DCI-arkkitehtuuri – askel kohti ylläpidettävää olio-ohjelmointia

Miikka Kähkönen

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Pro gradu -tutkielma
Ohjaaja: Timo Poranen
joulukuu 2015

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Miikka Kähkönen: DCI-arkkitehtuuri – askel kohti ylläpidettävää olio-ohjelmointia
Pro gradu -tutkielma, 57 sivua, 14 liitesivua
joulukuu 2015

Ohjelmistokehityksessä järjestelmien ylläpidettävyys on sen yksi tärkeimmistä ominaisuuksista. Tutkielmassa perehdytään DCI-arkkitehtuuriin, joka on uudenlainen olio-ohjelmointitapa. Arkkitehtuurin on tarkoitus parantaa olioperustaisten järjestelmien ylläpidettävyyttä perinteiseen olio-ohjelmointiin verrattuna. DCI-arkkitehtuuri painottaa ohjelmoinnissa lean-ajatteluun ja ketterään kehitykseen kooditasolla.

Tutkielmassa tarkastellaan DCI-arkkitehtuuria ja arvioidaan sitä arkkitehtuurin mukaisesti luodun laajan peliesimerkin avulla. Pelin arkkitehtuuria arvioidaan ketterän kehityksen ja lean-ajattelun näkökulmista. Ketterä kehitys painottaa nopeuteen ohjelmistokehityksessä ja sen suhteen peliä arvioidaan erilaisilla kompleksisuusmittareilla sekä konkreettisen muutoksen toteuttamisella. Lean-ajattelun näkökulmasta arkkitehtuurin arviointi perustuu siitä johtuvaan ylimääräisen koodiin, toteutetun koodin arvoon ja arkkitehtuurin jäykkyyteen.

Tutkielman tulosten perusteella DCI-arkkitehtuuri tuki lean-ajattelua ja ketterää kehitystä. Tutkielmassa käytettyjen olio-ohjelmoinnin kompleksisuusmittareiden tulokset osoittautuivat kuitenkin kyseenalaisiksi, sillä mittareita ei ole suunniteltu DCI-arkkitehtuuria silmällä pitäen ja arkkitehtuuri ei täytä niiden määritelmiä kokonaan. DCI-arkkitehtuurin käyttö ei aiheuttanut ongelmia pelin toteutuksessa, joten sen tutkimista tulisi jatkaa tarkemmin reaalimaailman pienten ja keskisuurten ohjelmistoprojektien tapaustutkimuksilla.

Avainsanat ja -sanonnat: olio-ohjelmointi, DCI-arkkitehtuuri, lean-ajattelu, ketterä ohjelmistokehitys, ylläpidettävyys

Sisällys

1 Johdanto

Olio-ohjelmointi on tällä hetkellä ehkä suosituin ohjelmointiparadigma. Olio-ohjelmoinnin alkuperäinen tarkoitus oli tuoda ohjelmointi lähemmäs ihmisen ajatusmallia ja helpottaa siten ohjelmistokehitystä. Olio-ohjelmoinnin piti luoda uudelleenkäytettävää ja modulaarista koodia, mutta kriitikoiden mukaan sen käyttö on luonut koodista monimutkaista ja epäselvää [Cardelli, 1996; Seibel & Armstrong, 2009]. Nykyisessä ohjelmistokehityksessä ohjelmistoja pitää luoda yhä nopeammin ja laajemmin, mihin monimutkaiset ohjelmat eivät sovi. Ketterä ohjelmistokehitys on saanut jatkuvasti enemmän suosiota ohjelmistokehityksessä, mutta sen painottama nopeus on vähentänyt suunnittelua. Suunnittelun puute nopeasti tuotetuissa olioperustaisissa ohjelmissa on tehnyt niistä aikaisempaa monimutkaisempia. [Coplien & Bjørnvig, 2010]

Olio-ohjelmointi ei välttämättä ole monimutkaisuuden syy. Reenskaug on kehittänyt DCI-arkkitehtuurin, jonka tarkoituksena on saavuttaa ymmärrettävää olioperustaista koodia. [Reenskaug, 2009] DCI-arkkitehtuuri eroaa nykyisestä luokka-pohjaisesta olio-ohjelmoinnista merkittävästi niin ohjelman koodin kuin ohjelmistokehityksenkin näkökulmasta. [Reenskaug & Coplien, 2009] Ohjelmistokehityksen näkökulmasta DCI-arkkitehtuuri yhdistää ketterän kehityksen ja lean-ajattelun siten, että ketteryydestä tulevaa epävakautta tasapainotetaan leanin järkevällä suunnittelulla. [Coplien & Bjørnvig, 2010] DCI-arkkitehtuurissa olio-ohjelmoinnin konseptia laajennetaan rooleilla ja toimintoa sisältävillä rajapinnoilla, joiden avulla järjestelmää voidaan aikaisempaa paremmin kuvata ihmisen ajatusmallin mukaisesti. [Reenskaug & Coplien, 2009] Tutkielmassa käydään läpi DCI-arkkitehtuuri ja arvioidaan sillä tehtyä peliä ketterän kehityksen sekä lean-ajattelun näkökulmista.

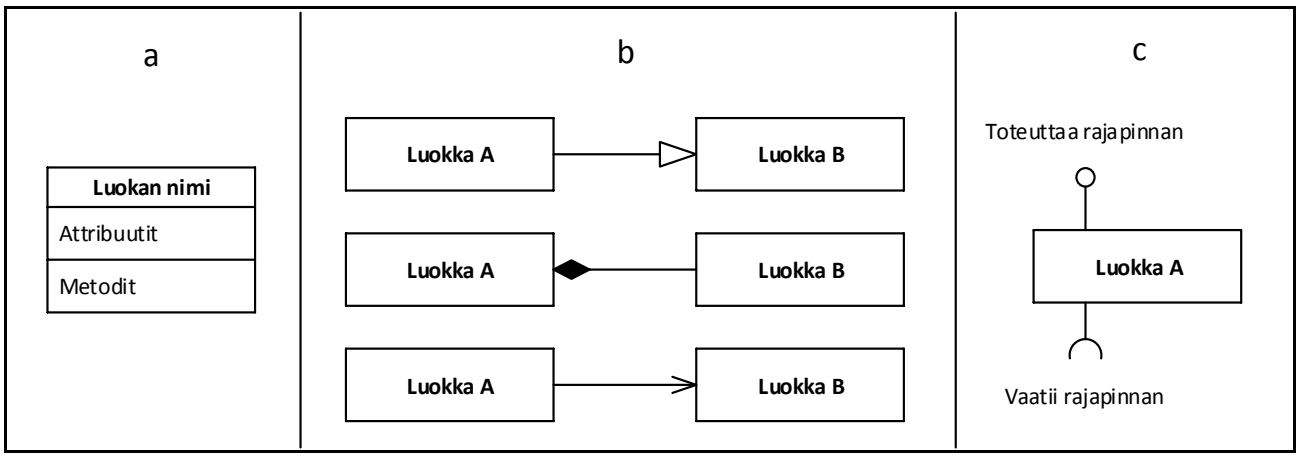
Tutkielman varsinainen sisältö alkaa luvussa 2, jossa käydään läpi olio-ohjelmointia ympäröiviä konsepteja. Luvun 2 aluksi pohjustetaan UML, joka on olio-ohjelmoinnissa ja yleisesti ohjelmistokehityksessä käytetty merkintäkieli. Ohjelmistojen laatuun liittyviä termejä ja oleellista pohjatietoa käsitellään luvun 2 toisessa osiossa, mitä seuraa olioperustaisuuden määrittäminen. Olioperustaisuuden jälkeen käsitellään siihen liittyviä metriikoita ja lopuksi ketterää ohjelmistokehitystä ja lean-ajattelua. Luvussa 3 määritetään DCI-arkkitehtuuri ja sen rakenne. Luvussa 4 esitellään DCI-arkkitehtuurilla tehty peli, jonka ominaisuuksia arvioidaan luvussa 5. Luvussa 6 on yhteenveto tutkielmasta, sen tuloksista ja aiheen jatkotutkimuksesta.

