Juho Tarkkanen

Entiteetti-komponentti-järjestelmä ja läheiset oliosuuntautuneet suunnittelumallit

Tietotekniikan kandidaatintutkielma

6. toukokuuta 2020

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Juho Tarkkanen

Yhteystiedot: juho.a.tarkkanen@student.jyu.fi

Ohjaaja: Antti-Jussi Lakanen

Työn nimi: Entiteetti-komponentti-järjestelmä ja läheiset oliosuuntautuneet suunnittelumallit

Title in English: Entity component system and related object-oriented design patterns

Työ: Kandidaatintutkielma

Opintosuunta: Tietotekniikka

Sivumäärä: 28+3

Tiivistelmä: Entiteetti-komponentti-järjestelmä (ECS) on kytkentöjen purkamiseen datan ja logiikan välillä keskittyvä ohjelmoinnin malli. Koostumissuhteita vahvistamalla ECS tarjoaa hyötyjä järjestelmien ylläpitoon, laajentamiseen ja uudelleenkäyttämiseen, ja siitä voidaan havaita samankaltaisuuksia tunnettuihin olio-ohjelmoinnin suunnittelumalleihin. Suunnittelumallit ovat yleisesti hyvin kattavia kuvauksia tyypillisten ohjelmointiongelmien ratkaisutavoista. ECS:n määritelmä taasen on kovin hajanainen. Käsitys ECS:stä monipuolistuu vertailemalla sitä samankaltaisiin perinteistä oliosuuntautuneisuutta ilmentäviin suunnittelumalleihin.

Avainsanat: entiteetti-komponentti-järjestelmä, suunnittelumalli, oliosuuntautunut ohjelmointi

Abstract: Entity component system (ECS) is a programming pattern oriented towards decoupling data and logic. Well-known object oriented design patterns share similarities with ECS and by using composition ECS provides benefits in maintainability, extensibility and reuse of systems. Design patterns are usually well-documented descriptions of ways to solve common problems in programming. In this context ECS is quite sparse. Comparing ECS with similar and traditionally object-oriented design patterns helps in diversifying its description.

Keywords: entity component system, design pattern, object-oriented programming

Kuviot

Sisältö

1 Johdanto

Ohjelmoinnissa käsitellään dataa, minkä helpottamiseksi kehitetään erilaisia abstraktioita. Laajasti hyödynnettävässä olio-ohjelmoinnissa abstraktiota käytetään datan välitykseen muun muassa luomalla olioiden välisiä merkityssuhteita. Esimerkiksi internet-lomake voisi *omistaa* monia tekstikenttiä ja numerokentät *olisivat osajoukko* kaikista mahdollisista tekstikentistä. Olio-ohjelmoinnin mahdollistamalla perinnällä kaikista mahdollisista tekstikentistä muodostettaisiin hierarkia, jonka juuressa olisi alkuperäinen yleistyvä tekstikenttä. Tällainen menettely yhtenäistää hierarkiassa esiintyviä rajapintoja ja parantaa operaatioiden uudelleenkäyttöä (Gamma ym. 1995, s. 18-19).

Perinnän yltiöpäinen käyttö voi kuitenkin johtaa ohjelmakoodin jäykkyyteen, jolloin omistussuhteiden suosiminen tekisi ohjelmien osista uudelleenkäytettävämpiä ja yksinkertaisempia (Gamma ym. 1995, s. 20). Uudelleenkäytettävyyden ja ylläpidon vaikeutuminen ovat Wiebuschin ja Latoschikin (2015) mukaan erityisesti näkyvissä reaaliaikaisissa interaktiivisissa järjestelmissä (engl. realtime interactive systems, RIS), joissa monet erilaiset toiminnot, käyttäjän syötteestä fysiikan simulointiin, sisältävät kukin useita osajärjestelmiä (Fischbach, Wiebusch ja Latoschik 2017). Omistussuhteilla koostamisen vahvistamiseksi ovat RIS:t ja niiden kaltaiset ohjelmistot hyödyntäneet entiteetti-komponentti-järjestelmänä tunnettua suunnittelumallia (Wiebusch ja Latoschik 2015; Fischbach, Wiebusch ja Latoschik 2017; Hodson ja Millar 2018).

Entiteetti-komponentti-järjestelmä (engl. entity component system, tässä tutkielmassa myöhemmin ECS) on datan ja logiikan erottelua tarjoava ohjelmoinnin malli, joka vahvojen koostumissuhteiden muodostamiseen erikoistuen pyrkii muun muassa parantamaan ohjelman ylläpitoa ja muuntautuvaisuutta. Tässä tutkielmassa kirjallisuuskatsauksen avulla pyritään tuomaan esille ECS:n hyötyjä ja haittoja. Lisäksi tarkastellaan ECS:ään liitettäviä, ohjelmistokehityksessä yleisiä suunnittelumalleja (engl. design patterns), minkä tarkoituksena on edelleen selventää käsitystä ECS:stä sekä sen käyttämisestä olio-ohjelmoinnissa.

Tutkielma alkaa ECS:n kuvauksesta ja etenee kohti pohtivampaa mallin tarkastelua muihin suunnittelumalleihin perustuen. Luvussa 2 esitellään ECS:n yleistä rakennetta, käyttötarkoi-

tusta ja toteutuksen periaatteita sekä kerrotaan ECS:tä erityisesti koituvista hyödyistä ja haitoista. Luvussa 3 kuvaillaan lyhyesti suunnittelumalleja käsitteenä sekä tarkastellaan ECS:ää tarkemmin niiden suunnittelumallien kannalta, jotka erityisesti näyttäytyvät ECS:lle läheisiltä. Luvussa 4 pohditaan ECS:n sopivuutta yleisesti käytettävään ja vahvemmin hierarkkiseen olio-ohjelmointiin erilaisten näkemysten myötä. Lopuksi luvussa 5 kootaan yhteen tutkielmassa käytettävistä näkökulmista nousseet johtopäätökset, sekä tuodaan esille jatkotutkimuskohteita.

2 Entiteetti-komponentti-järjestelmä

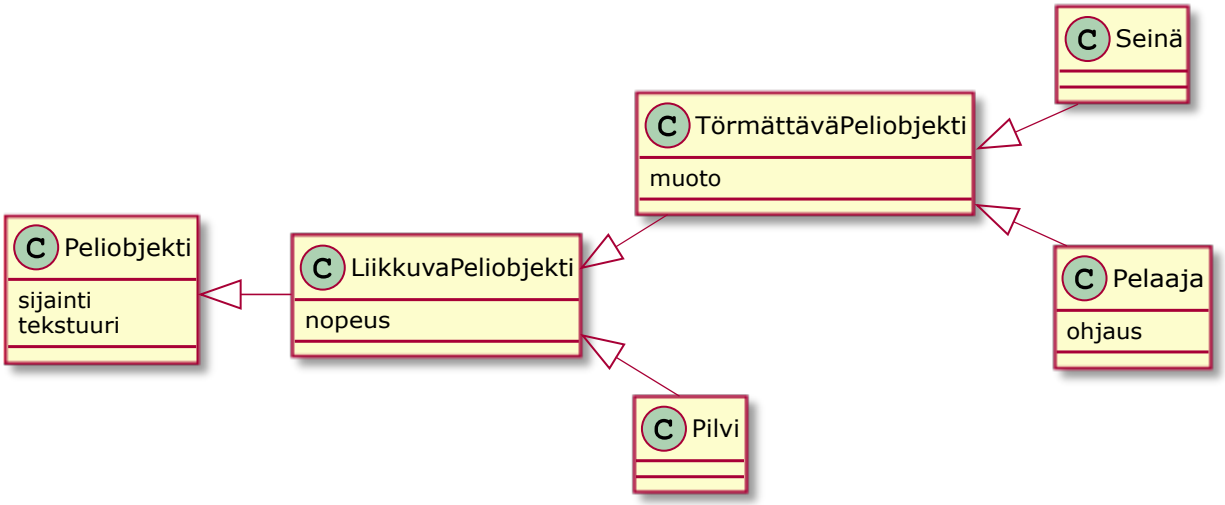
ECS on melko löyhästi määritelty kuvaus ohjelmoinnissa käytettävästä mallista (Raffaillac ja Huot 2019; Berge ym. 2014; Hodson ja Millar 2018) ja sille tyypillinen koostaminen perustuu käsitteisiin *entiteetti*, *komponentti* ja *järjestelmä*. Yleiset ECS:n käyttökohteet ovat ohjelmallisten tavoitteidensa osalta keskenään samankaltaisia.

