva kada se pozove kraj igre ili prelazak u glavni meni. Uz to, ako se želi implementirati pauza (klikom na meni), kako se mehanike vrte na svojoj dretvi, njihovo izvršavanje treba zaustaviti.

```
public class DialogueGameMechanicController : MechanicInterface.HasUIController<DialogueUI>
{
    1 reference
    public DialogueGameMechanicController(DialogueUI _uIContoller) : base(_uIContoller) { }

    2 references
    protected override void SubscribeConditions()
    {
        (Condition.Conditions[ConditionEnum.Dialogue] as ISubController<ActedDefaultDel>)
            .SubscribeController(ActivateUI);
    }
    6 references
    public override void Stop()
    {
        (Condition.Conditions[ConditionEnum.Dialogue] as ISubController<ActedDefaultDel>)
            .UnSubscribeController(ActivateUI);
        ObjectTypes.InteractableObjectTypes[InteractableObjectTypeEnum.Computer]
            .UnSubscribeLogic(Button);
        base.Stop();
    }
    1 reference
    public void MakeUIStartable()...
    3 references
    protected override void ActivateUI(ComputerConditionSetEventArgs e)
    {
        ObjectTypes.InteractableObjectTypes[InteractableObjectTypeEnum.Computer].SubscribeLogic(Button);
        ConditionSet = e;
        ConditionSet.UIContoller = UIContoller;
        UIScreens = ConditionSet.GetUIScreen();
        UnityEngine.Debug.Log(NotFirstTask);
        UnityEngine.Debug.Log(e.DefaultObject.ObjectType);
        if (NotFirstTask)
            UIContoller.StartUI(UIScreens, new ExtraData(ConditionSet.GetConditionDefinition,
                ConditionSet.CheckAnswers, RefreshChildCount));
    }
```

Slika 27. Primjer postavljanja mehanike za dijalog u igri

Na slici 28. prikazan je dio mehanike koji se izvršava pri interakciji igrača s igrom te je nužan u slučaju korištenja *default* tipova entiteta. Tako je moguće utjecati na odvijanje logike koja se izvršava svaki tik.

```
protected override void Button(SubObjectTypeEnum button_id)
{
    switch (button_id)
    {
        case SubObjectTypeEnum.MiddleButton:
            //temp variables just to be safe
            var tempUI = UIScreens;
            if (string.IsNullOrEmpty(ConditionSet.GetDialogue(ConditionSet.Child_Id + 1)))
            {
                ObjectTypes.InteractableObjectTypes[InteractableObjectTypeEnum.Computer].NullifyLogic();
                UIContoller.StopUI(tempUI, 0);
                ConditionSet.CheckAnswers(true);
            }
            else
                UIContoller.Restart(tempUI, 0, 5);
            break;
        default:
            break;
    }
}
```

Slika 28. Metoda koja se izvršava pri pozivu eventa logike koji je definiran u nadapstraktnoj klasi tipova *objekata*

Time preostaje drugi sloj logike. Sastoji se od enum-a, nadapstraktne klase i dvije metode (slika 29.). Jedna obrađuje komponente samo jednog tipa *objekta* dok druga obrađuje komponente definirane grupe tipova *objekata*. Kako se efekt prve metode može postići preko tipa *objekta*, u sustavima prvog i trećeg sloja koristit će se metoda za obrađivanje grupa tipova *objekata*. U listu koja je samo za čitanje definiraju se identifikatori tipova *objekata* nad kojima će se izvršiti definirana logika odnosno, izvršit će se nad entitetima s tim identifikatorima (njihovim komponentama). Kao što je navedeno, drugi sloj logike koristi paralelizam te je zbog toga potrebno pretvoriti podatke u polje.

```
public sealed class Electrical : DamageTypeAbstract
{
    readonly List<InteractableObjectTypeEnum> temp = new List<InteractableObjectTypeEnum>()
    {
        InteractableObjectTypeEnum.Stasis,
        InteractableObjectTypeEnum.Gas
    };
    4 references
    public override void ChangeGroupStat(int amount)
    {
        Consequence[] consequencesArray = new Consequence[temp.Count];
        for (int i = 0; i < consequencesArray.Length; i++)
        {
            consequencesArray[i].Code = -2;
            consequencesArray[i].CodeEventPhase = -2;
            consequencesArray[i].DamageType = DamageTypeEnum.Electrical;
            consequencesArray[i].ObjectType = temp[i];
            consequencesArray[i].Probability = ProbabilityEnum.Unavoidable;
            consequencesArray[i].Severity = (Tracking.Type.SeverityEnum)amount;
        }
        Decider.ProcessConsequences(ref consequencesArray);
    }

    2 references
    public override void ChangeObjectStat(DamageDelegate invokeStatChange, int amount)
    {
        invokeStatChange.Invoke(InteractableObjectStatEnum.EnergyConsumption, amount);
    }
}
```

Slika 29. Definirana grupa drugog sloja logike

S poljem kojeg se kreiralo potrebno ga je pretvoriti u nativno (eng. native) polje kako bi se kasnije moglo pristupiti podacima obrađenim paralelizmom (slika 30.). Sljedeće, instancira se job (eng. posao) kojeg se zakaže da paralelno obrađuje podatke u poljima deset po deset. Pri završetku, za svaki indeks u polju mijenjaju se vrijednosti komponenata entiteta i poziva se logika trećeg sloja.

```
public static void ProcessConsequences([ReadOnly] ref Consequence[] consequencesArray)
{
    GenerateJob(ref consequencesArray);
}

2 references
private static void GenerateJob([ReadOnly] ref Consequence[] consequencesArray)
{
    var tempNativeArray = new NativeArray<Consequence>(consequencesArray, Allocator.TempJob);
    var damagePercentage = new NativeArray<int>(consequencesArray.Length, Allocator.TempJob);
    var job = new ConsequencesJob()
    {
        Consequences = tempNativeArray,
        RandomNumber = new System.Random().Next(100),
        DamagePercentage = damagePercentage
    };
    var jobHandle = job.Schedule(consequencesArray.Length, 10);
    jobHandle.Complete();
    for (int index = 0; index < job.Consequences.Length; index++)
    {
        Damage.DamageTypes[job.Consequences[index].DamageType].ChangeObjectStat(
            ObjectTypes.RegionTypes[job.Consequences[index].Region].
            Objects[job.Consequences[index].ObjectType].InvokeStatChange, job.DamagePercentage[index]);
    }
    tempNativeArray.Dispose();
    damagePercentage.Dispose();
}
```

Slika 30. Prikaz generiranja posla i alociranja memorije za nativno polje

6. Implementacija ECS arhitekture za frontend razvoj web aplikacija

Implementacija je pisana u TypeScript-u i korišten je Vue 3.0 za lakšu komunikaciju ECS-a i html elemenata. Ideja iza korištenja ECS-a u frontend okruženju je kako bi se izdvojio logički dio aplikacije te smanjio i pojednostavio broj API poziva na neki backend server. Dodatno, developer ima slobodu za prepravljanjem Vue komponenata (preslaganjem html elemenata i mijenjanjem dizajna) bez potrebe za prepravljanjem logike aplikacije. Uz to, kako se koriste entiteti jedan tip entiteta je dovoljan za opisati sve Vue komponente. Za backend je korišten Laravel i MySQL baza podataka. Glavni razlog iza ove implementacije je kao dokaz modularnosti baznog dijela ECS implementacije.