2 Ohjelmistokehitys

2.1 UML

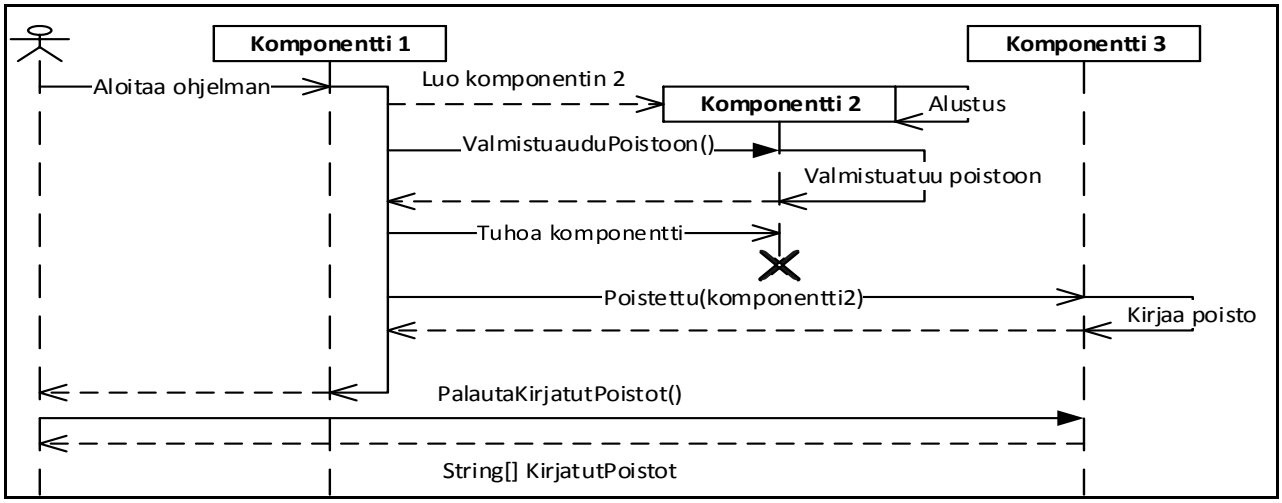
UML (Unified Model Language) on yleisesti ohjelmistokehityksessä käytetty graafinen mallinnuskieli. UMLn avulla voidaan esittää graafisesti järjestelmän erilaisia rakenteita, vuorovaikutuksia ja käyttäytymisiä. Tutkielmassa on käytetty luokkakaavioita rakenteen ja sekvenssikaavioita järjestelmän sisäisen vuorovaikutuksen esittämiseen. UMLn uusin versio on 2.5 vuodelta 2015. [UML, 2015]

Luokkakaaviolla esitetään luokkien rakennetta järjestelmässä. Tutkielmassa käsitellään luokkakonsepti kohdassa 2.3. Luokkakaaviossa luokan sisällä on kyseisen luokan attribuutit ja metodit (kuva 1a). Tutkielmassa luokkia yhdistetään toisiinsa kolmella tavalla: perinnällä, liittämisellä ja koostella (kuva 1b). Luokkakaavioon voidaan merkitä myös sen toteuttamat tai vaatimat rajapinnat (kuva 1c). Tarkempi dokumentaatio löytyy UML-dokumentaatiosta. [UML, 2015]



Kuva 1: UML-luokkakaavio.

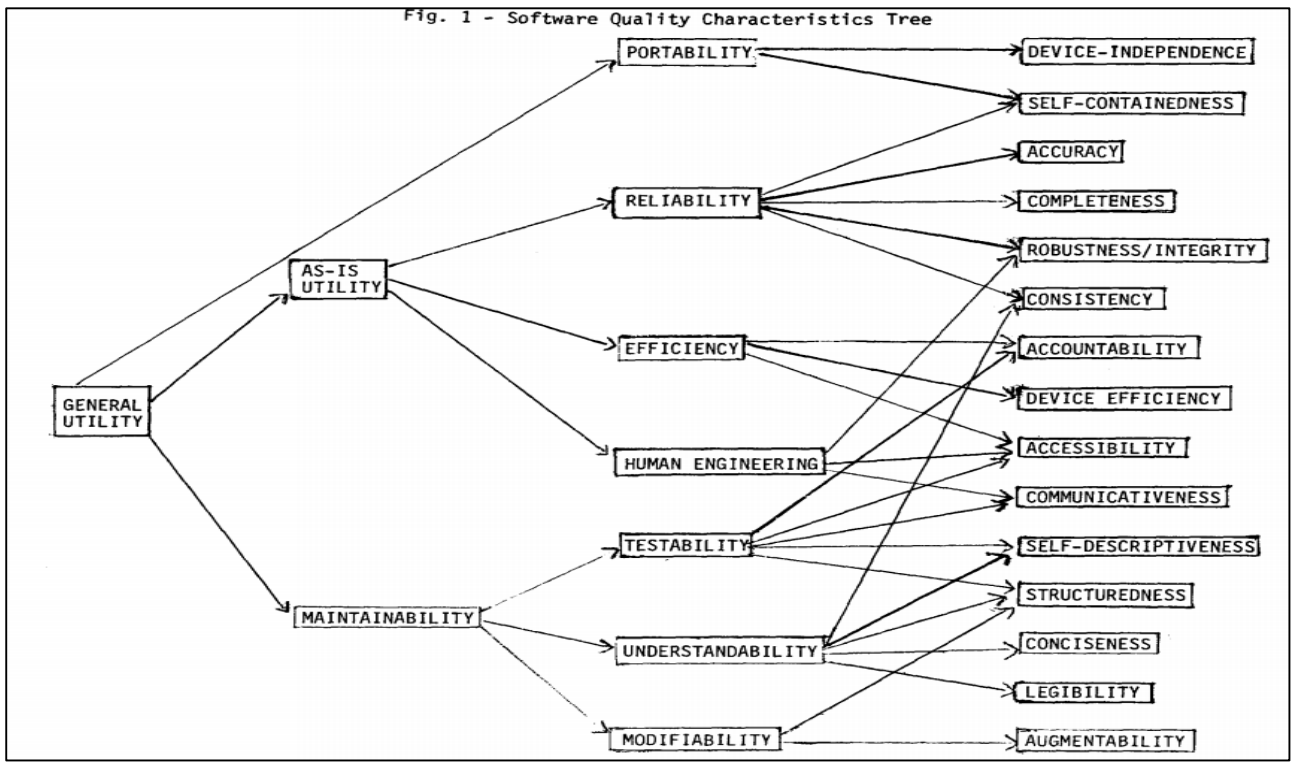
Sekvenssikaaviolla esitetään tutkielmassa järjestelmän sisäistä vuorovaikutusta, eli käytännössä järjestelmän sisäisten komponenttien välistä viestintää. Sekvenssikaaviossa näkyy kaikki komponentit, jotka ovat oleellisia kyseisessä viestinnässä (kuva 2). Komponentit erottuvat siitä alaspäin lähtevällä pystyviivalla. Komponenttien välillä on nuolia, jotka esittävät viestejä niiden välillä. Nuolen nimi kertoo, mistä viestistä on kyse. Nuolen suunta kertoo viestin suunnasta (kuva 2). Sekvenssikaaviota luetaan vasemmasta ylänurkasta alas seuraten vastaan tulevia nuolia. Tarkempi dokumentaatio löytyy UML-dokumentaatiosta. [UML, 2015]



Kuva 2: UML-sekvenssikaavio.

2.2 Ohjelmiston laatuominaisuudet

Ohjelmiston laadulla (software quality) tarkoitetaan ohjelmiston sopivuutta sen tarkoitukseen, ja ohjelmiston laatuominaisuudella (software quality attribute) tarkoitetaan ohjelmiston laatua jostain näkökulmasta. [Sommerville, 2011] Laatuominaisuudet ovat pitkälti subjektiivisia, ja niille on myös useita määritelmiä. Tutkielmassa käytetään hyvin tunnettuja Boehmin ja muiden [1976] määritelmiä (kuva 3). Ohjelmisto on mahdoton tehdä kaikkia laatuominaisuuksia silmällä pitäen. [Sommerville, 2011] Esimerkiksi ohjelmiston turvallisuuteen panostaminen haittaa monesti sen ymmärrettävyyttä tai tehokkuutta.



Kuva 3: Boehmin ja muiden tekemä ohjelmiston laatuominaisuuksien puu [Boehm et al., 1976].

Ohjelmiston laatuominaisuudet muodostavat yhdessä suurempia kokonaisuuksia. Näitä järjestelmäominaisuuksia ovat muun muassa ylläpidettävyys (maintainability). Tutkielmassa keskitytään ohjelmiston ylläpidettävyyden tarkasteluun, jolla tarkoitetaan yleisesti ohjelmiston valmiutta ottaa vastaan muutoksia, joita voi tapahtua esimerkiksi ohjelmiston määrittelyssä tai toteutuksessa. Ylläpidettävyys kattaa useita ohjelmiston laatuominaisuuksia riippuen määritelmästä, mutta Boehmin ja muiden määritelmässä se kattaa ymmärrettävyyden (understandability), testattavuuden (testability) ja muokattavuuden (modifiability). Tutkielmassa keskitytään ylläpidettävyyden tarkasteluun ymmärrettävyyden ja muokattavuuden näkökulmista. [Boehm et al., 1976]

Muokattavuudella tarkoitetaan sitä, kuinka helppoa ohjelmistoon on toteuttaa jokin tunnettu muutos. Muokattavuuden alakohtana on laajennettavuus (augmentability), joka kertoo, kuinka helppoa järjestelmän nykyistä toimintaa on laajentaa. Ymmärrettävyydellä tarkoitetaan, kuinka selkeää ohjelmiston toiminta on sitä tutkivalle henkilölle. Ymmärrettävyyden alakohdat ovat luettavuus (legibility), järjestelmällisyys (structuredness), tietoisuus (conciseness), yhtenäisyys (consistency) ja kuvaavuus (self-descriptiveness) (taulukko 1). [Boehm et al., 1976]

Luettavuus	Kertoo kuinka selkeää järjestelmän toiminta on koodista luettuna.
Järjestelmällisyys	Kertoo kuinka selkeää ohjelmiston jaottelu on.
Tietoisuus	Kertoo kuinka paljon ylimääräistä tietoa järjestelmässä on sen toiminnan suhteen.
Kuvaavuus	Kertoo kuinka paljon järjestelmä kertoo sen omista vaatimuksista, oletuksista, toiminnasta jne.
Yhtenäisyys	Sisäinen yhtenäisyys kertoo kuinka paljon järjestelmässä on käytetty yhdenmukaista notaatiota jne. Ulkoinen yhtenäisyys tarkoittaa kuinka selkeästi jokin osio liittyy järjestelmän vaatimuksiin.