2.1 Käyttökohteiden taustaa

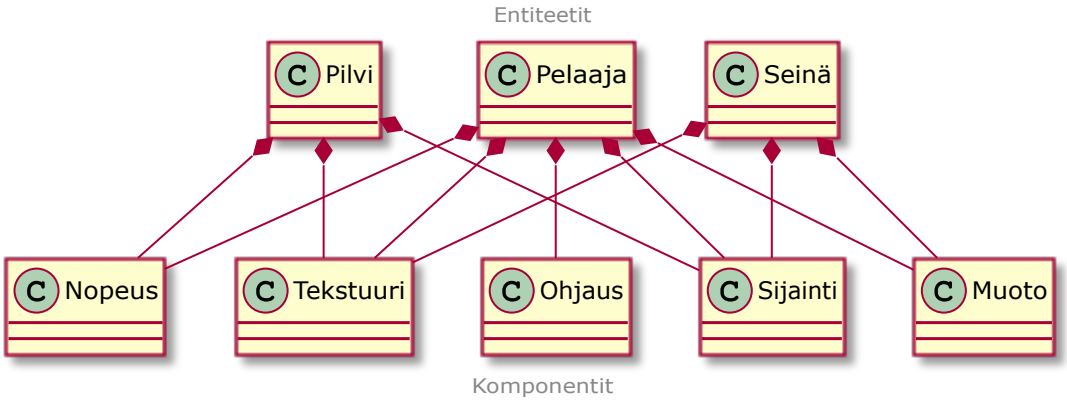
ECS, tai sen perusperiaatteita vastaava malli on näkyvimmin muotoutunut videopelien yhteydessä (Martin 2007b; West 2007; Bilas 2002) ja sitä on hyödynnetty esimerkiksi Unity-pelimoottorissa (Meijer 2019). ECS:n käyttö on akateemisessa kirjallisuudessa usein myös yhteydessä reaaliaikaisiin interaktiivisiin järjestelmiin. Nämä RIS:t keskittyvät virtuaalitodellisuuksien, pelimoottoreiden ja reaaliaikaisten simulaatioiden tavoin tuottamaan graafista näkymää, joka vaatii muun muassa fysiikan simulointia ja käyttäjän syötteen tulkitsemista (Lange, Weller ja Zachmann 2016). Niissä halutaan hyödyntää ECS:ää pääasiassa saavuttamaan datasuuntautuneen suunnittelun suorituskykyä (Fontana ym. 2017), ajonaikaista muuntautuvuutta (Garcia ja Almeida Neris 2014) sekä kytkentöjen (engl. coupling) purkamista (Wiebusch ja Latoschik 2015).

2.2 Määrittely

ECS pyrkii estämään ohjelmiin muodostuvista kytkennöistä syntyviä ongelmakohtia. Näitä ovat esimerkiksi uudelleenkäytettävyyden perustuminen olioiden perinnälle ja olioiden vahvasta tyypityksestä johtuva jäykkyys ajonaikaisissa muutoksissa, jotka voivat hidastaa ohjelman kehitystyötä erityisesti uusia ominaisuuksia lisättäessä. Esimerkiksi Langen, Wellerin ja Zachmannin (2016) mukaan perinnän käyttäminen polymorfismin saavuttamiseksi voi tuottaa ohjelmistokehityksen jäykkyyttä syvillä ja leveillä luokkahierarkioilla kuten kuviossa 1. ECS:n koostumissuhteilla tällaista hierarkiaa pyritään madaltamaan kuvion 2 osoittamalla tavalla.



Kuvio 1. Esimerkki perinteisillä menetelmillä muodostuvasta syvästä perintärakenteesta luokkakaaviona. Kuvion hierarkiaa joudutaan muokkaamaan, ettei Seinä-luokkaan tarpeettomasti sisällytettäisi LiikkuvaPeliobjekti:n nopeus-ominaisuutta.

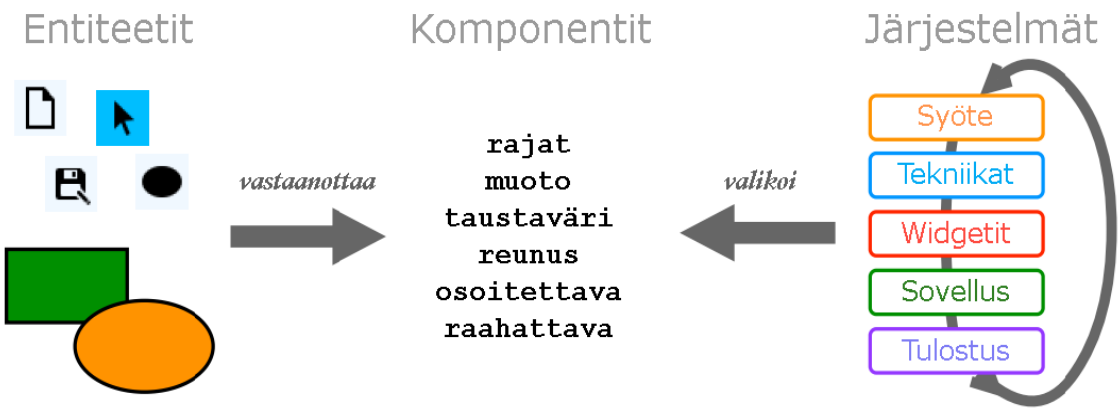


Kuvio 2. Luokkahierarkia (kuvio 1) purettuna entiteetteihin ja komponentteihin ECS:n tavoin koostamalla.

ECS:n määritelmä alkaa luonnollisesti sen nimessä esiintyvistä käsitteistä. Näiden käsitteiden määritelmät näyttävät usein seuraavan videopelialan blogikirjoituksia kuten West (2007) ja Martin (2007b), joista jälkimmäinen määrittelee käsitteet entiteetti, komponentti ja järjestelmä. Koska ECS:llä ei ole virallista määritelmää, Berge ym. (2014, suomennokset minun) antavat ECS:n keskeisimmistä käsitteistä oman kuvauksensa perustuen Martiniin (2007a) ja Westiin (2007):

- ”Entiteetti on yksittäinen yleiskäyttöinen objekti, joka ei sisällä dataa eikä toiminnallisuutta (ts. metodeja). Sen ainoana tarkoituksena on toimia peliobjektin tunnisteena.”
- ”Komponentit liitetään entiteetteihin ja sisältävät raa’an datan, mutta eivät toiminnallisuutta. Niiden tarkoituksena on määrittää [peli]objektille tietty muoto ja tapa kuinka se toimii ympäristössään. Komponentin liittäminen entiteettiin asettaa tämän entiteetin edustamaan tiettyä muotoa. Entiteettiin voi olla liitettynä useita komponentteja ja kukin komponentti voidaan liittää useaan entiteettiin.”
- ”Järjestelmä määrittelee todellisen toiminnallisuuden. Yleensä jokaista komponenttia (muotoa) vastaa yksi järjestelmä, joka mallintaa ja toteuttaa pelin globaalin interaktion ja toiminnallisuuden.”

Kuviosta 3 ilmenevät karkeasti entiteettien, komponenttien ja järjestelmien väliset suhteet ja kunkin käsitteen merkitys graafisessa ohjelmassa.