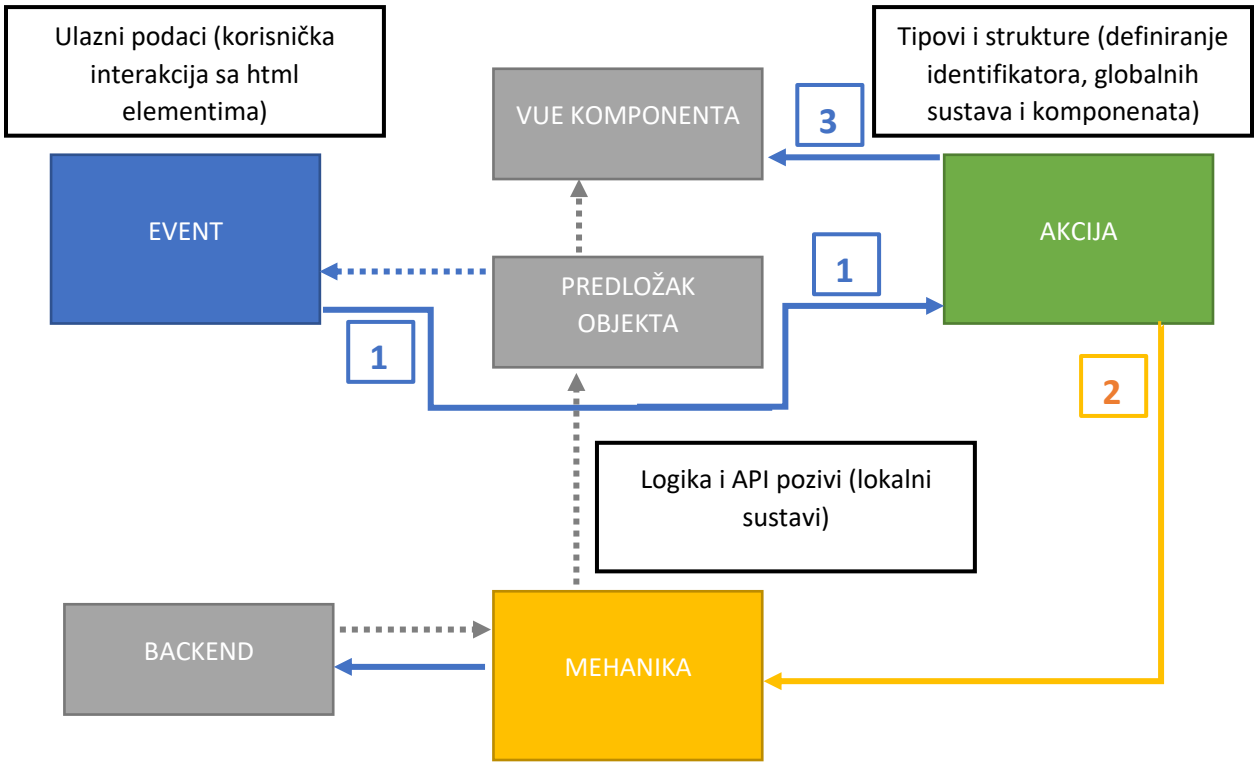
Teoretska prednost je u tome kako je jedino potrebno pamtiti entitete (njihove identifikatore i komponente), čime se otvara mogućnost da je za cijeli backend dovoljno koristiti samo Redis bazu podataka. Uz to, jedna od nuspojava specifično ove implementacije je modularno verzioniranje.

Kao i u prijašnjoj implementaciji, definirano korištenje ECS-a je čisto eksperimentalno te je moguće da se određene značajke ne koriste kako je to zamišljeno u standardnom opisu ECS-a. Elementi aplikacije su pretvoreni u entitete približno kao što je to prikazano na slici 31.

Slika 31. Isječak web aplikacije na kojem se vide podijeljeni entiteti po *regijama* (po bojama iz slike 6)