Taulukko 1: Ymmärrettävyyden alakohdat [Boehm et al., 1976].

2.3 Olio-ohjelmointi ja olioperustaisuus

Olio-ohjelmointi on yksi ohjelmointiparadigma, jossa järjestelmä rakennetaan keskenään kommunikoivista olioista. Termillä alun perin tarkoitettiin vain järjestelmän sisällä tapahtuvaa viestintää. [Kay, 2003] Olioperustaisuus mielletään korkeamman tason prosessiksi, jossa järjestelmää ajatellaan olioilla koko sen kehityksen ajan. Olioperustaisuudessa olio-ohjelmointi viittaa konkreettiseen ohjelmointiin, jossa käytetään olioperusteisuuden paradigman ydinominaisuuksia. Paradigmaa on kehitetty jo 1960 luvulta lähtien, ja vuonna 1967 kehitetty Simula67 oli ensimmäinen olio-ohjelmointia mahdollistava varsinainen ohjelmointikieli. Olioparadigman kehitys otti seuraavan

askeleen vuonna 1970 Xerox Parcin Smalltalkilla, joka oli ensimmäinen ”puhdas” olio-ohjelmointikieli. Olio-ohjelmointi ei levinnyt kuitenkaan merkittävästi ohjelmistotuotantoon, jossa oli totuttu proseduurilliseen ohjelmointiin ja uuden olioparadigman leviäminen koki muutosvastarintaa. Olio-ohjelmointi löi läpi kuitenkin 1980-luvun puolivälissä C-kielen luokkalisäyksellä ja siitä kehitetyllä C++:lla. Nykyään olioparadigma on yksi tärkeimmistä ohjelmointi-paradigmoista. [Koskimies, 2000]

Järjestelmän kuvaaminen olioilla on sanottu olevan ihmiselle luontaisempi tapa jäsentää järjestelmää verrattuna muihin paradigmoihin [Coplien & Bjørnvig, 2010]. Olio-ohjelmoinnista onkin tullut yksi yleisimmistä ohjelmointitavoista ja sen toteutus on jakautunut kahteen eri toteutustapaan: luokkaperustaiseen ja prototyyppiperustaiseen ohjelmointiin. Luokkaperustaisuus kuvaa oliot staattisten luokkien edustajiksi ohjelman ajonaikana. Luokat määritellään ennen ajoa, ja siitä ajonaikana luodut oliot sisältävät luokan ominaisuudet. Prototyyppiperustaisuudessa olioita kloonataan ajon aikana siten, että kloonattu olio jää omaksi oliokseen, ja kloonauksessa tuotettu olio perii alkuperäisen olion ominaisuudet. Olioperustaisuuden varsinaisilla olioilla on useita määritelmiä, mutta yleisesti ajatellaan oliolla olevan seuraavat ominaisuudet:

1. Olio voi suorittaa sille ominaisia toimintoja. (Operaatio)
2. Olio pitää sisällään sille ominaista tietoa. Olion kaikkien tietojen yhdistelmää kutsutaan sen tilaksi. (Attribuutti)
3. Oliolla on yksilöivä tunniste, viite. Tämä tunniste on aina yksikäsitteinen olioiden suhteen.
4. Olion tiedot ja operaatiot ovat suojattu siten, että vain ulkopuolisille tahoille tarkoitetut operaatiot ja tiedot ovat ulkoapäin käytettävissä.

Riippuen ohjelmointikielestä näillä ominaisuuksilla on eri nimityksiä kuten operaation kohdalla metodi. [Koskimies, 2000]

Riippumatta luokka- tai prototyyppiperustaisesta kehityksestä, molemmissa tapauksissa päädytään ehkä olioperustaisuuden merkittävimpään ongelmaan, eli olioiden tunnistamiseen järjestelmässä. Olion määritelmä on sovellettavissa monella tapaa, eikä siten ole selvää, mitkä asiat järjestelmässä ovat oliota. Olioiden tunnistamiseen on olemassa monia sääntö- ja ohjeoppaita, mutta olioiden tunnistaminen on lopuksi aina järjestelmäkohtainen. Osasyy, miksi olioiden tunnistaminen on vaikeaa ja järjestelmäkohtaista, on eri laatuominaisuuksien esiin tuominen järjestelmässä. Riippuen siitä, mitkä järjestelmän osat kuvataan olioina, vaikutetaan näihin järjestelmäominaisuuksiin, kuten ylläpidettävyyteen, uudelleenkäyttöön ja tehokkuuteen. Järjestelmän kuvaamisessa on tehtävä päätös, mitä ominaisuuksia halutaan korostaa. [Koskimies, 2000] Olioperustainen ohjelmointi on monimutkaista ja vielä monimutkaisempaa on sen tekeminen uudelleenkäytettäväksi. [Gamma et al., 1994]

Olio-ohjelmoinnin sääntö- ja ohjeoppaiden yksi merkittävin ohjeiden aihealue on suunnittelumallit. Suunnittelumallit ovat yleisen tason ratkaisuja toistuviin olio-

ohjelmoinnin ongelmiin. [Gamma et al., 1994] Pääosin ne ovat yleistyksiä hyvistä ratkaisuista, jotka toimivat ohjelmoijan ja nykyisin myös järjestelmäsuunnittelijan työkaluina ohjelmistokehityksessä.

Gamma ja muut [1994] ovat kirjoittaneet kirjan, joka on kokoelma olio-ohjelmoinnin suunnittelumalleista. Kirja on olio-ohjelmoinnin yksi merkittävimmistä teoksista. Kirjassa on useita kymmeniä suunnittelumalleja, joista jokaiselle löytyy seuraavat tiedot: ratkaisu, nimi, ongelma ja seuraamukset. Ongelmalla tarkoitetaan sitä ongelmaa, johon ratkaisua tulisi soveltaa ja seuraamuksella mallin käytön seuraamuksia, esimerkiksi muistivaatimusta. Suunnittelumallien hyödyntäminen olioperustaisessa kehityksessä auttaa myös jäsentämään ja tunnistamaan järjestelmästä oliota sekä muita toimijoita. Mikäli kehittäjä pyrkii ratkaisemaan ongelmaa, joka on jo ratkaistu suunnittelumallilla, on yleensä suotavaa hyödyntää tätä mallia. Mallin käyttäminen antaa viitteitä ainakin osittaisista ominaisuuksista, tai eri olioiden olemassaolosta. Taulukot 2 ja 3 kuvaavat esimerkin suunnittelumallin dokumentaation rungosta ja yksinkertaistetun esimerkin fasaadi-mallista (facade).

Suunnittelumallin nimi ja luokka

Tarkoitus
Lyhyt kuvaus suunnittelumallista siten, että siitä käy ilmi mitä malli tekee, mihin ratkaisu perustuu ja mihin tilanteeseen sitä käytetään.

Tunnetaan myös
Millä muilla nimillä malli voidaan tuntea.

Motivaatio
Esimerkkiskenaario ongelmasta, joka on ratkaistu käyttämällä kyseistä mallia. Tarkoituksena helpottaa mallin yleistettyä määritelmää.

Käyttömahdollisuudet
Tilanteet, mihin mallia voidaan hyödyntää ja miten kyseiset tilanteet voidaan huomata.

Rakenne
Visuaalinen esitys mallin luokista.

Osallistujat
Luokat ja/tai oliot, jotka osallistuvat mallin toteutukseen, sekä niiden vastuut mallin toiminnassa.

Yhteistoiminta
Kuvaus siitä, miten osallistujat muodostavat mallin ja sen toiminnan.

Seuraamukset
Seuraamuksista tulisi käydä ilmi seuraavat asiat: miten malli tukee sen tarkoitusta; mitkä ovat mallin käyttöön liittyvät kompromissit ja seuraamukset; mitä osia järjestelmän rakenteesta voi muokata muihin vaikuttamatta.

Toteutus
Mallin toteutukseen liittyvät ongelmat, huomiot ja muut käytännön asiat, jotka liittyvät mallin käyttöön. Malliin liittyvät ohjelmointi-kielellisten huomioiden tulisi löytyä täältä.