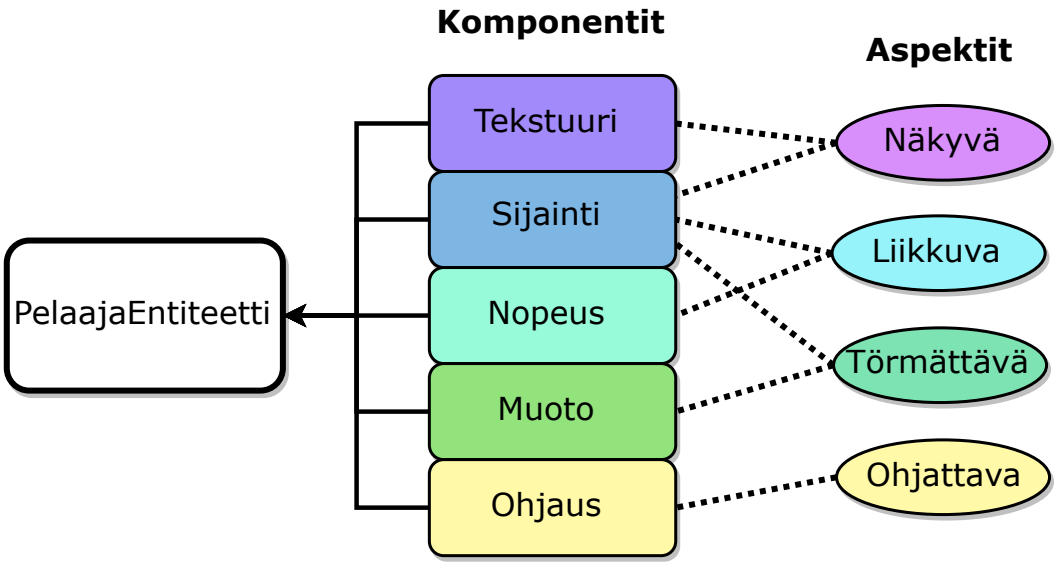


Kuvio 3. ECS:n keskeisimpien käsitteiden vuorovaikutus. Entiteetit saavat ominaisuuksia komponenttien myötä ja järjestelmät käsittelevät yhteensopivia entiteettejä näihin ominaisuuksiin perustuen. (mukaellen Raffaillac ja Huot 2019)

ECS:ssä entiteetit toimivat ohjelman dataa, komponentteja yhdistävinä tunnisteina. Verrattaessa olio-ohjelmointiin entiteetit ovat lähellä oliota sekä lukumäärässään (Martin 2007b), että ollessaan eräänlaisia kapseloituja ohjelman toiminnan perusyksiköitä. Informaatiota ei tosin pyritä sitomaan luokkarajapinnan taakse, vaan kapselointia edustaa ECS:n vaatimus entiteetin tuntemisesta sen komponentteihin pääsemiseksi ja siten datan käsittelemiseksi yhteensopivalla tavalla. Oliosuuntautuneesta kapseloinnista ECS:ssä mainitsevat Wiebusch ja Latoschik (2015) ja tätä tarkastellaan myöhemmin luvussa 4.

Bergen ym. (2014) määritelmässä (luku 2.2) esiintyviä komponenttien määrittämiä ”muotoja” (engl. aspect) kutsutaan selvyyden vuoksi tässä tutkielmassa *aspekteiksi*. Entiteetteihin liitettävät komponentit edustavat jotakin aspektia ja entiteettien toiminta määrittyy aspektien mukaan. Komponenttien ja aspektien suhdetta entiteettien toiminnallisuuteen selvennetään kuviossa 4. Komponentteja voidaan liittää tai niitä voidaan poistaa entiteeteistä ohjelman ajon aikana, mikä mahdollistaa entiteettien ja koko ohjelman toiminnan dynaamisen muuttamisen (Raffaillac ja Huot 2019; Lange, Weller ja Zachmann 2016; Garcia ja Almeida Neris 2014).

Toimintalogiikka määritellään järjestelmissä, joiden lukumäärä kutakuinkin vastaa ECS:ää hyödyntävän ohjelman tai ohjelman osan käsittämiin komponentteihin liittyviä eroteltavia toimintoja. Järjestelmät käyvät jatkuvasti läpi kaikkia ohjelman entiteettejä ja kullekin järjestelmälle valitaan yhteensopivat entiteetit aspekteihin perustuen (Raffaillac ja Huot 2019; Wiebusch ja Latoschik 2015). Ohjelman toiminta saattaa vaatia erillisten järjestelmien yhteistyötä, jolloin järjestelmien suoraan toisiinsa yhdistämisen sijaan ne on järjestetty sopivasti ajettaviksi (Raffaillac ja Huot 2019).



Kuvio 4. Esimerkki entiteetistä, johon liitetyt komponentit muodostavat sille neljä eri aspektia.

2.3 Toteutuksista

2.3.1 Laajennettu käsitteistö

ECS:ään voidaan liittää entiteetin, komponentin ja järjestelmän lisäksi muitakin käsitteitä. Tällaiset lisäkäsitteet pääasiallisesti kuvaavat ECS:ään tarvittavia toimintoja, jotka lopulta mahdollistavat suunnittelumallilla tavoitellun kokonaisuuden toteuttamisen. Tutkimuksessaan Raffaillac ja Huot (2019) nostavat esille seuraavat ECS-toteutuksissa alati toistuvat, mutta käsitteistöön vakiintumattomat elementit:

- *Kuvailutermi* (engl. descriptor) - Komponentteihin perustuvia ehtoja; perustavanlaatuinen mekanismi, jolla järjestelmät tunnistavat käsittelemänsä entiteetit.
- *Valinta* (engl. selection) - Joukko saman kuvauksen omaavia entiteettejä. Valinta päivitetään aina, kun entiteeteiltä poistetaan tai niille lisätään komponentteja.
- *Konteksti* (engl. context) - Kokoaa tiettyyn ympäristöön tai ”maailmaan” tarvittavat palaset; *valintoja* tarjoava kokoelma entiteettejä, komponentteja ja globaaleja muuttujia.
- *Kaavain* (engl. template) tai *tehdas* (engl. factory) - Ennalta määritelty tapa luoda en-

titeettejä oletusarvoisilla komponenteilla.

Fontanan ym. (2017) kuvailu ”entiteettijärjestelmästä” selkeästi sisältää vastaavuuksia Raffaillacin ja Huotin (2019) käsitteisiin *valinnasta* ja *kaavaimesta*. ECS-suunnittelumallia hyödyntävässä ohjelmassa entiteettijärjestelmä Fontanan ym. (2017) mukaan takaa kunnollisen pääsyn entiteetteihin ja komponentteihin. Tässä entiteettijärjestelmä luo ja tuhoaa entiteettejä sekä hallinnoi komponenttien kokoelmien eheyttä. Entiteettijärjestelmä luodessaan entiteetin myös asettaa sille tunnisteen sekä oletusarvoisina kaikki tähän entiteettijärjestelmään liittyvät komponentit. Kokoelmien eheyden jatkuva ylläpido on yhteydessä valinnan toimintaan ja entiteettien oletusarvoinen tuottaminen taasen vastaa kaavainta. Lisäksi ECS:n perusmalliin sisältyvä ajatus komponenttien edustamasta aspektista on samankaltainen kuin Raffaillacin ja Huotin (2019) *kuvailutermi*. Molemmat liittyvät entiteettien ominaisuuksien tunnistamiseen ja eräänlaiseen järjestelmän tarvitsemaan tyypitykseen. Näiden käsitteiden eriävä nimeäminen johtunee ECS:n karkeasta määrittelystä, mutta niiden toistuminen sekä perusmallissa että toteutuksissa korostaa entiteettien tunnistamisen keskeisyyttä.

2.3.2 Esimerkkitoteutus

Algoritmissa 2.1 kuvataan Python 3 -ohjelmointikieltä käyttäen yksinkertainen komentoriviohjelma, joka esittelee ECS:n pääpiirteitä. Algoritmin 2.1 lyhentämätön ohjelmakoodi on nähtävissä liitteessä A.