6.1. Sveobuhvatno objašnjenje izvođenja implementacije

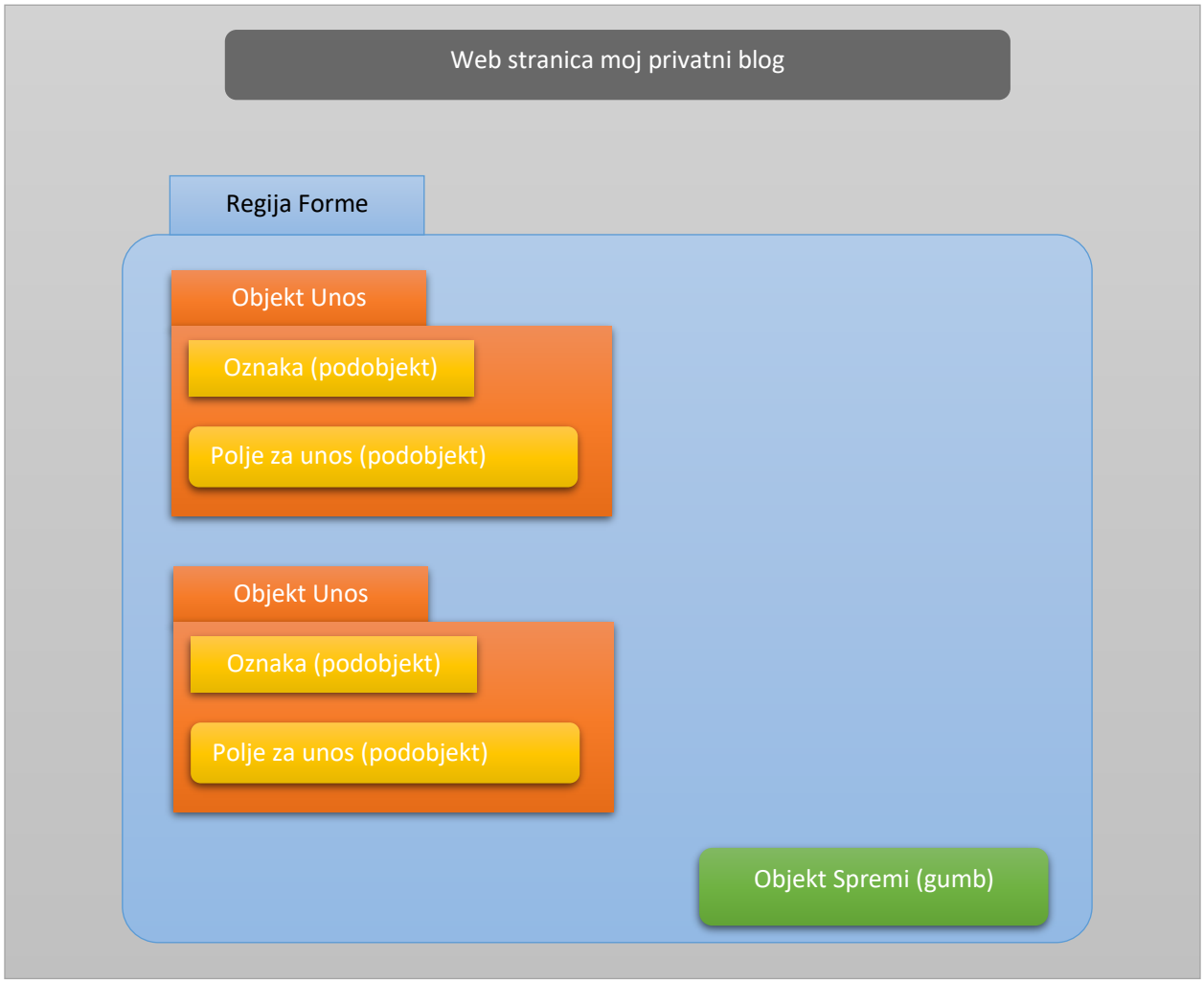
Na slici 32. prikazano je apstraktno izvođenje web aplikacije, odnosno ECS-a i njemu ovisnih dijelova programa. Kako postoje html eventi, nije potrebno implementirati Event modul te on na slici 32 predstavlja početak korisničke interakcije s aplikacijom. Inicijalizacija Vue komponente se vrši preko mehanike tako da se generira entitet za svaku Vue komponentu. Tada se interakcijom nad html elementom, ovisno o identifikatorima entiteta, izvršava određena *akcija*. Pri izvršenju *akcije*, moguće je pozvati lokalni sustav kako bi se izvršila neka radnja preko API poziva. U trećem koraku, izvršene promjene se ispišu korisniku preko Vue komponente.



Slika 32. Apstraktni prikaz izvođenja web aplikacije pri interakciji korisnika

6.2. Opis implementacije entiteta ECS-a

Struktura entiteta ostaje ista kako je objašnjena u baznoj implementaciji. Za razliku od implementacije za razvoj igara, *regije* su utilizirane i pridružuju se entitetima kao identifikator čime zauzimaju virtualni prostor na ekranu (slika 31). Tip *objekta* i dalje reprezentira skup *podobjekata* te je smanjena korist *akcija* kako su one ugrađene u html elemente. Kao što je prikazano na slici 33., input polja i oznake (eng. label) nemaju *akcije*, već se koristi Vue v-model direktiva kojom se postiže dvosmjerno povezivanje podataka (isto se može postići preko sustava). Entitet s identifikatorom *objekta* gumb je u tome različit kako je često njegova uloga izvršavanje nekog API poziva koji je definiran u sustavima.



Slika 33. Skica sučelja forme kroz koji je prikazana uloga identifikatora (po bojama iz slike 6)

6.2.1. Identifikator tipa *regija*

Regije i ostali identifikatori se definiraju na vrlo sličan način prijašnjoj implementaciji. Prvo se definira enum te se onda otvara apstraktna klasa (slika 34.). Kako ECS5 standard ne podržava *namespace* strukture, koriste se moduli kako bi se moglo kompresirati datoteke u jednu JavaScript datoteku (čime se znatno poboljšavaju performanse web aplikacije). Pristup tipovima *regija* je također par ključa i vrijednosti jedino ovog puta koriste se JavaScript objekti umjesto generičnih asocijativnih polja.

```
import { Manager } from './regionTypes/types'

export enum RegionEnum {Form, Table, TableColumn, Show}

export class RegionType {
    public static RegionTypes: { [index: number]: Manager.Events.Type.RegionAbstract } =
    {
      [RegionEnum.Form]: new Manager.Events.Type.Form(),
      [RegionEnum.Table]: new Manager.Events.Type.Table(),
      [RegionEnum.TableColumn]: new Manager.Events.Type.TableColumn(),
      [RegionEnum.Show]: new Manager.Events.Type.Show()
    }
}
```

Slika 34. Definiranje JavaScript objekta koji grupira enum *regije* i definiciju *regije*

Definira se *regija* „form“ (hrv. obrazac) koja sadrži metodu pretplati (slika 35.). Time će se označiti skupina entiteta koja čini jedan obrazac kao što je vidljivo na slici 33.

```
import { StatChangeDel } from '@/interface/manager/containerClasses/statChangeEventArgs'
import { ObjectTypeEnum, ObjectType } from '../objectType'
import { SubObjectTypeEnum } from '../subObjectType'

export namespace Manager.Events.Type{

   export abstract class RegionAbstract {
      public abstract Subscribe(_objectType: ObjectTypeEnum,
        _subObjectType: SubObjectTypeEnum, _statChangeDel:StatChangeDel) : void;
   }

  export class Form extends RegionAbstract {
    public Subscribe (_objectType: ObjectTypeEnum,
      _subObjectType: SubObjectTypeEnum, _statChangeDel:StatChangeDel): void {
      ObjectType.ObjectTypes[_objectType].Subscribe(_subObjectType, _statChangeDel)
    }
  }
}
```

Slika 35. Definiranje regije form (hrv. obrazac)

6.2.2. Identifikator tipa objekt

Kako je definirana *regija* „form“, sljedeće treba definirati elemente obrasca. Osnovni elementi bi bili polje za dodavanje vrijednosti (eng. input field) i gumb za poziv ECS sustava te ih se pridruži njihovim enum vrijednostima (slika 36.).

```
import { Manager } from './objectTypes/types'

export enum ObjectTypeEnum {
  Row, Field, Button, Text, ShowResolve
}

export class ObjectType {
 public static ObjectTypes: { [index: number]: Manager.Events.Type.ObjectTypeAbstract } =
 {
   [ObjectTypeEnum.Field]: new Manager.Events.Type.Field(),
   [ObjectTypeEnum.Button]: new Manager.Events.Type.Button(),
   [ObjectTypeEnum.Row]: new Manager.Events.Type.Row(),
   [ObjectTypeEnum.Text]: new Manager.Events.Type.Text(),
   [ObjectTypeEnum.ShowResolve]: new Manager.Events.Type.ShowResolve()
 }
}
```