Esimerkkikoodi
Koodiesimerkkejä siitä, miten kyseinen malli voidaan toteuttaa käyttäen C++ tai Smalltalkia.

Tunnetut käytöt
Esimerkkitilanteet, missä kyseistä mallia on oikeasti käytetty. Kirjassa on vähintään kaksi käyttötilanteita, jotka ovat esimerkkejä eri toimialueilta.

Liittyvät mallit
Mallin läheisesti liittyvät toiset suunnittelumallit ja niiden erot käsiteltävään malliin. Mallin käyttöön liittyvät muut suunnittelumallit tulee esittää tässä kohdassa.

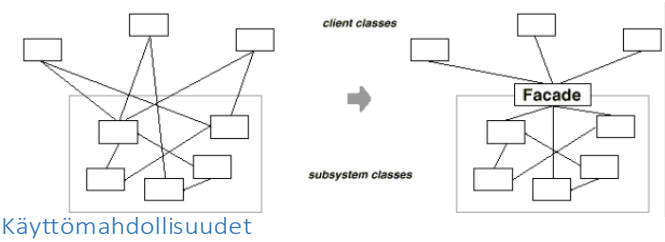
Taulukko 2: Yleinen dokumentaatio-ohje.

Fasaadi (eng. Facade)

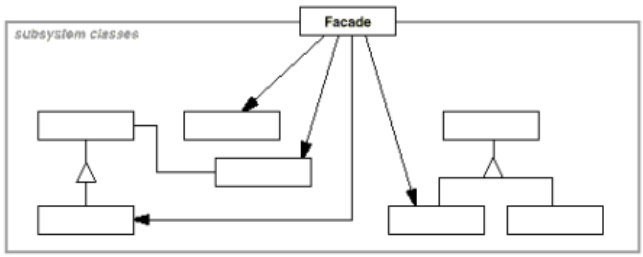
Tarkoitus
Tarjota yhteistetty rajapinta monen alijärjestelmän rajapintoihin. Fasaadi määrittää korkean tason rajapinnan alijärjestelmien helpompaa käyttöä varten.

Tunnetaan myös
-

Motivaatio
Järjestelmän hajoittaminen alijärjestelmiin helpottaa vähentämään se monimutkaisuutta. Yleinen suunnittelutavoite on minimoida kommunikaatiota ja riippuvuutta alijärjestelmien välillä. Yksi mahdollinen tapa toteuttaa kyseinen tavoite on luoda fasaadi-olio, jonka kautta alijärjestelmien ja ulkoisten järjestelmien kaikki kommunikaatio kulkee.



Käyttömahdollisuudet
Mallia käytetään kun
- halutaan tarjota yksinkertainen rajapinta monimutkaiseen järjestelmään.
- halutaan erottaa alijärjestelmät toisistaan.
- halutaan järjestää alijärjestelmät tasoihin. Jokaisen tason välille voidaan luoda fasaadi-olio, jonka kautta tasot kommunikoivat keskenään.

Rakenne


Osallistujat
- Fasaadi, joka tietää, mikä alijärjestelmä on vastuussa mistäkin pyynnöstä.
- Alijärjestelmäluokka, joka toteuttaa fasaadin kautta tulevan työn.

Yhteistoiminta
Asiakaspäätteet tekevät pyyntöjä fasaadi-oliolle, joka puolestaan lähettää pyynnöt sen toteuttavaan alijärjestelmään. Asiakaspäätteet eivät suoraan voi kommunikoida alijärjestelmille.

Seuraamukset
Fasaadi vähentää olioiden määrää, joiden kanssa asiakaspääte kommunikoi. Tämä helpottaa alijärjestelmien käyttöä asiakaspäätteen näkökulmasta.

Toteutus
Fasaadimallin abstraktoinilla mahdollistetaan vielä korkeamman tason erottaminen alijärjestelmien ja asiakaspäätteiden välille.

Esimerkkikoodi
Koodiesimerkit ovat liian pitkiä tässä taulukossa esitettäväksi.

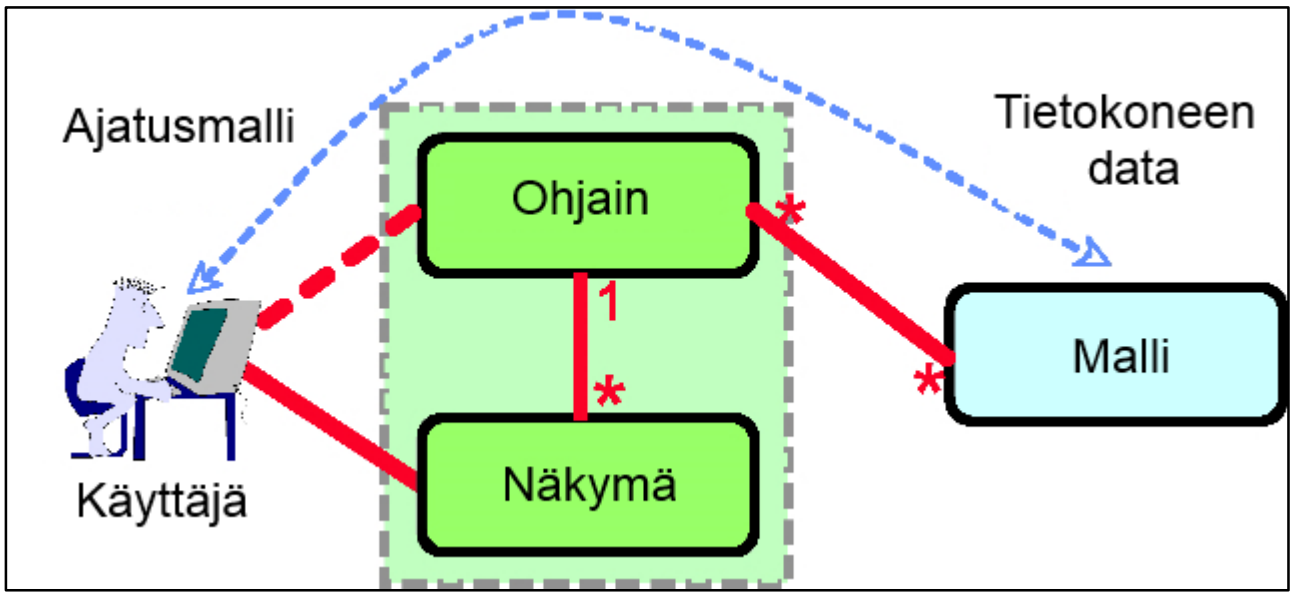
Tunnetut käytöt
ET++-ohjelmistot sekä Choices-käyttöjärjestelmä.

Liittyvät mallit
Abstract Factory -mallia voidaan hyödyntää fasaadin kanssa luomaan alijärjestelmäolioita alijärjestelmästä riippumatta. Fasaadeja tarvitaan yleensä vain yksi, joten sen toteutukseen käytetään yleensä Singeton-mallia.

Taulukko 3: Fasaadi–mallin yksinkertaistettu versio.

Olioperustaisuudella on olion tunnistamisongelman lisäksi toinen merkittävä käytännön ongelma. Suurin osa olio-ohjelmointia mahdollistavista kielistä on usean paradigman kieliä. Vaikka tämä itsessään ei ole ongelma, on nykyinen olio-ohjelmointi enemmänkin olion ohjelmointia kuin olio-ohjelmointia. Olioperustaisuuden ideasta ”…toisilleen kommunikoivista olioista” on unohdettu muut oliot ja keskitytty vain yhden olion, tai luokkaperustaisuudessa luokan, ohjelmointiin. [Coplien, 2012] Suunnittelumalleilla voidaan ratkaista ohjelmointiongelmia, mutta kommunikaatio-ongelma on kuitenkin ohjelmointikulttuurillinen ongelma [Coplien & Bjørnvig, 2010]. Reenskaug on pyrkinyt ratkaisemaan tätä kehittämällä hyvin tunnetun MVC-kehyksen, mutta sekään ei ole onnistunut ratkaisemaan kommunikaatio-ongelmaa kokonaan [Coplien & Bjørnvig, 2010].

MVC-mallin tarkoitus on erottaa käyttäjän ja järjestelmän välinen vuorovaikutus tiedon esityksestä [Reenskaug & Coplien, 2009]. Tätä erotusta hyödyntämällä on tarkoitus luoda niin kutsuttu suoran käsittelyn illuusio. Suoran käsittelyn illuusiossa (direct manipulator metaphor) järjestelmän toiminta näyttäisi käyttäjälle siltä, että käyttäjä käsittelee suoraan näytöllä olevia objekteja, vaikka ne eivät itse järjestelmässä ole objekteja. Esimerkkinä tästä toimii puhelimella soittaminen. Todellisuudessa puhelimella soittaminen tapahtuu vaiheittain ja koostuu soiton osista, mutta käyttäjälle näyttää siltä kuin puhelu olisi vain soittajan ja vastaajan välinen toiminta. MVC-malli (kuva 4) jakaa ohjelman malliin (model), näkymiin (view) ja niitä yhdistäviin ohjaimeen (controller). Malli koostuu järjestelmän tiedoista ja toimintalogiikasta, näkymä tiedon esityksestä sekä ohjain näitä yhdistävästä logiikasta ja ulkopuolisesta kommunikaatiosta. [Reenskaug & Coplien, 2009]



Kuva 4: MVC-malli [Reenskaug & Coplien, 2009].