Algoritmi 2.1. ECS-esimerkki (mukaellen *Implementing Component-Entity-Systems* 2013)

```
# Entiteetin rekisteröinti ja simulaatioon lisääminen
def lisaa_entiteetti(entiteetin_komponentit):
    for aspekti in entiteetin_komponentit.keys():
        komponentit[aspekti][lisaa_entiteetti.e_id] = entiteetin_komponentit[aspekti]
    # Entiteetin aspekteina käytetään liitettyjen komponenttien nimitysten joukkoa
    entiteetit[lisaa_entiteetti.e_id] = set(entiteetin_komponentit.keys())
    lisaa_entiteetti.e_id += 1

# ...
```

```
# Käytettävät järjestelmät:
# Entiteettien kulkusuuntaa käsittelevä järjestelmä
def ohjain():
    ohjain_aspekti = set(["suunta", "ohjain"])
    for entiteetti, aspekti in entiteetit.items():
        # Aspektin/kuvailutermin käyttötarkoitus (ks. luvut (2.2) ja (2.3.1))
        if ohjain_aspekti.issubset(aspekti):
            # Tässä järjestelmässä hyödynnetään komponentiksi asetettua funktiota
            uusi_suunta = komponentit["ohjain"][entiteetti]()
            komponentit["suunta"][entiteetti] = uusi_suunta

# ...

# Ohjelman alku:
# Käytettävät komponentit ja siten aspektit määritellään
# ja entiteettejä kokoava tietorakenne sekä juokseva tunniste alustetaan.
komponentit = { "elama": dict(), "suunta": dict(), "ohjain": dict(), "sijainti": dict() }
entiteetit = dict()
lisaa_entiteetti.e_id = 0

# Entiteettejä lisätään maailmaan tehdasfunktioihin ennalta määritetyillä
# oletusarvoisilla, tai parametrien mukaisilla, komponenteilla (ks. kaavain luvussa (2.3.1)).
# Tässä esimerkissä tehtaat palauttavat hakurakenteen halutuista komponenteista.
lisaa_entiteetti( luo_puu((2,1)) )
# Pelaajan tunniste talletetaan tässä ohjelman kulun kontrolloimiseksi
pelaaja_entiteetti = lisaa_entiteetti.e_id
lisaa_entiteetti( luo_pelaaja() )

# Käynnistetään simulaation pääsilmukka
while(komponentit["elama"][pelaaja_entiteetti] > 0):
    # Entiteettejä voidaan lisätä osaksi suorituksessa olevaa simulaatiota
```

```
lisaa_entiteetti( luo_pilvi() )
# Järjestelmät ajetaan simulaatiolle sopivassa järjestyksessä
tulosta_tilanne()
ohjain()
liike()
elama()
```

2.4 Hyötyjä

ECS:n tapa käsittää data erillisinä komponentteina, sekä vahvoja koostumissuhteita käyttäen erotella nämä komponentit myös niillä suoritettavasta logiikasta, johtaa myönteisiin vaikutuksiin niin ohjelman toiminnassa kuin myös sen kehitystyössä.

2.4.1 Ohjelman toiminnallisen laadun parannuksia

Komponenttien vaihtaminen ajonaikaisesti kullekin entiteetille mahdollistuu niiden ollessa rakenteellisesti erillisiä toisistaan vahvojen koostumissuhteiden vuoksi (Wiebusch ja Latoschik 2015). Garcian ja Almeida Nerisin (2014) mukaan ajonaikaisen muuntautuvuuden myötä voidaan myös alustaa entiteettejä ulkoisen tiedoston perusteella ohjelman suorituksen aikana, jolloin sovelluksen käyttäjäprofiilit voidaan helpommin määrittää tekstitiedostoissa ja muuttaa entiteettien käyttäytymistä koskematta itse ohjelmakoodiin.

Raffaillacin ja Huotin (2019) mukaan yhden järjestelmän ajaminen kokoelmalle entiteettejä mahdollistaa tehokkaampia suoritustapoja kuin metodien kutsuminen peräkkäin jokaiselta oliolta. Heidän mukaansa ohjelman operaatioiden rinnakkaistuminen helpottuu, kun järjestelmät käsittelevät entiteettien kokoelmia. Suorituksen rinnakkaistuminen mahdollistaa tietokoneen suorittimen tehokkaamman hyödyntämisen. Entiteettien yhtäaikaiseen saavuttamiseen perustuen, Raffaillac ja Huot (2019) esittävät erilaisten optimointimenetelmien käyttämisen helpottuvan. Heidän mukaansa esimerkiksi graafisten käyttöliittymien asemointiongelmiin voidaan entiteettien viestinvälitysoperaatioiden sijasta käyttää vaihtoehtoisia menetelmiä. Lisäksi myös käyttäjän näkyviin piirrettävä sisältö voidaan yksinkertaisin menetelmin kasata taulukoiksi tietokoneen grafiikkasuorittimen käyttöön.

Kun käsiteltävä data ohjataan suorittimen käyttöön suuremmissa erissä, datan säilytys keskusmuistia nopeammassa välimuistissa johtaa sovellusohjelman kannalta merkityksellisten loogisten operaatioiden suorittamiseen lyhyemmässä ajassa (Acton ja Games 2014). *Datasuuntaunut suunnittelu* perustuu karkeasti kuvattuna ohjelman käyttämän muistin fyysiseen yhtenäistämiseen tällaista tarkoitusta varten. Fontanan ym. (2017) mukaan ECS erityisesti soveltuisi datasuuntautuneeseen suunnitteluun, koska dataa voidaan ECS:lle ominaisella tavalla käsitellä erillisten komponenttien yhtenäisinä joukkoina. Kun sopivat järjestelmät ajetaan kerralla sopiville joukoille, voitaisiin tietokoneen suorittimen välimuistia hyödyntää kattavammin, johon ECS:llä saavutettava suorituskyky Fontanan ym. (2017) mukaan perustuu.

2.4.2 Ohjelmistokehitystyön parannuksia

Syviä perintähierarkioita voidaan ECS:n vahvoilla koostumissuhteilla madaltaa, mikä vähentää perinnästä johtuvien kompromissien tai yliluokkien uudelleensuunnittelun tarvetta ohjelman entiteettejä muodostettaessa (Lange, Weller ja Zachmann 2016). Entiteetille tarvitsee vain valita halutut komponentit, joiden perusteella se tulee osaksi sopivaa ohjelman logiikkaa ilmentävää järjestelmää.

Raffaillacin ja Huotin (2019) mukaan ECS:n järjestelmät keskittävät entiteettien toiminnallisuutta voimakkaasti ja auttavat siten esimerkiksi vähentämään ohjelmakoodin muutoksien sekä mahdollisesti ongelmallisten takaisinkutsujen käyttämisen määrää. Koska järjestelmät lisäksi ovat omia irrallisia osiaan, myös ohjelmassa käytettävien algoritmien keskinäinen vaihtaminen ja simulaatioiden uudelleenkäyttäminen Langen, Wellerin ja Zachmannin (2016) mukaan helpottuu. Uudelleenkäytettävyyden vahvistuminen ECS:llä esiintyy Wiebuschin ja Latoschikin (2015) mukaan erityisesti, kun mallin yhteydessä käytetään entiteettejä takautuvasti luokittelevia merkityspiirteitä (engl. semantic traits).

2.5 Ongelmakohtia

ECS:ssä Raffaillacin ja Huotin (2019) mukaan ohjelman yksittäisten toimintojen ylläpitäminen vaikeutuu, sillä järjestelmien hyödyt perustuvat erityisesti usean samankaltaisen en-

titeetin kertaluontoiseen käsittelemiseen. Heidän mukaansa ECS ei myöskään tarjoa selkeää keinoa, jolla entiteetit yhdistetään komponenttien perusteella sopiviin järjestelmiin. Myös Wiebusch ja Latoschik (2015) kuvaavat riippuvuuksien selvittämisen ongelmallisuuksia järjestelmien ja entiteettien välillä. Entiteettien kuvaaminen olioilla, esimerkiksi käännösaikaisen tyypityksen hyödyntämiseksi, johtaa heidän mukaansa jäykkiin perintähierarkioihin ja vaihtoehtoisesti entiteettien ja komponenttien dynaaminen toisiinsa liittäminen vaatisi erillistä ajonaikaista tyypintarkastusta.

Datasuuntautuneeseen suunnitteluun liittyen Fontana ym. (2017) mainitsevat, ettei fyysisen muistin järjestely ole olioiden välisten suhteiden ajatteluun tottuneille ohjelmoijille luontaista. Vaikka tämän ongelman helpottamiseen ECS heidän mukaansa soveltuu, se ei hävitä datan paikallisuutta rajoittavien olosuhteiden ilmentymistä. Ensimmäisellä testillään Fontana ym. (2017) osoittavat ECS:llä toteuttamansa järjestelmän yltävän merkittävästi parempaan suorituskykyyn verrattuna oliosuuntautuneeseen toteutukseensa, mutta toisessa ja dataa hajautetummin ilmentävässä testissä ECS-pohjainen järjestelmä suoriutuikin oliototeutusta hieman hitaammin. Heidän mukaansa tämä ero johtui suoraan toisessa testissä käsiteltävän datan jakautumisesta toisistaan erillisiin entiteettijärjestelmiin.