Slika 36. Definiranje JavaScript objekta koja grupira enum *objekta* i definiciju *objekta*

Definiranje samih tipova *objekata* se sastoji od metode koja vraća Vue komponentu vezanu za taj tip *objekta*. Koristi se apstraktna tvornica za generiranje tipova *objekata* (slika 37.), a pretplate na event koji mijenja neku od komponenata istih tipova *objekata* se definiraju u apstraktnoj klasi *IChangeStat*.

```
export abstract class IChangeStat extends ObjectTypeAbstract {
 private StatChangeEvent:SimpleEventDispatcher<StatChangeEventArgs> =
 new SimpleEventDispatcher<StatChangeEventArgs>();
 public InvokeStatChange (_statType: StatTypeEnum, _amount: any) : void{
   this.StatChangeEvent.dispatch(new StatChangeEventArgs(_statType, _amount))
 }
 public Subscribe (_subObjectType:SubObjectTypeEnum, _statChangeDel:StatChangeDel) :void{
   /* if(statChangeDel == null)▬      else */
   this.StatChangeEvent.subscribe(SubObjectType.SubObjectTypes[_subObjectType].
     Subscribe(_statChangeDel))
 }
}
export class Field extends IChangeStat {
  public GetVueComponent () {
    return InputComponent
  }
}
```

Slika 37. Definiranje tvornice apstrakcija za definiranje *objekata* (prvi dio)

Definiranje tipova *objekata* je pojednostavljeno kako postoji samo jedan tip entiteta. Novitet je korištenje Vue komponenata čime tip *objekta* doslovno reprezentira Vue komponentu u ECS-u. Također, definirane su pomoćne metode za upravljanjem pretplata nad logikom koja se definira u mehanikama te put do *akcija* za mijenjanje vrijednosti podataka komponenata (slika 38).

```
export type LogicDelegate = (subObjectType: SubObjectTypeEnum) => void;

export namespace Manager.Events.Type{

    export abstract class ObjectTypeAbstract {
       private LogicInvoked: SimpleEventDispatcher<SubObjectTypeEnum> =
       new SimpleEventDispatcher<SubObjectTypeEnum>();
      // protected abstract SubscribeCondition(sender: () => void) : void;
      public abstract Subscribe(subObjectType:SubObjectTypeEnum,
        statChangeDel:StatChangeDel) : void;

      public abstract GetVueComponent(): any;
      public InvokeStatChange (_statType: StatTypeEnum, _amount: any): void {
        throw new Error('Method not implemented.')
      }
      public ChooseSubType (_object : ObjectTemplate) : boolean {
        return SubObjectType.SubObjectTypes[_object.SubObjectEnum].
        ChooseAction(_object, this.InvokeLogic.bind(this))
      }
      protected InvokeLogic (_subObjectType: SubObjectTypeEnum) : void {
        this.LogicInvoked.dispatch(_subObjectType)
        console.log(this.LogicInvoked)
      }
      public SubscribeLogic (logicDel: LogicDelegate) : void {
        this.LogicInvoked.subscribe(logicDel)
      }
      public UnSubscribeLogic (logicDel : LogicDelegate) : void {
        this.LogicInvoked.unsubscribe(logicDel)
      }
      public NullifyLogic () : void {
        this.LogicInvoked.clear()
      }
    }
```

Slika 38. Definiranje tvornice apstrakcija za definiranje *objekata* (drugi dio)

6.2.3. Identifikator tipa *podobjekt*

U ovoj implementaciji svi tipovi *podobjekata* mogu sadržavati komponente. Razlika između roditelja i djece je u tome da komponente pridružene djeci su namijenjene samo za čitanje nakon inicijalnog upisa te se takve tipove *podobjekata* ne pretplaćuje na event koji mijenja vrijednosti komponenata. Kao što je prikazano na slici 39., tipovi *podobjekata* su jednaki prošloj implementaciji, osim što nema lažnih roditelja kako svi tipovi *podobjekata* mogu imati komponente.

```
import { Manager } from './subObjectTypes/types'

export enum SubObjectTypeEnum {ParentObject, Middle, Left, Right}

export class SubObjectType {
 public static SubObjectTypes: { [index: number]: Manager.Events.Type.SubObjectTypeAbstract } = {
   [SubObjectTypeEnum.ParentObject]: new Manager.Events.Type.ParentObject(),
   [SubObjectTypeEnum.Middle]: new Manager.Events.Type.Middle(),
   [SubObjectTypeEnum.Left]: new Manager.Events.Type.Left(),
   [SubObjectTypeEnum.Right]: new Manager.Events.Type.Right()
 };
}
```

Slika 39. Definiranje JavaScript objekta koji grupira enum *podobjekta* i definiciju *podobjekta*

Nadapstraktna klasa za definiranje tipova *podobjekata* sadrži metodu odaberi *akciju* i metodu kao kraj puta pretplate. Definiraju se po tipovima roditelj, lijevo, desno i sredina (slika 40.).

```
export namespace Manager.Events.Type{

    export abstract class SubObjectTypeAbstract {
      public ChooseAction (_object: ObjectTemplate, _data : any, _invokeLogic: LogicDelegate) : boolean {
        return ActionType.ActionTypes[_object.ActionEnum].Act(_object, _data, _invokeLogic)
      }

      public Subscribe (_statChangeDel: StatChangeDel): StatChangeDel {
        return _statChangeDel
      }
    }
    export class ParentObject extends SubObjectTypeAbstract {}
    export class Middle extends SubObjectTypeAbstract {}
    export class Left extends SubObjectTypeAbstract {}
    export class Right extends SubObjectTypeAbstract {}
}
```

Slika 40. Definiranje *podobjekata* i njihove nadapstraktne klase

6.2.4. Identifikator tipa *akcija*

Kako Vue pruža v-on direktivu koja je pretplaćena na DOM evente, globalni sustavi prvog sloja služe za transformaciju unesenih podataka ili za utjecanje na lokalne sustave (mehaniku). Definiraju se *akcije* „none“, „click“ i „insert“ (hrv. ništa, klik i unos) (slika 41.).

```
import { Manager } from './actionTypes/types'

export enum ActionTypeEnum {None, Click, Insert}

export class ActionType {
 public static ActionTypes: { [index: number]: Manager.Events.Type.MethodTypeAbstract } = {
   [ActionTypeEnum.None]: new Manager.Events.Type.None(),
   [ActionTypeEnum.Click]: new Manager.Events.Type.Click(),
   [ActionTypeEnum.Insert]: new Manager.Events.Type.Insert()
 }
}
```