MVC-malli auttaa käyttäjää ymmärtämään, mitä järjestelmän tiedot ovat. Ongelma MVC-mallissa ja olio-ohjelmoinnissa on järjestelmän toiminta. Järjestelmän toiminnoille ei ole olemassa määritettyä paikkaa. [Reenskaug & Coplien, 2009] MVC-mallin avullakaan ei voida vastata kysymykseen, miten oliot muodostavat järjestelmän kokonaisuuden.

Olioperustaisuuden ongelmiin kuuluu olioiden tunnistamisen lisäksi kehityssuunnan valinta ja ohjelmointikulttuuriin liittyvä kommunikoinnin vähäinen prioriteetti. Olioperustainen ohjelmistokehitys on kuitenkin tällä hetkellä yksi yleisimmistä kehitysparadigmoista. Sen merkittävimpiin vahvuuksiin kuuluu tapa kuvata asioita ihmiselle luontaisella tavalla. Lisäksi sen käyttöavaruus on mahdollisesti kaikista tämän hetkisistä kehitysparadigmoista laajin. Olioperustainen ohjelmistokehitys on saanut suosiota myös järjestelmäkehityksen tasolla, jossa muutos vesiputousmallista ketterään kehitykseen on asettanut uusia vaatimuksia ohjelmistojen ylläpidettävyydelle.

2.4 Olio-ohjelmoinnin metriikat

Ohjelmoinnissa käytetään monia metriikoita järjestelmän laadun mittaamiseksi. Monet laatuvaatimukset ovat kuitenkin subjektiivisia ja alalla on ollut merkittäviä vaikeuksia löytää yksiselitteisiä metriikoita. [Chhabra & Gupta, 2009] Tässä tutkielmassa on kiinnostuttu mittaamaan järjestelmän ylläpidettävyyttä. Järjestelmän kompleksisuus on ymmärrettävyyteen ja siten ylläpidettävyyteen liittyvä käsite, johon on olemassa mittareita. Järjestelmän kompleksisuus käsittää järjestelmän ymmärrettävyyttä kooditasolla. Kompleksisuus pyrkii mittaamaan koodista ei-subjektiivisia ymmärrettävyyteen liittyviä arvoja ja ominaisuuksia.

Yleisesti ohjelmoinnissa järjestelmän syklomaattinen kompleksisuus (cyclomatic-complexity, CC) kertoo ehtorakenteiden luomasta monimutkaisuudesta. Syklomaattinen kompleksisuus voidaan olettaa yleisesti tunnetuksi, joten sen tarkempi käsittely jätetään tutkielman ulkopuolelle. Olio-ohjelmoinnissa kompleksisuutta voidaan arvioida perintöpuiden korkeudella ja luokkien koolla. Reenskaug [2007] huomioi myös olio-ohjelmoinnin monimuotoisuuden (polymorphism) vaikuttavan järjestelmän luettavuuteen ja siten ymmärrettävyyteen.

Gupta ja Chhabra [2011] ovat esittäneet tavan laskea olioperustaisen ohjelman yhtenäisyyttä. Luokan tai olion yhtenäisyys kertoo, kuinka paljon niiden metodit tai attribuutit ovat riippuvaisia toisistaan. Mitä suurempi riippuvuus on, sitä vaikeampi niitä on muuttaa vaikuttamatta toiseen. Dynaaminen yhtenäisyys lasketaan olion suhteessa johonkin skenaarioon. Skenaario on järjestelmän ajonaikainen tapaus, jossa tutkittavaa oliota käytetään. Dynaamisen yhtenäisyyden laskukaavat löytyvät liitteestä A.

Olioperustaisen ohjelman kompleksisuutta voidaan mitata myös kognitiivis-spatiaalisella kompleksisuudella, joka on yhdistelmä spatiaalisesta ja kognitiivisesta kompleksisuudesta. Spatiaalinen kompleksisuus arvioi, kuinka vaikeaa lähdekoodia on ymmärtää.

Spatiaalisessa kompleksisuudessa lasketaan metodien ja attribuuttien määritelmien ja niiden käytön välinen etäisyys. Etäisyytenä käytetään koodirivien välistä eroa. Kognitiivisella kompleksisuudella tarkoitetaan ohjelman kontrollirakenteiden monimutkaisuutta. [Gupta & Chhabra, 2009] Jokaiselle kontrollirakenteelle on löydetty sitä vastaava painoarvo (basic control structure, BCS). Kontrollirakenteet ja niiden painot löytyvät taulukosta 4. Rivien, joilla on useampi kontrollirakenne, painoarvot lasketaan summaamalla sisäkkäisten kontrollirakenteiden painoarvot. Tällainen rivi on esimerkiksi iteraattorin sisällä oleva if-lause. [Shao & Wang, 2003]

Kontrollirakenne	Paino	Esimerkkejä
Sekvenssi	1	Yksinkertainen komentorivi
Yhden tapauksen ehtolause	2	if-else
Usean tapauksen ehtolause	3	Case, sisäkkäiset if-else if
Iteraattorit	3	for, while jne
Funktiokutsu	2	Vain käyttäjän määrittämät kutsut
Rekursio	3	
Rinnakkaisuus ja keskeyttäjät	4	

Taulukko 4: Kontrollirakenteiden painoarvot (BCS) [Shao & Wang, 2003].

Gupta ja Chhabra ovat muodostaneet näistä kognitiivis-spatiaalisen metriikan, jolla voi laskea sekä luokan että olion kognitiivis-spatiaalisen kompleksisuuden. Kognitiivis-spatiaalinen kompleksisuus mittaa olioiden ja luokkien määritelmän ja käytön välistä etäisyyttä huomioiden käyttötilanteen monimutkaisuuden. Kognitiivis-spatiaalinen kompleksisuuden laskukaavat on esitetty liitteessä B.

2.5 Ketterä ohjelmistokehitys ja lean

Ketterä ohjelmistokehitys (agile software development) on yleistermi ketterän ajatusmallin ohjelmistokehitysmenetelmille. Ketterän ohjelmistokehityksen pohjaksi on luotu ketteryyden manifesti (agile manifesto) [Agile Manifesto, 2001], joka pitää sisällään ketterän ohjelmistokehityksen ideologian. Ketteriä menetelmiä on useita, mutta yhteistä niille on, että ne ovat lähes poikkeuksetta inkrementaalisia. Inkrementaalisuus tarkoittaa ohjelmiston vaiheittaista kehitystä siten, että järjestelmästä saadaan eri versioita testattavaksi määritellyin väliajoin. Inkrementaalimalleja on useita, mutta pääosin ne ovat joko prototyyppi- tai versiopohjaisia. Prototyyppipohjaisessa kehityksessä testiversiot ovat hyvin hiomattomia kokonaisuuksia, joilla voi testata jotain järjestelmän osa-aluetta. Versiopohjaisessa kehityksessä järjestelmän jokainen versio on toimiva järjestelmä ja jokainen uusi versio lisää vanhempaan jotain uutta, esimerkiksi toiminnollisuutta tai tehokkuutta. Riippumatta inkrementaalisuuden toteutuksesta, sen merkittävin hyöty on varmistaa järjestelmän kehittyminen haluttuun suuntaan, sillä jokaisesta versiosta ja

prototyypistä on tarkoitus saada palautetta asiakkaalta. Inkrementaalinen ohjelmistokehitys on vastaus vesiputousmallin ongelmalliseen hitauteen muuttuvassa ympäristössä, jossa järjestelmän vaatimukset voivat muuttua jo kehitysvaiheessa. Erona inkrementaaliseen kehitykseen vesiputousmallissa on se, että siinä on pääasiallisesti tarkoitus tehdä ohjelman määrittäminen, suunnittelu, ohjelmointi ja ylläpito kerralla oikein, mainitussa järjestyksessä. Ketteryyden ajatus keksittiin 1990-luvulla, ja se on saanut jatkuvasti vankempaa asemaa ohjelmistokehityksessä. [Sommerville, 2011]

Ketterät menetelmät pohjautuvat ketteryyden ideologiseen pohjaan, joka on selitetty ketteryyden manifestissa. Manifesti koostuu neljästä pääajatuksesta: ”Yksilöitä ja kanssakäymistä enemmän kuin menetelmiä ja työkaluja; Toimivaa ohjelmistoa enemmän kuin kattavaa dokumentaatiota; Asiakasyhteistyötä enemmän kuin sopimusneuvotteluja; Vastaamista muutokseen enemmän kuin pitäytymistä suunnitelmassa.” [Agile Manifesto, 2001]

Ketterät menetelmät, ainakin ideologisella tasolla, keskittyvät muutosten hallintaan ja nopeaan järjestelmän kehitykseen. Ketterät menetelmät saavuttavat tätä nopeutta ja toimintaa hyödyntämällä kehitystiimin yksilöosaamista. Yleensä kehitystiimissä on itseohjautuvia laajan alan osaajia. [Hayata et al., 2012]

Ideologisella tasolla ketterät menetelmät kuulostavat hyvältä, mutta niiden toteutus on vaikeaa. Olio-ohjelmointi on ollut luonteva apu ketterässä kehityksessä, sillä se sopeutuu ylläpidollisesti paremmin muutoksiin kuin muut yleiset ohjelmointimenetelmät. Ketteryyttä on kuitenkin kritisoitu sen järjestelmällisyyden ja ohjeistuksen puutteesta. Puutteiden johdosta menetelmiä ei ole otettu käyttöön koko kehityksessä tai hyötyjä on haluttu haalia luopumatta kuitenkaan vanhoista menetelmistä. [Hayata et al., 2012] Nämä näkyvät esimerkiksi manifestin neljään ideaan tehdyillä lisähuomioilla [Buckley, 2015], jotka parodisoivat alkuperäisiä ajatuksia.