ECS:n määrittelyn suurpiirteisyys on Raffaillacin ja Huotin (2019) mukaan johtanut eriäviin tulkintoihin mallista. Lange, Weller ja Zachmann (2016) käsittelevät, kuinka ECS:n perusmalli ei esimerkiksi sisällä ratkaisuja sitä usein hyödyntävien RIS:ien vaatimaan rinnakkaiskäsittelyyn, jolloin toteutuksessa joudutaan turvautumaan suunnittelumallia käsittelemättömään kirjallisuuteen. Tämä ECS:n kuvauksen hajanaisuus voi vaikeuttaa sen käyttöönottoa ja tehokasta hyödyntämistä. Esimerkiksi Gamma ym. (1995) kuvaavat ohjelmoinnin suunnittelumalleja kattavasti, sisällyttäen tietoa muun muassa toteutukseen käytettävistä luokista, sopivista sovelluskohteista ja seurauksista sekä eri mallien välisistä suhteista.

3 Entiteetti-komponentti-järjestelmä ja suunnittelumallit

Tässä luvussa esitellään ECS:ää käsittelevässä kirjallisuudessa mainittuja suunnittelumalleja ja arvioidaan niiden yhteyksiä ECS:ään. Suunnittelumallien tarkastelun halutaan auttavan käsittämään ECS:n periaatteita helpommin, antaen erilaisia ja mahdollisesti tunnistettavampia näkökulmia käsiteltäviin ongelmiin ja niiden ratkaisuihin. Tämä on perusteltua, sillä ECS:n määrittely akateemisessa yhteydessä vaikuttaa tutkielmassa käsitellyn kirjallisuuden perusteella suunnittelumalleja löyhemmältä. Vähäinen kuvaus johtunee ECS:n suhteellisesta uutuudesta tai kenties vahvasti erikoistuneesta käyttötarkoituksesta RIS:ien kaltaisiin sovelluksiin.

3.1 Suunnittelumalleista yleisesti

Suunnittelumallit yleisesti ovat ohjelmointityössä käytettäviä ratkaisutapoja yleisille ohjelmointiongelmille. Gamman ym. (1995) kokoamat suunnittelumallit ovat olio-ohjelmoinnissa uudelleenkäytettävyyttä korostavia tapoja, joita on pitkään käytetty ammattimaisessa kontekstissa, täten vahvistaen asemansa hyödyllisinä välineinä ohjelmoinnissa. Näillä suunnittelumalleilla pyritään helpottamaan ohjelmistokehityksessä samalla kuhunkin ohjelmaan erikoistuvien, että käyttöön yleistyvien osien tuottamista, sekä vähentämään toistuvien ongelmien uudelleenratkaisemisen tarvetta.

Gamman ym. (1995, s. 3) mukaan suunnittelumalli yksinkertaisimmillaan koostuu seuraavista osista: nimi kiteyttää lyhyesti mallin ratkaiseman ongelman, ratkaisutavan ja tulokset; ongelma kertoo millaisessa tilanteessa mallia tulisi käyttää; ratkaisu käsittää mallin rakenteen, vastuut, suhteet ja yhteistyön muiden ohjelman osien kanssa; ja seuraukset kertovat mallin käytön tuloksista ja vaatimista kompromisseista. ECS ei kuitenkaan vaikuta olevan tällä tavoin keskitetysti ja seikkaperäisesti akateemisessa kontekstissa kuvattu.

3.2 ECS:n yhteyksiä sille läheisiksi kuvattuihin suunnittelumalleihin

3.2.1 Komponentti

ECS:n komponentti-käsitteestä erillinen Komponentti-suunnittelumalli (engl. Component) on esitelty pelialan kirjoituksessa Nystrom (2014). Komponentin tarkoituksena on Nystromin (2014) mukaan eristää liian suureksi kasvavan luokan erilaiset toiminnallisuudet omiksi komponenttiluokikseen. Tällä tavalla osiin purettava luokka muuttuu komponenttiensa säiliöksi ja on helpompi ylläpitää.

Wiebusch ja Latoschik (2015) viittaavat Komponenttiin, mainitessaan ECS:lle vahvasti läheisistä suunnittelumalleista. Vahva yhteys on selvästi huomattavissa: Oliot Komponentti-suunnittelumallissa muodostetaan niille halutuilla ominaisuuksilla ja kukin näistä ominaisuuksista edustaa suoraan ohjelmassa tapahtuvaa konkreettista toimintaa, kuten käyttäjän syötteeseen vastaamista, fysiikoihin reagoimista tai näkyviin piirtymistä (Nystrom 2014). Tämä vastaa ECS:n entiteettien keräämiä komponentteja, joista järjestelmät valitsevat edustamaansa toimintaan sopivat. Kummassakin suunnittelumallissa on siten tarkoituksena luoda yleiskäyttöisiä objekteja, joihin voidaan koostamalla liittää mitä vain haluttua käytöstä.

Selvin ero ECS:ään on Komponentti-suunnittelumallissa komponenttien oliotyyppinen kapselointi. Komponentti Nystromin (2014) esimerkissä muodostetaan perimällä abstraktista luokasta objekteihin liitettäville komponenteille yhteinen metodi niiden sisältämän datan päivittämiseksi. ECS:ssä tällainen datan päivittäminen erotettaisiin vahvemmin erillisiin järjestelmiin, jolloin entiteetin ei itse tarvitse ylläpitää komponenttiensa tilaa kutsumalla niiden metodeja. Nystromin (2014) kuvaus Komponentti-suunnittelumallista ei näin myöskään saavuttaisi ECS:n luvussa 2.4.1 listattuja hyötyjä ajonaikaisesta muuntautuvuudesta tai suoraviivaisemmasta prosessoinnista.

Komponentista Nystrom (2014) vielä mainitsee sen muistuttavan Strategia-suunnittelumallia (Gamma ym. 1995) sillä keskeisellä erolla, että komponentti-oliot ovat strategia-olioita useammin tilallisia.

3.2.2 Strategia

Gamma ym. (1995, s. 315) luonnehtivat Strategia-suunnittelumallia (engl. Strategy) samaan tehtävään sopivien algoritmien joukon määrittelynä, josta voidaan tarpeen mukaan valita käyttötarkoitukseen sopivin. Olioon voidaan näin kytkeä monenlaisia rajapinnaltaan samanlaisia algoritmeja ilman niiden toteutuksen ja valintaprosessin sisällyttämistä olion luokkaan (Gamma ym. 1995, s. 317-318).

Nystromin (2014) Komponenttia ja Strategiaa yhdistävä perusperiaate objektien ja käytöksen erottelemisesta toistuu myös ECS:n ja Strategian välillä, josta Wiebusch ja Latoschik (2015) mainitsevat. Gamman ym. (1995, s. 317-318) kuvauksessa algoritmit kapseloidaan strategia-olioihin, joita voitaisiin verrata ECS:n järjestelmiin molempien toimiessa pääasiallisesti syötteenä annettavan datan perusteella. Strategia välittää datan konteksti-olioiden kautta ja ECS mahdollistamalla komponentteihin pääsyn entiteettien kautta.

Strategian keskeisesti mahdollistama algoritmien *ajonaikainen* vaihtelu ei vaikuta ilmenevän ECS:ssä. Kuitenkin ECS:n järjestelmien tapa suorittaa toimintoja niille yhteensopiviin entiteetteihin perustuen mahdollistaa järjestelmien keskinäisen vaihtamisen, kun haluttu toiminta ja komponenttien data ts. järjestelmän konteksti pysyy samana.

3.2.3 Tarkkailija

Gamma ym. (1995, s. 293, suomennos minun) määrittelevät Tarkkailija-suunnittelumallin (engl. Observer) tarkoituksena ”määritellä yhdestä moneen suhde, jossa yhden olion tilan muutos johtaa sitä tarkkailevien olioiden automaattiseen päivittämiseen [uuden tilan perusteella]”; tarkkailtava kohde (engl. subject) ilmoittaa tilamuutoksestaan tarkkailijoilleen (engl. observers), jotka sitten päivittyvät haluamillaan tavoilla.