Slika 41. Definiranje JavaScript objekta koji grupira enum *akcije* i definiciju *akcije*

Na slici 42. pod *akcijom* „insert“ koji bi se koristio kao identifikator kod polja za dodavanje vrijednosti (input field), može se postaviti da se ta „insert“ *akcija* okine pri nekom Vue eventu te transformira unesenu vrijednost (primjerice, tako da kapitalizira prvu riječ u tekstu). Ponovno postoji red izvođenja *akcije* koji se u ovom slučaju dijeli na početak akcije i izvršenje akcije. Prednost u ovom okruženju je u tome što je puno jednostavnije paralelizirati izvršenje *akcije*.

```
export class Insert extends MethodTypeAbstract {
  public Act (_object: ObjectTemplate, _data : any, _invokeLogic: LogicDelegate): boolean {
    this.Enact(_data).then(response => (_object.Stats[StatTypeEnum.Value].Data = response))
    return true
  }

  public async Enact (_data : any): Promise<any> {
    return await _data.charAt(0).toUpperCase() + _data.slice(1)
  }
}
```

Slika 42. Definiranje *akcije* insert (hrv. unos)

6.2.5. Kreiranje entiteta

Za generiranje entiteta treba definirati „object template“ za svaki pojedini entitet, njihove identifikatore i komponente. Njihovu definiciju se može definirati odmah u Vue komponenti, ili ju se može dohvatiti s backend servera. Te definicije se pohranjuju kao polje u mehaniku (slika 43.) koja se inicijalizira s Vue komponentom.

```
objectTemplates = this.mechanic.InitSet(
  [
    new ObjectTemplate(RegionEnum.TableColumn, ObjectTypeEnum.Button, SubObjectTypeEnum.Left, ActionTypeEnum.Click, {
      [StatTypeEnum.Title]: StatType.StatTypes[StatTypeEnum.Title]().CreateStat().InitData('Izbriši'),
      [StatTypeEnum.Design]: StatType.StatTypes[StatTypeEnum.Design]().CreateStat().InitData('btn btn-outline-danger'),
      [StatTypeEnum.Id]: StatType.StatTypes[StatTypeEnum.Id]().CreateStat().InitData(this.object.Stats[StatTypeEnum.Id].Data)
    }),
    new ObjectTemplate(RegionEnum.TableColumn, ObjectTypeEnum.Button, SubObjectTypeEnum.Middle, ActionTypeEnum.Click, {
      [StatTypeEnum.Title]: StatType.StatTypes[StatTypeEnum.Title]().CreateStat().InitData('Uredi'),
      [StatTypeEnum.Design]: StatType.StatTypes[StatTypeEnum.Design]().CreateStat().InitData('btn btn-outline-warning'),
      [StatTypeEnum.Id]: StatType.StatTypes[StatTypeEnum.Id]().CreateStat().InitData(this.object.Stats[StatTypeEnum.Id].Data)
    }),
    new ObjectTemplate(RegionEnum.TableColumn, ObjectTypeEnum.Button, SubObjectTypeEnum.Right, ActionTypeEnum.Click, {
      [StatTypeEnum.Title]: StatType.StatTypes[StatTypeEnum.Title]().CreateStat().InitData('Pregledaj'),
      [StatTypeEnum.Design]: StatType.StatTypes[StatTypeEnum.Design]().CreateStat().InitData('btn btn-outline-success'),
      [StatTypeEnum.Id]: StatType.StatTypes[StatTypeEnum.Id]().CreateStat().InitData(this.object.Stats[StatTypeEnum.Id].Data)
    })
  ]
)
```

Slika 43. Generiranje entiteta u Vue komponenti

Korištenjem entiteta otvaraju se nove mogućnosti, poput modularnog verzioniranja, a kako se komponente mogu seliti s entiteta na entitet, za promjenu izgleda i mogućnosti pojedinog entiteta dovoljno je samo zamijeniti identifikatore kao što je prikazano na slici 44. Modularno verzioniranje se postiže tako da se sprema cijeli obrazac kao Json u tablicu baze podataka (slika 45.) te pri promjeni obrasca koji je definiran na backend-u (slika 46.) pri uređivanju spremljenih podataka, spremljeni podaci će se prikazati na starom obrascu.

Modularno verzioniranje može poslužiti pri agilnom razvoju aplikacije, u slučaju gdje se mijenjaju dijelovi obrasca te ispunjavanje novog obrasca zahtjeva ispunjavanje dodatnih polja, ili neki od prijašnjih polja su izuzeti iz obrasca. Korisnik koji je ispunjavao stari obrazac koji kao u primjeru ove aplikacije generira neki blog može i dalje koristiti stari obrazac za uređivanje svog bloga, a u isto vrijeme uživati u ostalim nadograđenim modulima koji ne zahtijevaju novododane entitete. Naravno, kako se koriste entiteti, kada je nadogradnja spremna, automatski ili manualno moguće je nadograditi blog na novu verziju.

```
export class TableMechanic extends MechanicAbstract {
  public async InitGet (_id = -1): Promise<ObjectTemplate[]> {
    this.ObjectTemplates = []
    const response = await http.get('http://blog.test/api/entity')
    return (this.ObjectTemplates = this.forEachElement(response.data))
  }

  private forEachElement (data: any) : ObjectTemplate[] {
    let _temp: ObjectTemplate[] = []
    data.forEach((_list: any) => {
      _temp = _temp.concat(_list.filter((_object : ObjectTemplate) =>
      { return _object.Stats[StatTypeEnum.Label].Data === 'Title' }).map((_object: any) => {
        return new ObjectTemplate(RegionEnum.Table, ObjectTypeEnum.Row,
          SubObjectTypeEnum.ParentObject, ActionTypeEnum.Click, this.reStructure(_object.Stats))
      }))
    })
    return _temp
  }
```

Slika 44. Primjer kako se pri dohvaćanju definicije entiteta oni transformiraju u nove Vue komponente (po bojama iz slike 6.)

entities | Enter a SQL expression to filter results (use Ctrl+Space)