Ketterät menetelmät keskittyvät nopeaan järjestelmän kehitykseen ja luovutukseen dokumentaation ja suunnitelmien sijaan. Nopea kehitys on lähes välttämätön muuttuvassa ympäristössä, mutta se johtaa monesti jo tehtyjen osien paranteluun tai uudelleen tekemiseen, mikä hyvällä suunnittelulla olisi voitu välttää. ”Suunnittelu” (design) on monesti tulkittu ”suunnitelmana” (plan), jonka päätöntä seuraamista manifestin mukaan tulisi pyrkiä välttämään. Suunnittelun ja kehityksen tulisi olla sellaisessa tasapainossa, ettei kumpaakaan tehdä turhaan. Ohjelmistokehityksessä tulevan turhan työn minimoimiseksi on ehdotettu lean-kehitysideologiaa. Lean korostaa suunnittelun osuutta kehityksessä turhan työn vähentämiseksi. [Hayata et al., 2012]

Lean-ohjelmistokehitys on ohjelmistokehityksen versio lean-ajattelusta. Lean-ajattelu muodostui 1990-luvulla, mutta se pohjautuu pitkälti Toyotan Toyota Production Systemiin vuosilta 1948–1975. [Strategos, 2015] Lean-ajattelu perustuu seitsemän tuotteistamisen

jätteen poistamiseen. Tuotteistamisen jätteitä ovat varasto, ylimääräinen käsittely, ylituotanto, kuljetus, odotus, liike ja viat. Jätteet voidaan kääntää ohjelmistokehityksen kannalta olennaisemmin seuraavasti: osittain tehty työ, ylimääräiset prosessit, ylimääräiset toiminnot (features), työtehtävien vaihtelu, odotus, liike ja viat. [Poppendieck & Poppendieck, 2003] Jätteiden tarkemmat selitykset löytyvät taulukkosta 5.

Jätteen nimi	Jätteen kuvaus
Osittain tehty työ (partly done work)	Käsittää kaiken osittain valmiiksi tehdyn työn, jonka toimintavarmuutta tai hyötyä ei voida varmistaa ennen kuin järjestelmä on tuotannossa. Osittain tehty työ haalii resursseja muista projektin osioista ja on suuri taloudellinen riski.
Ylimääräiset prosessit (extra processes)	Käsittää turhan paperityön. Projektiin tehtävät dokumentit vievät resursseja ja hidastavat projektin reaktioaikaa. Dokumentit häviävät, vanhenevat nopeasti ja peittävät laadullisia ongelmia. Dokumentti, jota kukaan ei lue, on täysin arvoton.
Ylimääräiset toiminnot (extra features)	Ohjelmistokehittäjillä voi tulla halu lisätä järjestelmään toimintoja, joita voidaan tarvita myöhemmin. Toimintoja saatetaan lisätä myös siksi, että halutaan testata järjestelmän teknistä kapasiteettia. Ylimääräinen koodi esiintyy ohjelmoinnin lisäksi muun muassa testauksessa, kääntämisessä ja integraatiossa. Ylimääräistä toimintoa ei edes välttämättä käytetä, joten sen toteuttaminen on ollut turhaa. Järjestelmään tulisi toteuttaa vain ne toiminnot, joita sillä hetkellä tarvitaan.
Työtehtävien vaihtelu (task switching)	Ihmisten tulisi työskennellä yhdessä projektissa kerrallaan. Usean projektin samanaikainen työstäminen hidastaa työntekijän toimintaa. Projektista toiseen vaihtaminen vaatii ajatuksien siirtämistä aiemmasta projektista uuteen, mikä vie aikaa. Mitä enemmän projektien välillä pitää vaihdella, sitä enemmän aikaa menee ajatusmaailman muuttamiseen. Usean projektin työstäminen asettaa myös sen työntekijät suurempaan riskiin tulla keskeytetyksi työnteon aikana, joka vähentää tehokasta työaikaa edelleen.
Odotus (waiting)	Odotus on yksi suurimmista jätteistä ohjelmistokehityksessä. Odotus käsittää kaiken projektin aloittamisen viivytyksistä järjestelmän sijoituksen viivytyksiin. Projektin sisäiset viiveet näkyvät sen reaktionopeudessa esimerkiksi kriittisiin muutoksiin.
Liike (motion)	Liike käsittää tilanteet, joissa jokin asia liikkuu kehityksen aikana. Liikkuvia asioita ovat ihmiset, mutta myös erilaiset projektin artefaktit kuten suunnitteludokumentit tai lähdekoodi. Ihmisten liikettä ovat esimerkiksi teknisten ongelmiin avun saanti tai asiakkaalta kysyttävä lisätieto. Artefaktien liikettä on esimerkiksi suunnitteludokumentin siirtäminen analysoijalta suunnittelijalle. Liikkeen minimoimiseksi, esimerkiksi ketterässä kehityksessä, vaaditaan kaikkien projektin työntekijöiden olevan lähellä toisiaan.
Viat (defects)	Isojenkin vikojen löytäminen alussa maksaa merkittävästi vähemmän kuin pienten vikojen löytyminen myöhään. Vikojen aiheuttamien ongelmien minimoimiseksi ne tulisi tunnistaa mahdollisimman ajoissa. Niiden tunnistamiseksi järjestelmää tulisi testata välittömästi, integroida useasti ja julkaista järjestelmä mahdollisimman nopeasti.

Taulukko 5: Leanin jätteet [Poppendieck & Poppendieck, 2003].

Lean-ajattelun, ja siten myös lean-ohjelmistokehityksen, pääasiallisena tavoitteena on luoda tuotannon arvovirran (value stream) kuvaus. Arvovirta tarkoittaa kaikkia vaiheita asiakkaan tarpeesta tuotteen tai palvelun toimittamiseen. [Poppendieck & Poppendieck, 2003] Arvovirran kuvauksen perusteella on helpompi hahmottaa, mitkä aktiviteetit tuottavat arvoa ja mitkä tuottavat jätettä. Ohjelmistokehityksessä on vähemmän fyysisiä komponentteja kuin varsinaisessa tuotteistuksessa ja laadun mittaaminen perustuu pitkälti hyvien käytäntöjen hyödyntämiseen. [Sommerville, 2011] Ohjelmistokehityksen arvovirran tulisi pohjautua projektin sidosjäseniin (stakeholder) ja itse ohjelmistokehityksen vaiheiden tulisi pohjautua sen arvovirtoihin [Coplien & Bjørnvig, 2010]. Lean-ohjelmistokehityksen pohjaaminen sidosjäseniin tarkoittaa, että projektin hyödyllisyyden varmistamiseksi näitä sidosryhmiä pitäisi hyödyntää parhaalla mahdollisella tavalla. Tästä Coplien ja Bjørnvig [2010] käyttävät ilmausta ”The Lean Secret”, jolla he tarkoittavat yksinkertaisesti ”Every body, All together, Early On”, eli ”Kaikki, yhdessä, aikaisin”.