Wiebuschin ja Latoschikin (2015) mukaan ECS Tarkkailija-suunnittelumallin tavoin tavoittelee datan ja logiikan erottelua. Tämä erottelu ilmenee tarkkailijoiden tapana toimia (logiikka) itsestään erillisen kohteen sisältämän tilan (data) perusteella. Tarkkailija-suunnittelumallia esiintynee ECS:ssä Raffaillacin ja Huotin (2019) selvityksen perusteella tilamuutosten kuuntelussa (engl. listener), kuten entiteettien välisissä viestintätapahtumissä, johon Elvezio, Sukan ja Feiner (2018) taasen viittaavat. Tällöin tarkkailun kohteena toimivalle entiteetille voi-

taisiin asettaa komponentti, joka listaa kaikki sitä tarkkailevat entiteetit. Kyseisen komponentin määrittämään kohde-aspektiin yhdistetty järjestelmä, jota algoritmin 3.1 pseudokoodi kuvaa, kävisi läpi kohteen tarkkailijat ja ilmoittaisi sopiville tarkkailijoille niiden kuunteleman tilan muutoksesta kohteessa. Tarkkailijat, jotka taasen tunnistuisivat tarkkailija-aspektista, toimisivat sitten niihin asetetun logiikan ja kohteen tilan perusteella.

Algoritmi 3.1. ECS-mallia mukaileva esimerkki Gamman ym. (1995) kuvaaman Tarkkailija-mallin toteuttamisesta

```
# Konkreettiset entiteetit ja komponentit merkitty etuliitteillä "e_" ja "k_",
# aspektit/kuvailutermit (ks. luvut 2.2 ja 2.3.1) vastaavasti "a_"
def TarkkailijaJarjestelma():
    # Entiteetit toimii kuten valinta (ks. luku 2.3.1)
    for e_Kohde in Entiteetit[a_Kohde]:
        # Tarkkailijat on jaettu kohteena olevan entiteetin mukaan
        for e_Tarkkailija, (k_Tila, logiikka) in Komponentit[a_Tarkkailija][e_Kohde].items():
            # Tarkistetaan vastaako tarkkailijan tuntema tila kohteen tämänhetkistä tilaa
            if (k_Tila != Komponentit[k_Tila.Aspekti][e_Kohde]):
                # Tarkkailija reagoi halutulla tavalla
                logiikka(e_Tarkkailija, Komponentit[k_Tila.Aspekti][e_Kohde])
                # Tarkkailijan tila synkronoituu kohteen mukaiseksi
                Komponentit[a_Tarkkailija][e_Kohde][e_Tarkkailija] =
                    (Komponentit[k_Tila.Aspekti][e_Kohde], logiikka)
```

Algoritmissa 3.1 esitelty toimintatapa myös seuraisi Gamman ym. (1995, s. 294-296) kuvaamaa Tarkkailija-mallia, jossa kohteella voi olla tiedossaan useita tarkkailijoita ja tarkkailijat sisältävät oman reagointilogiikkansa sekä päivittyvät kohteensa tilan perusteella. Algoritmissa 3.1 ei toisaalta toteudu kohteen itsenäisesti suorittamaa yleislähetystapahtumaa, vaan siinä yksitellen tarkastetaan yhteneväisyys jokaisen tarkkailijan kanssa ECS:n järjestelmien tasolla. Tällainen viestintäjärjestelmä myös määrittyy kahden *erillisen* komponentin myötä, jolloin merkityksellisten suhteiden muodostaminen ja purkaminen täytyy suorittaa näiden kummankin suhteen ja komponenttien käyttö monimutkaistuu.

3.2.4 Vierailija

Gamma ym. (1995, s. 333) esittävät Vierailija-suunnittelumallissa (engl. Visitor) määriteltävän kaksi hierarkiaa: elementit ja niitä käsittelevät vierailijat. Gamman ym. (1995, s. 331) mukaan Vierailija-suunnittelumallin avulla voidaan erilaisten elementtien kokoelmalle määritellä uusia yhteisiä operaatioita ilman elementtien määritelmien muuttamista. Elementtien yli iteroitaessa kutsutaan kunkin elementin toteuttamaa metodia, joka mahdollistaa haluttua operaatiota edustavan vierailija-olion pääsyn elementti-olion sisältämään dataan.

Useat erilaiset toiminnot samaan luokkaan liitettynä aiheuttavat sekavuutta ja tällöin olisi parempi, jos toiminnot voitaisiin erottaa käyttämiensä luokkien rajapinnasta (Gamma ym. 1995, s. 331-332). Wiebuschin ja Latoschikin (2015) mukaan Vierailija-suunnittelumalli liittyy ECS:ään tällaisen datan ja logiikan erottelun vuoksi. Myös Vierailija-suunnittelumallin tapa hallinnoida elementtien kokoelmaa niille operaatioita suorittaen muistuttaa ECS:ää: Vierailija-suunnittelumallia hyödyntävä ohjelma käyttää jotakin haluttua operaatiota kokoelman elementteihin ja itse operaation toteutus voi vaihdella perustuen kunkin elementin tarjoamiin ominaisuuksiin (Gamma ym. 1995, s. 334-335). ECS:ssä samankaltaisesti entiteeteille valikoituu niiden aspekteihin perustuen sopivat järjestelmät, joille entiteetit tarjoavat toimintaan kuuluvat komponentit.

Gamma ym. (1995, s. 336) painottavat, että toisin kuin operaatioita ilmentävissä vierailijoissa, iteroitavien elementtien luokkiin tehtävien muutosten tulisi olla sovelluskohteessa vähäisiä. Heidän mukaansa uusien operoitavien elementtien lisääminen tai niiden rajapintojen muutos vaatisi luultavimmin kunkin vierailijan päivittämistä tähän uuteen rajapintaan. Myös Visser (2001) sanoo luokan lisäämisen operoitavien elementtien hierarkiaan johtavan kaikkien vierailijoiden päivittämisen tarpeeseen. ECS:ssä tämä ongelma ei vaikuttaisi toistuvan, sillä hierarkkisen rakenteen vahvan hajottamisen lisäksi käsiteltävät elementit, komponentit, ovat olioita huomattavasti yksinkertaisempaa ”raakaa dataa” (ks. kuvio 3 ja luku 2.2), ja monimutkaisemmat komponentit voivat mahdollisesti olla rajapinnoiltaan yleisempiä (esim. matemaattiset vektorit tai matriisit).

3.3 Suunnittelumallien kaltaisuus

ECS:n yhtäläisyydet sille läheisiksi kuvattuihin suunnittelumalleihin näkyvät tarkastelun perusteella vahvimmin tarpeessa helpottaa monenlaisten ominaisuuksien hallinnointia joukoissa useita ohjelman yksiköitä, entiteettejä tai olioita. Helpotukset perustuvat pitkälti tällaisten yksiköiden erillisiksi osiksi purkamiseen: Komponentti-mallissa (luku 3.2.1) paketoidaan yleistyviä ominaisuuksia sopivasti kytkettäviksi, Strategia-mallissa (luku 3.2.2) yleistyvät algoritmit toimivat ulkopuolelta saamansa syötteen perusteella, Tarkkailija-mallin (luku 3.2.3) avulla mahdollistuu tilamuutoksen ja datan välitys usealle sitä tarvitsevalle ja Vierailija-mallilla (luku 3.2.4) voidaan suorittaa monia samankaltaisia toimintoja erilaisten alkioiden kokoelmassa.