Grid | Text

	id				json
1	3	21-07-	-07-25 0	-07-25	[{"Stats": [{"Data": "Naslov"}, {"Data": "dfg"}, {"Data": null}, {"Data": "Title"}, {"Data": "3"}], "Region": 0,
2	2	21-07-	-07-25 0	-07-25	[{"Stats": [{"Data": "Naslov"}, {"Data": "yxc"}, {"Data": null}, {"Data": "Title"}, {"Data": "2"}], "Region": 0,
3	1	21-07-	-07-25 0	-07-25	[{"Stats": [{"Data": "Naslov"}, {"Data": "asd"}, {"Data": null}, {"Data": "Title"}, {"Data": "1"}], "Region": 0,
4	4	21-07-	-07-25 0	-07-25	[{"Stats": [{"Data": "Naslov"}, {"Data": "asd"}, {"Data": null}, {"Data": "Title"}, {"Data": "4"}], "Region": 0,
5	5	21-07-	-07-25 0	-07-25	[{"Stats": [{"Data": "Naslov"}, {"Data": "Sv. Agata"}, {"Data": null}, {"Data": "Title"}, {"Data": "5"}], "Regio
6	7	21-07-	-07-25 0	[NULL]	[{"Stats": [{"Data": "Naslov"}, {"Data": "Crkva Sv. Valentina"}, {"Data": "form-control"}, {"Data": "Title"},
7	6	21-07-	-07-28 1	[NULL]	[{"Stats": [{"Data": "Naslov"}, {"Data": "Crkva Sv. Agata"}, {"Data": null}, {"Data": "Title"}, {"Data": "6"}],

Slika 45. Isječak iz MySQL baze podataka u kojoj se spremaju entiteti po *regijama*

```php
public function resolveRegion(){
    return response(
        [
            [
                "Stats" =>
                [
                    self::Title =>["Data" => "Naslov"],
                    self::Value => ["Data" => null],
                    self::Design =>["Data" => "form-control"],
                    self::Label =>["Data" => "Title"]
                ],
                "Region" => 0,
                "ObjectEnum" => self::Field,
                "SubObjectEnum" => 0,
                "ActionEnum" => 2
            ],
            [...],
            [...],
            [...]
        ]
    );
}
```

Slika 46. Priprema obrasca na backend-u

6.2.6. Opis implementacije komponente ECS-a

Komponente reprezentiraju neku vrijednost od dijelova html elemenata i mogu sadržavati neki od identifikatora vanjskih sustava poput baze podataka.

Kako se pojavljuju na slici 47., po redu objašnjenje pojedine komponente:

- Title (hrv. naslov)
  - Reprezentira ime oznake polja za unos kako će se on ispisati na ekranu.
- Value (hrv. vrijednost)
  - Vrijednost koju korisnik unosi u primjerice polje za unos.
- Design (hrv. dizajn)
  - Koristi se za definiranje CSS klasa koje će entitet koristiti za prikaz.
- Label (hrv. oznaka)
  - Unutarnja oznaka komponente koji može poslužiti pri filtriranju entiteta.
- Id
  - Sadrži identifikator koji se koristi pri radu nad bazom podataka.

```
import { Manager } from './statTypes/types'

export enum StatTypeEnum {Title, Value, Design, Label, Id}

export class StatType {
 public static StatTypes: { [index: number]: Manager.Events.Type.CreateStatDel } =
 {
   [StatTypeEnum.Title]: new Manager.Events.Type.Title().CreateStat,
   [StatTypeEnum.Value]: new Manager.Events.Type.Value().CreateStat,
   [StatTypeEnum.Design]: new Manager.Events.Type.Design().CreateStat,
   [StatTypeEnum.Label]: new Manager.Events.Type.Label().CreateStat,
   [StatTypeEnum.Id]: new Manager.Events.Type.Id().CreateStat
 }
}
```

Slika 47. Definiranje JavaScript objekta koji grupira enum komponente i definiciju komponenata

Komponente sadrže podatak kao svojstvo, moguće je inicijalizirati komponentu s nekim podatkom i naravno podržava treći sloj sustava (slika 48.).

```
export namespace Manager.Events.Type{

    export abstract class StatAbstract {
      public Data = '';
      public abstract CheckRequirements(_object:any):void
      public abstract CreateStat ():StatAbstract;
      public InitData (_data:string):StatAbstract {
        this.Data = _data
        return this
      }
    }
    export type CreateStatDel = () => StatAbstract;

    export class Title extends StatAbstract {
      public CreateStat (): StatAbstract {
        return new Title()
      }

      public CheckRequirements (_object: any): void {
        alert(this.Data)
      }
    }

    export class Value extends StatAbstract {
      public CreateStat (): StatAbstract {
        return new Value()
      }

      public CheckRequirements (_object: any): void {
        alert(this.Data)
      }
    }
}
```

Slika 48. Definiranje komponenata i njihove nadapstraktne klase

6.3. Opis implementacije sustava ECS-a

Apstraktna nadklasa lokalnih sustava (mehanika) se sastoji od metoda za pretplatu na logiku tipa *objekta* i definiranje logike koja će se izvršiti pri njegovom okidanju (slika 49.). Inicijalizacijskih metoda od kojih jedna dohvaća entiteta s servera, a druga se koristi pri postavljanju entiteta u Vue komponenti kao na slici 43. Mehanika nam još i služi i kao kolekcija entiteta određene *regije*. Naime, poželjno je da se za svaku *regiju* definira njezina mehanika.

```
export abstract class MechanicAbstract {
  protected abstract SubscribeConditions() : void;
  public abstract UnsubscribeConditions() : any;
  protected abstract Button(_subObjectType: SubObjectTypeEnum) : void;
  constructor () {
    this.SubscribeConditions()
  }

  public abstract InitGet(_id : number): Promise<ObjectTemplate[]>;

  public abstract InitSet(_objectTemplates: ObjectTemplate[]) : ObjectTemplate[];

  protected ObjectTemplates!: ObjectTemplate[];
}
```

Slika 49. Prikaz apstraktne nadklase lokalnih sustava

Na primjeru sa slike 50. gdje je prikazana mehanika za *regiju* „form“ (obrazac), ako se primjerice dodaje novi blog, prvo će se upitati bazu za novi *Id* koji će se koristiti pri spremanju obrasca u bazu podataka kako bi se ga moglo upisati u komponentu *Id*. Tada se dohvaća sam obrazac, odnosno definirani entiteti koji se u nastavku generiraju.

```
export namespace Manager.Mechanic{

  export class FormMechanic extends MechanicAbstract {
    private id = -1;
    private inEdit = false;

    public async InitGet (_id = -1): Promise<ObjectTemplate[]> {
      this.id = _id
      if (this.id === -1) {
        this.id = (await http.get('http://blog.test/api/entity/' + this.id)).data
        console.log(this.id)
        const response = await http.get('http://blog.test/api/form')
        return (this.ObjectTemplates = response.data.map((_object: any) => {
          return new ObjectTemplate(_object.Region, _object.ObjectEnum,
            _object.SubObjectEnum, _object.ActionEnum, this.reStructure(_object.Stats,
              { [StatTypeEnum.Id]: StatType.StatTypes[StatTypeEnum.Id]().CreateStat()
                .InitData(String(this.id)) }))
        }))
      }
      const response = await http.get('http://blog.test/api/entity/' + this.id)
      this.inEdit = true
      return (this.ObjectTemplates = response.data.map((_object: any) => {
        return new ObjectTemplate(_object.Region, _object.ObjectEnum,
          _object.SubObjectEnum, _object.ActionEnum, this.reStructure(_object.Stats))
      }))
    }
```