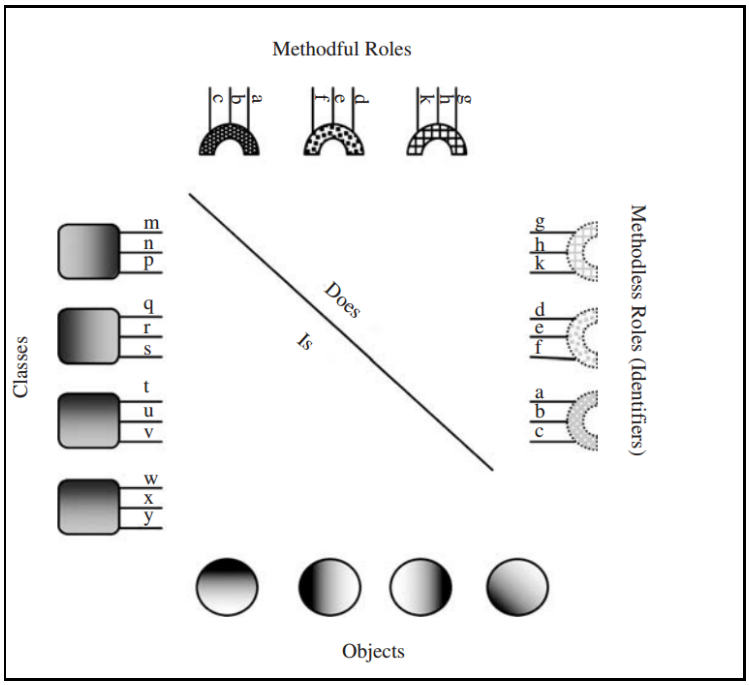
Lean keskittyy jätteen poistamiseen ja arvovirran visualisoimiseen tavoitteenaan parantaa projektin ja siten järjestelmäkehityksen toimintaa. Ketterät menetelmät keskittyvät nopeaan reagointiin muutoksissa, mutta monesti luovat lisätyötä uudelleen tekemisen tai parantelun muodossa. Leanin ja ketterien menetelmien yhdistäjänä on nähty järjestelmän arkkitehtuuri. Arkkitehtuurin kehittäminen pakottaa ajattelemaan järjestelmän toimintaa ja tulevaisuutta. Ketterät menetelmät monesti ohittavat arkkitehtuurisuunnittelun, koska perinteisesti arkkitehtuuri keskittyy vain projektin alkupäähän ja ottaa liikaa kantaa järjestelmän toteutukseen. Coplienin ja Bjørnvigin [2010] mukaan arkkitehtuurin on tarkoitus antaa järjestelmälle muoto, eikä ottaa kantaa sen varsinaiseen toteutukseen. Lean-ajattelua ja ketteriä menetelmiä yhdistävän arkkitehtuurin on oltava perinteistä arkkitehtuuria abstraktimpi sekä siten myös kevyempi.

DCI (Data-Context-Interaction) on Reenskaugin luoma sekä hänen ja Coplienin kehittämä usean paradigman arkkitehtuuri. DCI-arkkitehtuurin perusideana on pyrkiä jakamaan kehitettävä ohjelma selkeämmin kahteen osaan: mikä-järjestelmä-on eli järjestelmän toimialalogiikkaan (what-the-system-is) ja mitä-järjestelmä-tekee eli järjestelmän toimintoihin (what-the-system-does). Tällä jaottelulla järjestelmän hitaasti muuttuvat osat, oleellisesti toimialaan liittyvät logiikat, saadaan erotettua sen nopeasti muuttuvista osista, asiakkaan tarpeisiin vastaavista toiminnoista. DCI-arkkitehtuuri on Reenskaugin ja Coplienin ratkaisu olio-ohjelmoinnin kommunikaatio-ongelmaan ja lean-ajattelun sekä ketterän kehityksen yhdistämiseen. [Coplien & Bjørnvig, 2010]

3 DCI-arkkitehtuuri

DCI-arkkitehtuurin on luonut Trygve Reenskaug. Reenskaug on ollut ohjelmistokehityksessä mukana yli 50 vuotta ja on kehittänyt Malli-Näkymä-Ohjain-mallin (model-view-controller framework, MVC) vuonna 1979. MVC-malli on edelleen yksi tunnetuimmista ohjelmointikehyksistä ja sen tarkoitus on luoda illuusio järjestelmän toiminnasta siten, että käyttäjä kokisi järjestelemän toimivan hänen ajatusmallinsa mukaisesti. Käyttäjän ajatusmalli on tärkeä konsepti ohjelmistokehityksessä, sillä käyttäjän ajatusmallin ja järjestelmän toimintamallin sotkeutuminen aiheuttaa käytettävyysongelmia. [Nielsen, 2010]

MVC-malli pyrki oleellisesti erottamaan käyttäjälle näkyvän osan järjestelmän logiikasta, mutta silti järjestelmän toimialalogiikka heijastui käyttäjille. DCI korjaa tilanteen siten, että toimialalogiikka ja toiminnot on erotettu jo alusta asti (kuva 5). DCI-arkkitehtuurin pohjimmainen tarkoitus on saada järjestelmä toimimaan käyttäjän ajatusmallin mukaisesti, jotta järjestelmän toimintamallin ja käyttäjän ajatusmallin välillä ei olisi suurta eroa. [Reenskaug & Coplien, 2009]



Kuva 5: DCI-arkkitehtuuri [Coplien & Bjørnvig, 2010].

3.1 Toimialalogiikka

DCI pyrkii jakamaan järjestelmän toimialalogiikkaan ja toimintoihin. Toimialalogiikka voi olla hyvin erilaista eri toimialojen välillä, eikä sen yleensä tarvitse näkyä käyttäjälle. Toimialalogiikka on käytännössä se osa järjestelmää, joka kuvaa varsinaisen yritystoiminnan tiedot ja järjestelmän tilan. [Coplien & Bjørnvig, 2010]

Toimialalogiikka tehtynä olio-ohjelmoinnilla kuvaa luokkapohjaisesti toteutettavan järjestelmän tilat ja tiedot. Perinteisestä olio-ohjelmoinnista poiketen DCI-arkkitehtuurissa näihin luokkiin ohjelmoidaan mahdollisimman vähän järjestelmän toimintojen logiikkaa. Toimialalogiikan toteutus on toimintoja vähemmän muuttuva osa järjestelmässä, joten sen tutkimiseen ja suunnitteluun tulisi käyttää tarpeeksi resursseja. [Coplien & Bjørnvig, 2010] Toimialalogiikka tukee myös iteratiivista kehitystä siten, että kaikkea ei ole pakko ohjelmoida kokonaan, vaan toimialalogiikan osioista voidaan ohjelmoida vain sen hetkisten toimintojen vaatimat osiot. [Simola, 2012]

3.2 Toiminnot

Toiminnot ovat asioita, joita varten järjestelmää käytetään. Toiminnot ovat käyttäjän varsinainen kiinnostuksen kohde ja siten oleellinen osa heidän ajatusmalliaan. DCI jakaa toiminnot kontekstiin (context), rooleihin (roles) ja niiden vuorovaikutukseen (interaction).

Konteksteilla kuvataan käyttäjän tekemiä toimintoja järjestelmässä. Järjestelmän kontekstit ovat pitkälti sen käyttötapaukset (use case). Kontekstit ilmentyvät järjestelmässä olioina, jotka ovat varustettuna jonkinlaisella ”käynnistä” metodilla. Konteksti on oma kokonaisuus, joka alustuksen jälkeen tehdään kutsumalla esimerkiksi DoIt-metodia.

Toisin kuin kontekstit, jotka ovat olioita järjestelmässä, vuorovaikutus viittaa rooleihin ja niiden kommunikaatioon. Roolit ovat järjestelmän toimintojen logiikkaa ja ne on paketoitu yhteisen nimittäjän alle. Roolien käsitettä voidaan selittää näytelmämetaforalla. Näytelmässä näyttelijöillä on erilaisia roolihahmoja, jotka määrittävät näyttelijän esiintymistavan näytelmässä. DCI-arkkitehtuurissa näyttelijää vastaa data-olio, roolihahmoja roolit ja näytöstä konteksti. Rooliperustaisuutta on kehitetty 1990-luvulla ja DCI voidaan mieltää sen osittaisena jatkeena. Roolien olemassaolo johtuu ihmisten tavasta kuvata asioita. Ihmiselle on luontaista elää maailmassa, jossa eri roolit antavat niiden toteuttajilleen ominaisuuksia, jotka ovat joskus ristiriitaisia. Ihminen voi esimerkiksi työssään olla esimies, kotonaan vanhempi ja illalla tuttavien kesken ystävä. Jokaisessa roolissa kyseinen ihminen voi toimia eri tavalla. Roolit ovat tärkeä osa ihmisen ajatustapaa, ja tämä näkyy myös ihmisen tavassa luoda ajatusmalleja järjestelmästä. [Coplien & Bjørnvig, 2010] Roolit ovat luonteva tapa esittää ominaisuuksia, joita käyttäjä mieltää yhteenkuuluviksi ja näin järjestelmä saadaan toimimaan käyttäjän ajatusmallia läheisemmin. Järjestelmän kehityksessä luodaan käyttötapauksia järjestelmän vaadituista toiminnoista. Käyttötapauksia analysoimalla haetaan järjestelmästä löytyvät roolit. Käyttötapauksista luodaan kontekstit. [Coplien & Bjørnvig, 2010]

3.3 Malliesimerkki

Coplien ja Bjørnvig [2010] havainnollistavat DCI-arkkitehtuuria pankkiesimerkillä. Pankkiesimerkissä tapahtuu kahden tilin välinen tilisiirto. Tilisiirrosta on tehty käyttötapaus, jonka avulla siitä muodostetaan DCI-arkkitehtuurin mukainen toteutus.