Kuvattujen suunnittelumallien erot ECS:ään koostuvat oliosuuntautuneesta viestinnästä ja datan sijainnista sekä saatavuudesta. Komponentti-mallissa kytkettyjen komponenttien päivitys tapahtuu niiden metodeissa yksitellen. ECS mahdollistaisi päivitysten tapahtumisen järjestelmien tasolla suoraan raakaa dataa läpikäyden. Strategia-mallin mahdollistama algoritmien kytkeminen halutussa tilanteessa olisi ECS:ssä luultavasti monimutkaisempaa, sillä järjestelmät valitsevat entiteetit sopivien aspektien mukaan, jotka taasen vaihtuvat komponenttien myötä; suoritettavan algoritmin vaihtaminen voisi vaatia sille välttämätöntä erillistä ECS-komponenttia. Algoritmin 3.1 tapainen Tarkkailija-mallin käyttäminen tarkoittaisi entiteettien sisällyttämistä osaksi komponenttia, mikä ei tutkielmassa käsitellyssä kirjallisuudessa ole tullut ilmi. Entiteetin ja komponentin sekoittaminen tällä tavalla voisi kenties vaatia erityistä huomiointia entiteettien ja komponenttien mallintamisessa tai olla ECS:n käytölle perustavanlaatuisesti luonnotonta. Vierailija-mallissa keskeisesti muodostettava hierarkia, joka mahdollistaa kokoelman rajapintojen yhteneväisyyden toimintoja varten, on ECS:n hierarkkisuuden hävittämiseen nähden ristiriidassa; ECS:ssä rajapinnat tunnistetaan aspekteista, jotka määrittyvät raa'asta yleiskäyttöisestä datasta.

ECS ei sille läheisiksi kuvattujen suunnittelumallien tarkastelun myötä vaikuta olevan aivan perinteinen oliosuuntautunut suunnittelumalli. ECS:ää voidaankin nähdä kutsuttavan myös arkkitehtuurimalliksi (engl. architectural pattern) (Hodson ja Millar 2018) tai paradigmaksi (Berge ym. 2014). Martin (2007a) nimittää sitä suoraviivaisesti järjestelmäksi. ECS:stä on kuitenkin tarkastelun perusteella nähtävissä muissa suunnittelumalleissa toistuvia perus-

periaatteita koostamisen käytöstä perinnän sijaan. ECS:n nouseminen ajan saatossa ohjelmistoteknisestä ammattimaisesta kontekstista (Bilas 2002; Martin 2007a; West 2007) vastaa tunnettujen suunnittelumallien syntytapoja (Gamma ym. 1995, s. 1-2), mikä myös on oletettavasti vahvistanut ECS:n uskottavuutta ohjelmistokehityksessä.

4 Entiteetti-komponentti-järjestelmä olio-ohjelmoinnissa

Oliosuuntautuneesta perinnästä syntyvien läheisten kytkentöjen tunnistetusta ongelmallisuudesta ja koostamisen myönteisistä vaikutuksista huolimatta, on Fischbachin, Wiebuschin ja Latoschikin (2017) mukaan perintä hyvin yleinen ja jopa pääasiallisin tekniikka oliopohjaisessa ohjelmistoarkkitehtuurissa. ECS:n kaltaisten suunnittelumallien korostamien periaatteiden yleistymiseksi on siten aiheellista tutkia muita mahdollisuuksia ohjelmointiparadigmojen tasolla.

Wiebuschin ja Latoschikin (2015) mukaan ECS:n ollessa oliosuuntautuneille suunnittelumalleille läheinen, sen toteutuksessa käytetään usein oliosuuntautunutta ohjelmointikieltä. Heidän mukaansa olio-ohjelmoinnin tuoma kapselointi on myös ECS:n kontekstissa hyödyllistä, mutta entiteettien ja komponenttien esittäminen olioina attribuutteineen johtaa helposti jäykkiin hierarkioihin. Koska ECS:n perustaviin periaatteisiin kuuluu datan ja logiikan erottelu tavoilla, jotka ovat huomattavimmin ristiriidassa juuri oliosuuntautuneeseen kapselointiin kuuluvan attribuuttien ja metodien rakenteellisesti yhteen liittämisen kanssa, ei olio-ohjelmointi tunnu ECS:lle luontaiselta. Joissakin tapaustutkimuksissa käytettiin ECS:n toteutuksessa Scala-ohjelmointikieltä, joka nähtiin hyödyllisenä tyyppiparametreista johtuvan käännönaikaisen tyypityksen (Wiebusch ja Latoschik 2015) ja suppean syntaksin (Fischbach, Wiebusch ja Latoschik 2017) myötä. Scala on olio- ja funktionaalista paradigmaa yhdistelevä ohjelmointikieli (Odersky ja Rompf 2014) ja tämän yhdistelyn esiintyminen ECS:n toteuttamisessa edelleen viittaa ECS:n ja vahvan oliokeskeisyyden yhteensopimattomuuteen.

Kirjoituksessaan Martin (2007b) tulkitsee entiteettijärjestelmien (engl. entity systems) olevan osajoukko komponenttisuuntautuneesta ohjelmoinnista (engl. component oriented programming), joka Wanging ja Qianin (2005) mukaan painottaa luokkien ja olioiden tyyppien sijaan sopivia rajapintoja ja ohjelman kokoonpanoa. Martin (2007b) vertaa ECS:n käyttöä myös relaatiotietokannan kanssa ohjelmoimiseksi. Tällaista näkemystä tukevat myös muusta kirjallisuudesta havaittava tapa kuvantaa ECS:n entiteettejä taulukkona ohjelmassa esiintyvistä komponenteista (Raffaillac ja Huot 2019; West 2007; Fontana ym. 2017). Vertaus relaatiotietokantaan on helposti käsitettävissä: entiteetit ovat taulun rivejä yksilöiviä avaimia ja muissa sarakkeissa luetellaan avaimeen liitettävää yksinkertaista dataa, komponentteja.

5 Yhteenveto

Tässä kandidaatintutkielmassa kuvattiin entiteetti-komponentti-järjestelmänä -tunnetun suunnittelumallin periaatteita. Eri näkemyksiin perustuen avattiin ECS:n määritelmää ja tuotiin esille hyötyjä ja haittoja. Tarkasteltiin myös ECS:n yhteyksiä ja eroja muutamaan sille läheiseen suunnittelumalliin sekä pohdittiin käyttöä oliosuuntautuneessa ohjelmoinnissa.

Entiteetti-komponentti-järjestelmän määrittely on huomattu hajanaiseksi ja eri määritelmistä voidaan havaita vaihtelevia nimityksiä samankaltaisille periaatteille. Kirjallisuudessa esiintyy myös entiteetti-komponentti (engl. entity-component) -nimitystä ECS:n kaltaisesta mallista. Nämä ongelmakohdat vaikeuttavat ECS:n tutkimista pelkästään kirjallisuuteen perustuen. Tapaustutkimus pääasiallisesti pelialalta mukailtuihin kuvauksiin ECS:stä (kuten Raffaillac ja Huot (2019), Wiebusch ja Latoschik (2015), Lange, Weller ja Zachmann (2016) ja Fontana ym. (2017)) tai olemassaolevien toteutusten, jollaisia Raffaillac ja Huot (2019) tuovat esille, dokumentaatioon perustuen, olisi siten suoraviivaisin puhtaan ECS:n tarkastelutapa.

Tässä tutkielmassa ECS:n toteutukseen liittyneet lähdeviitteettömät pohdinnat perustuivat osin, tutkielmaan sisältymättömien kokemusten pohjalta, kirjoittajalle muodostuneisiin mielikuviin entiteettien, komponenttien ja järjestelmien esittämisestä, näkyvyydestä ja yhteistoiminnasta (erityisesti Algoritmi 2.1 ja Algoritmi 3.1). Jatkotutkimuksena ECS voitaisiin toteuttaa sekä näiden esiin tulleiden pohdintojen, että tutkielmassa esiteltyjen käyttökohteiden tarkentamiseksi. Erityisesti ECS:n käyttöön liitetyt suorituskykyparannukset datasuuntautuneeseen suunnitteluun perustuen, vaikuttavat huomionarvoisilta. Jatkotutkimuksessa voitaisiin pyrkiä tarkastelemaan suorityskyvyllistä tehokkuutta äärimmäisissä RIS:ien kaltaisissa interaktiivisten peliobjektien simulaatiotilanteissa ECS:ää hyödyntäen.

Lähteet

Acton, Mike, ja Insomniac Games. 2014. “Data-oriented design and c++”. *Luento. CppCon.*

Berge, C Schulte zu, Artur Grunau, Hossain Mahmud ja Nassir Navab. 2014. *CAMPVis—a game engine-inspired research framework for medical imaging and visualization.* Tekninen raportti. Tech. rep. Technische Universität München, 2014.