Slika 50. Isječak form mehanike (lokalni sustav) na kojem je prikazano dohvaćanje definicije entiteta s backend-a

Implementacija lokalnog sustava te način na koji se vrše „post“ i „patch“ zahtjevi (eng. request) za mehaniku „form“ je prikazan na slici 51. Treba pripaziti na korištenje lokalnih sustava pri upotrebi više *regija* tako da ih se filtrira u globalnim sustavima i uvijek očiste pretplate na eventu logike.

```
protected SubscribeConditions (): void {
  ObjectType.ObjectTypes[ObjectTypeEnum.Button].SubscribeLogic(this.Button.bind(this))
}

public UnsubscribeConditions () {
  ObjectType.ObjectTypes[ObjectTypeEnum.Button].NullifyLogic()
}

protected async Button (_subObjectType: SubObjectTypeEnum): Promise<void> {
  switch (_subObjectType) {
    case SubObjectTypeEnum.Middle:
      if (this.inEdit) {
        await http.patch('http://blog.test/api/entity/' + this.id, this.ObjectTemplates)
          .then(response => (router.push({ name: 'Show', params: { id: response.data.id } })))
      } else {
        await http.post('http://blog.test/api/entity', this.ObjectTemplates)
          .then(response => (router.push({ name: 'Show', params: { id: response.data.id } })))
      }
      break
    default:
      break
  }
}
```

Slika 51. Isječak form mehanike na kojoj se vidi definirana logika lokalnog sustava

7. Usporedba implementacije sa Entitas implementacijom

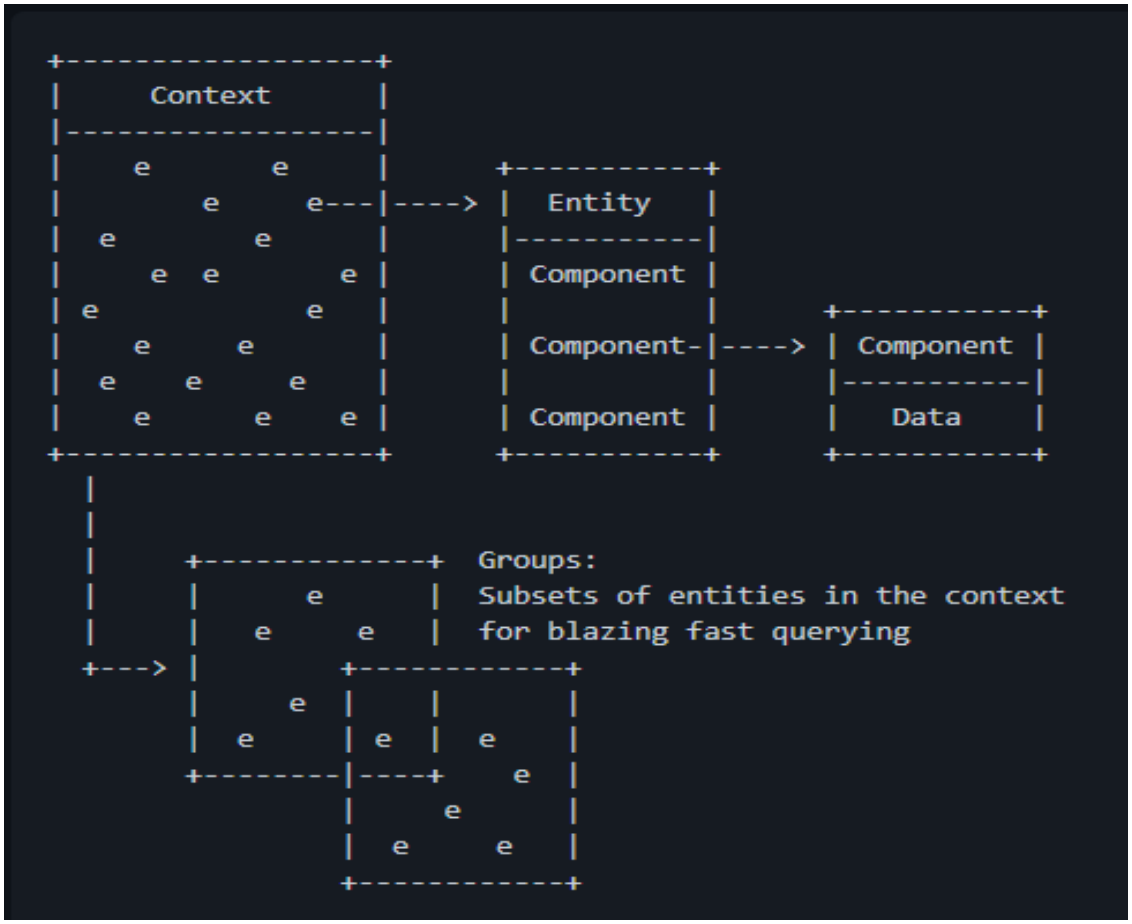
Entitas se reklamira kao brza, lagana ECS implementacija niske kompleksnosti. Razlog usporedbe s Entitas-om je kako je implementacija pisana za C# i TypeScirpt. U oba slučaja dan je primjer implementacije za razvoj igre, pri čemu je C# implementacija usko vezana s Unity razvojnom platformom. Trenutno, implementacija se doima zapuštenom kako je zadnje ažuriranje bilo prije tri godine [9].