Bilas, Scott. 2002. “A data-driven game object system”. Teoksessa *Game Developers Conference Proceedings.*

Elvezio, Carmine, Mengu Sukan ja Steven Feiner. 2018. “Mercury: A messaging framework for modular ui components”. Teoksessa *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems,* 1–12.

Fischbach, Martin, Dennis Wiebusch ja Marc Erich Latoschik. 2017. “Semantic entity-component state management techniques to enhance software quality for multimodal VR-systems”. *IEEE transactions on visualization and computer graphics* 23 (4): 1342–1351.

Fontana, Tiago, Renan Netto, Vinicius Livramento, Chrystian Guth, Sheiny Almeida, Laércio Pilla ja José Luís Güntzel. 2017. “How Game Engines Can Inspire EDA Tools Development: A use case for an open-source physical design library”. Teoksessa *Proceedings of the 2017 ACM on International Symposium on Physical Design,* 25–31.

Gamma, Erich, Richard Helm, Ralph Johnson ja John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software.* Pearson Education India.

Garcia, Franco Eusébio, ja Vânia Paula de Almeida Neris. 2014. “A data-driven entity-component approach to develop universally accessible games”. Teoksessa *International Conference on Universal Access in Human-Computer Interaction,* 537–548. Springer.

Hodson, Douglas D, ja Jeremy Millar. 2018. “Application of ECS game patterns in military simulators”. Teoksessa *Proceedings of the International Conference on Scientific Computing (CSC),* 14–17. The Steering Committee of The World Congress in Computer Science, Computer ...

Implementing Component-Entity-Systems. 2013. https://www.gamedev.net/tutorials/_/technical/game-programming/implementing-component-entity-systems-r3382/, viitattu 05.05.2020.

Lange, Patrick, Rene Weller ja Gabriel Zachmann. 2016. "Wait-free hash maps in the entity-component-system pattern for realtime interactive systems". Teoksessa *2016 IEEE 9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS),* 1–8. IEEE.

Martin, Adam. 2007a. *Entity Systems are the future of MMOG development - Part 1.* http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/, viitattu 20.03.2020.

———. 2007b. *Entity Systems are the future of MMOG development - Part 2.* http://t-machine.org/index.php/2007/11/11/entity-systems-are-the-future-of-mmog-development-part-2/, viitattu 10.03.2020.

Meijer, Lucas. 2019. *On DOTS: Entity Component System.* https://blogs.unity3d.com/2019/03/08/on-dots-entity-component-system/, viitattu 4.2.2020.

Nystrom, Robert. 2014. *Component.* http://gameprogrammingpatterns.com/component.html, viitattu 09.04.2020.

Odersky, Martin, ja Tiark Rompf. 2014. "Unifying functional and object-oriented programming with Scala". *Communications of the ACM* 57 (4): 76–86.

Raffaillac, Thibault, ja Stéphane Huot. 2019. "Polyphony: Programming Interfaces and Interactions with the Entity-Component-System Model". *Proceedings of the ACM on Human-Computer Interaction* 3 (EICS): 1–22.

Wang, Andy Ju An, ja Kai Qian. 2005. *Component-oriented programming.* John Wiley & Sons.

West, Mick. 2007. *Refactoring Game Entities with Components.* http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/, viitattu 20.03.2020.

Wiebusch, Dennis, ja Marc Erich Latoschik. 2015. “Decoupling the entity-component-system pattern using semantic traits for reusable realtime interactive systems”. Teoksessa *2015 IEEE 8th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS),* 25–32. IEEE.

Visser, Joost. 2001. “Visitor combination and traversal control”. Teoksessa *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications,* 270–282.

Liitteet

A Lyhentämätön ECS-esimerkki

Algoritmissa 5.1 esitetään Python 3 -ohjelmointikielellä toteutettu esimerkki ECS:n käyttämisestä sovelluksessa.

Algoritmi 5.1. Lyhentämätön ECS-esimerkki (mukaellen *Implementing Component-Entity-Systems* 2013)

```
def lue_syote():
    syote = input("Syötä␣haluamasi␣suunta␣['vas','oik']>␣")
    if syote == "vas":
        return (−1,0)
    elif syote == "oik":
        return (1,0)
    return (0,0)

# Entiteetin rekisteröinti ja simulaatioon lisääminen
def lisaa_entiteetti(entiteetin_komponentit):
    for aspekti in entiteetin_komponentit.keys():
        komponentit[aspekti][lisaa_entiteetti.e_id] = entiteetin_komponentit[aspekti]
    # Entiteetin aspekteina käytetään liitettyjen komponenttien nimitysten joukkoa
    entiteetit[lisaa_entiteetti.e_id] = set(entiteetin_komponentit.keys())
    lisaa_entiteetti.e_id += 1

# Tehtaat entiteettien alustukseen:
luo_pilvi = lambda: { "sijainti": (3, 12), "suunta": (1, 0) }
luo_puu = lambda v: { "elama": 120, "sijainti": v }
luo_pelaaja = lambda: { "elama": 90, "suunta": (0,0), "ohjain": lue_syote, "sijainti": (1,0) }
```

```
# Käytettävät järjestelmät:
# Entiteettien kulkusuuntaa käsittelevä järjestelmä
def ohjain():
    ohjain_aspekti = set(["suunta", "ohjain"])
    for entiteetti, aspekti in entiteetit.items():
        # Aspektin/kuvailutermin käyttötarkoitus (ks. luvut (2.2) ja (2.3.1))
        if ohjain_aspekti.issubset(aspekti):
            # Tässä järjestelmässä hyödynnetään komponentiksi asetettua funktiota
            uusi_suunta = komponentit["ohjain"][entiteetti]()
            komponentit["suunta"][entiteetti] = uusi_suunta

def liike():
    liike_aspekti = set(["suunta", "sijainti"])
    for entiteetti, aspekti in entiteetit.items():
        if liike_aspekti.issubset(aspekti):
            nx,ny = komponentit["suunta"][entiteetti]
            sx,sy = komponentit["sijainti"][entiteetti]
            komponentit["sijainti"][entiteetti] = (sx+nx, sy+ny)

def elama():
    elama_aspekti = set(["elama"])
    for entiteetti, aspekti in entiteetit.items():
        if elama_aspekti.issubset(aspekti):
            komponentit["elama"][entiteetti] −= 30

def tulosta_tilanne():
    for entiteetti, e_aspekti in entiteetit.items():
        s = ""
        print("Entiteetti", entiteetti)
        for k_aspekti in komponentit.keys():
            s += "␣␣{:9}::␣{}\n". format(k_aspekti,
```

```
                                        str(komponentit[k_aspekti][entiteetti])
                                        if k_aspekti in e_aspekti
                                        else "")
        print(s)

# Ohjelman alku:
# Käytettävät komponentit ja siten aspektit määritellään
# ja entiteettejä kokoava tietorakenne sekä juokseva tunniste alustetaan.
komponentit = { "elama": dict(), "suunta": dict(), "ohjain": dict(), "sijainti": dict() }
entiteetit = dict()
lisaa_entiteetti.e_id = 0

# Entiteettejä lisätään maailmaan tehdasfunktioihin ennalta määritetyillä
# oletusarvoisilla, tai parametrien mukaisilla, komponenteilla (ks. kaavain luvussa (2.3.1)).
# Tässä esimerkissä tehtaat palauttavat hakurakenteen halutuista komponenteista.
lisaa_entiteetti( luo_puu((2,1)) )
# Pelaajan tunniste talletetaan tässä ohjelman kulun kontrolloimiseksi
pelaaja_entiteetti = lisaa_entiteetti.e_id
lisaa_entiteetti( luo_pelaaja() )

# Käynnistetään simulaation pääsilmukka
while(komponentit["elama"][pelaaja_entiteetti] > 0):
    # Entiteettejä voidaan lisätä osaksi suorituksessa olevaa simulaatiota
    lisaa_entiteetti( luo_pilvi() )
    # Järjestelmät ajetaan simulaatiolle sopivassa järjestyksessä
    tulosta_tilanne()
    ohjain()
    liike()
    elama()
tulosta_tilanne()
```