Implementacija se sastoji od nekoliko zanimljivih značajki kao što je prikazano na slici 52. [9]:

- Entitet
  - Spremnik koji sadrži komponente kojima reprezentira određeni objekt u aplikaciji. Moguće je dodavati, mijenjati i brisati komponente. Dodatno imaju evente kojima naglašuju kada se dogodila neka od tih promjena.
- Kontekst
  - Tvornička metoda u kojoj se entiteti generiraju i brišu. Koristi se za filtriranje većih grupa entiteta.
- Grupa
  - Omogućuju brzo filtriranje entiteta u kontekstu. Kontinuirano se ažuriraju pri promjeni entiteta i u trenutku vraćaju grupe entiteta. Kako su grupe predmemorirane, preporučuje se njihovo korištenje gdje god moguće. Također, imaju evente kojima naglašavaju kada se desila neka promjena.
- Kolektor
  - Pruža jednostavan način za reagiranje nad promjenama u grupama u nekom vremenskom tijeku. Nadgledava promjene nad entitetima kod kojih se desila promjena nad nekom komponentom, tada prikuplja te entitete i obrađuje ih.

Glavni nedostatak ove implementacije i ostalih ECS implementacija koje se baziraju na konceptu relacijskih baza je u tome što povećavanjem broja entiteta i komponenata sama konceptualna tablica postaje ogromna. Kako se kod Entitas implementacije koriste polja pokazivača te svaki pokazivač zauzima 8 bajtova na 64bit arhitekturi, pri korištenju 50 komponenata svaki entitet će imati oko 400 bajtova. Tih 400 bajtova se transponira na 40KB pri korištenju 100 entiteta. Kao lijek na ovaj problem predloženo je da se entitete dijele u kontekste, ali zahtjeva se da nijedan entitet ne prelazi izvan tog konteksta.

Razlog tom problemu je pristup implementaciji ECS-a bez korištenja konceptualnih relacijskih baza, a predlaže se da se sve definira prije kompajliranja. Dodatno, kako se koristi više identifikatora za definiranje entiteta, moguće je imati entitete koji nemaju komponente, već služe čisto kao pomoćni dijelovi entiteta koji povećavaju broj direktnih akcija nad jednim entitetom. Što se tiče komponenata, često implementacije ECS-a dozvoljavaju korištenje singleton uzorka (u Entitas-u to je riješeno preko unikatnog konteksta) kako bi se podijelila komponenta na više entiteta. Kako je to definirano u baznoj implementaciji, taj problem se rješava preko trećeg sloja sustava. Također, kako se za definiciju svih dijelova ECS-a koriste apstraktne klase, moguće je proširiti bilo koju od tih klasa te nadodati željene funkcionalnosti.



Slika 52. Apstraktni prikaz arhitekture implementacije Entitas [9]

8. Zaključak

Idealna arhitektura uzorka entitetsko komponentnog sustava bi značila da je svaki objekt u aplikaciji entitet. Shvaćeno doslovno to je neostvarivo s obzirom da ne može postojati klasa entitet, već je to apstraktni koncept koji ne sadrži nikakvu podatkovnu strukturu osim samog identifikatora.

Svi podaci o entitetima se spremaju u komponente, a komponente su najjednostavniji spremnici podataka bez ikakvih metoda te se one pridružuju entitetima.

Naposljetku, definiraju se sustavi koji sadrže svu logiku igre, mogu sadržavati podatke koji su relevantni samo tom sustavu, ali ne bi smjeli pohranjivati podatke o entitetima koje obrađuju.

Međutim, strogo praćenje ovih principa je kontraproduktivno. Arhitektonske uzorke treba promatrati kao smjernice. Naprimjer, dodavanjem trećeg sloja sustava koji se definira u komponenti dozvoljava utjecanje jedne komponente na drugu čime se poništava potreba za jedinstvenim komponentama te se smanjuje dupliciranje koda kod sustava prvog sloja. Ponekad je čak i pogodno da sustavi održavaju lokalnu strukturu entiteta (kao što je primjer kod implementacije web aplikacije) kako bi se mogla pojednostaviti bolja međusobna interakcija te ta uloga u ovoj ECS implementaciji pripada lokalnim sustavima.

Izrada ove implementacije je započeta od ideje da se preuzme glavni princip željenog uzorka, odnosno ideja da se podijele metode od podataka. Kako se nije koristio predloženi standard kojeg je donio Adam Martin, umjesto pisanja vlastitog eloquent-a, odnosno jezik za vršenje upita nad konceptualnom relacijskom bazom, odabrano je korištenje apstraktnih tvornica gdje se struktura izvršavanja programa i definicija svakog pojedinog entiteta koji će se pojavljivati u aplikaciji definira prije kompajliranja programa.

Takav način će uvijek zahtijevati više boilerplate (hrv. predložak) koda i ograničuje fleksibilnost kod razvijanja aplikacije. Komponente su zaključane na nekoliko svojstava koji su jednostavni tipovi podataka i implementacija nadmeće definiran način izvođenja ECS sustava.

Za daljnji razvoj implementacije ponajprije bi trebalo preraditi komponente tako da su fleksibilnije, mogu sadržavati različite tipove podataka, da se podaci komponente ne spremaju kod entiteta, već se kreira referenca na njihov položaj u nekom globalnom polju u koji će se spremati kao struktura. Također, umjesto spremanja identifikatora entiteta u entitet, navedena referenca na neko globalno polje može poslužiti kao spremnik i za identifikatore entiteta.

9. Literatura

[1] Hansen, Hugo i Oliver Öhrström. "Benchmarking and Analysis of Entity Referencing Within Open-Source Entity Component Systems." (2020).

[2] Härkönen, Toni. "Advantages and Implementation of Entity-Component-Systems." (2019).

[3] Gregory, Jason, et al. “Runtime Object Model Architectures.” Game Engine Architecture, 1st ed., A K Peters/CRC Press, 2009, pp. 715–34, static.latexstudio.net/wp-content/uploads/2014/12/Game_Engine_Architecture-en.pdf.

[4] Martin, Adam. “Entity Systems Are the Future of MMOG Development.” Entity Systems Are the Future of MMOG Development, 3 Sept. 2007, t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1.

[5] “Introducing GameplayKit - WWDC 2015 Videos.” Apple Developer, 6 Oct. 2017, https://web.archive.org/web/20171006113323/https://developer.apple.com/videos/play/wwdc2015/608/.

[6] Barton, Seth. “Unity Unleashes Megacity Demo - Millions of Objects in a Huge Cyberpunk World.” MCV Develop, 24 Oct. 2018, www.mcvuk.com/development-news/unity-unleashes-megacity-demo-millions-of-objects-in-a-huge-cyberpunk-world.

[7] “EnTT Minecraft.” GitHub, 2018, github.com/skypjack/entt.

[8] Unity Technologies. “Unity User Manual.” Unity User Manual, docs.unity3d.com/Manual/index.html. Accessed 31 Aug. 2021.

[9] “Entitas.” GitHub, github.com/sschmid/Entitas-CSharp/wiki/Introduction-to-Entitas. Accessed 31 Aug. 2021.

10. Popis slika

11. Popis tablica