Toiminnot ilmenevät järjestelmässä aluksi skenaariokuvauksina ja käyttötapauksina, joita analysoimalla muodostetaan järjestelmän roolit. Pankkiesimerkissä on rahan siirto -käyttötapaus (taulukko 6), josta on muodostettu teknisempi versio (taulukko 7). Kyseisestä käyttötapauksesta on löydetty kaksi roolia: RahaLähde (MoneySource) ja RahaReikä (MoneySink). Käyttötapauksen teknisestä versiosta muodostetaan myöhemmin konteksti. [Coplien & Bjørnvig, 2010] Pankkiesimerkkin avulla havainnollistetaan data-osiota, rooleja ja konteksteja seuraavissa kohdissa.

Use case name:	Transfer Money	
User intent:	To transfer money between his or her own accounts.	
Motivation:	The AccountHolder has an upcoming payment that must be made from an account that has insufficient funds.	
Preconditions:	The AccountHolder has identified himself or herself to the system.	
Happy day scenario:		
Step	**Actor Intention**	**System Responsibility**
1	AccountHolder enters a source account and requests an accounttransfer	System displays the source account, provides a list of valid destination accounts, and a field to enter the amount.
2	AccountHolder selects a destination account, enters the amount, and accepts	System displays transfer information and requests a password.
3	AccountHolder enters the password and accepts the transfer	System moves money and does accounting.
Variations:		
1a. AccountHolder has only one account: tell the AccountHolder that this cannot be done.		
2a. The accounts do not exist or are invalid…		
Post-conditions:		
The money is moved.		
The accounts balance.		
The log reflects.		

Taulukko 6: Esimerkin käyttötapaus [Coplien & Bjørnvig, 2010].

1. **Source Account** verifies funds available (deviation 2a in Table 7-4)	1. Source account begins transaction.
	2. Source account verifies that its current balance is greater than the minimum account balance plus the withdrawal amount, and throws an exception if not.
2. **Source Account** and **Destination Account update their balances**	3. Source account reduces its own balance by the amount.
	4. Source account requests that Destination Account increase its balance.

3. **Source Account** updates statement information	5. Source account updates its log to note that this was a transfer.
	6. Source account requests that Destination Account update its log.
	7. Source account ends transaction.
	8. Source account returns status that the transfer has succeeded.

Taulukko 7: Esimerkin roolit [Coplien & Bjørnvig, 2010].

3.4 Data

Data on kokonaisuus, joka pitää sisällään järjestelmän tilan. DCI-arkkitehtuurissa data pyritään pitämään mahdollisimman yksinkertaisena ja staattisena. Mitä lähempänä datan kuvaus on varsinaista toimialalogiikkaa ja mitä yksinkertaisempi sen toteutus on, sitä vähemmän se nähdään muuttuvan ohjelman elinaikana. DCI-arkkitehtuurissa lean-ajattelua tulisi soveltaa dataosion arkkitehtuurin kehittämiseen siten, että se tehdään kerralla oikein. [Coplien & Bjørnvig, 2010]

Data kuvataan monesti luokilla, mutta se voi olla myös puhtaasti rajapintamäärittely. Rajapinnan avulla voidaan erottaa varsinaiset toteutukset muun ohjelman suunnittelusta. Data on tarkoitus määritellä mahdollisimman hyvin projektin alussa. Tekemällä rajapintamäärittelyt aikaisessa toteutusvaiheessa voidaan järjestelmää tehdä iteratiivisesti ja siten lean-tyylisesti siirtää toteutusta mahdollisimman myöhäiseen vaiheeseen. [Hayata et al., 2012]

Riippumatta siitä, kuvataanko data suoraan luokilla tai rajapinnoilla, niistä syntyviä oliota käytetään vuorovaikutusosiossa. Oliot ovat tässä vaiheessa hyvin yksinkertaisia ja pääosin toteuttavat vain käsittelijöitä (accessors), joilla olion tilatietoihin päästään käsiksi. Oliolla voi mahdollisesti olla omat varmistukset tietojen oikeellisuudesta tai muuta järjestelmän toiminnollisuudesta riippumatonta logiikkaa. Data-olioille sijoitetaan ajon aikana rooleja, joista ne saavat varsinaiset toiminnallisuudet, jotka hyödyntävät tai muokkaavat data-olioiden tilaa. [Coplien & Bjørnvig, 2010]

Data-oliot pitävät sisällään järjestelmän tilan. Pankkiesimerkissä käytetään oliota SäästöTili data-oliona (koodikatkelma 1). Rahan siirto -käyttötapauksessa SäästöTili näyttelee roolia RahaLähde. KohdeTili on myös data-olio, ja rahan siirrossa se näyttelee RahaReikä-roolia. SäästöTili ja KohdeTili pitävät sisällään vain saldonsa tiedot sekä tavan suurentaa tai pienentää sitä summalla x. Todellisuudessa oikeina luokkina toimivat jäljitysketjut sekä transaktiokirjaukset. [Coplien & Bjørnvig, 2010]

```
// Abstract domain object
public abstract class Account {
   public abstract void DecreaseBalance(double amount);
   public abstract void IncreaseBalance(double amount);
   public abstract void Log(string message);}
// Concrete domain object
public class SavingsAccount :Account, TransferMoneySource, TransferMoneySink
{
   private double balance;
   public SavingsAccount() { balance = 10000;}
   public override void DecreaseBalance(double amount) { balance -= amount;}
   public override void IncreaseBalance(double amount) { balance += amount; }
   public override void Log(string message) { Console.WriteLine(message); }
   public override string ToString() { return "Balance " + balance; }
}
```

Koodikatkelma 1: Esimerkin data-luokat, C# [Coplien & Bjørnvig, 2010].

3.5 Roolit

Järjestelmän toiminnollisuus, eli oleellisesti järjestelmän palvelut, ovat koko järjestelmän ydin. Käyttäjät hyödyntävät järjestelmää yleensä työkaluna jonkin asian toteuttamiseen, joten sen tulisi palvella käyttäjän tarpeita. Ihmisen ajatusmallit luovat rooleja ja siten DCI-arkkitehtuurissa roolit saavat keskeisen aseman. [Coplien & Bjørnvig, 2010]

Roolit sisältävät järjestelmän toimintojen logiikan. Aiemmin mainitut data-oliot toteuttavat niille määritettyjä rooleja, mutta roolien sisällä olioon viitataan yleistyksellä. Yleistyksellä varmistetaan, että jokainen roolia toteuttava olio tekee sen samalla logiikalla. Rooleissa on avainsana ”self”, jolla rooli viittaa sitä toteuttavaan data-olioon.

Pankkiesimerkin rahan siirto -käyttötapauksesta löydettiin RahaLähde ja RahaReikä -roolit (koodikatkelma 2). RahaLähde-roolilla on metodi siirräRahaa. SiirräRahaa-metodille annetaan tieto summasta ja viite RahaReikä-tyyppiseen olioon. SiirräRahaa-metodi tarkistaa aluksi, onko kyseisellä data-oliolla, joka roolin suorituksen aikana kutsutaan avainsanalla self, katetta summan siirtämiseen. Metodi vähentää RahaLähde-roolia näyttelevän olion saldoa ja lisää RahaReikä-roolia näyttelevän olion saldoa, mikäli RahaLähteellä oli katetta. Riippumatta siitä tapahtuiko siirto, metodi tekee kirjaukset tapahtuman kulusta.

```
// Methodless role types
public interface TransferMoneySink { }
// Methodful roles
public interface TransferMoneySource { }

public static class TransferMoneySourceTraits
{
   public static void TransferFrom(this TransferMoneySource self, TransferMoneySink recipient, double amount)
   {
      Account self_ = self as Account;
      Account recipient_ = recipient as Account;

      if (self_ != null && recipient_ != null) {
         self_.DecreaseBalance(amount);
         self_.Log("Withdrawing " + amount);
         recipient_.IncreaseBalance(amount);
         recipient_.Log("Depositing " + amount);
      }
   }
}
```

Koodikatkelma 2: Esimerkin roolit, C# [Coplien & Bjørnvig, 2010].

3.6 Kontekstit

Kontekstit ovat data-olioiden ja roolien yhdistäjiä sekä kutsujia. Näytelmämetaforassa konteksti on näytös, joka alkaa näyttelijöiden (data-oliot) roolituksella, ja jatkuu näytöksen ohjaamisella ja näytöksen lopetuksella. Käytännössä järjestelmän kontekstit ilmenevät sen käyttötapauksilla. Käyttötapaukset pitävät sisällään järjestelmän korkean tason algoritmin jonkin toiminnon toteutuksen. Käyttötapauksista muodostetaan konteksti-olio, joka pitää sisällään teknillisemmän version käyttötapauksesta. Toimintoon liittyvän algoritmin löytyminen koodista edesauttaa järjestelmän testaamista sen toimintojen suhteen, ei pelkästään loogisten virheiden varalta. [Reenskaug & Coplien, 2009]

Konteksti-oliot toimivat kahdessa vaiheessa. Ensimmäisessä vaiheessa määritetään, mitä rooleja siinä käytetään, ja vaadittavat data-oliot sijoitetaan näihin rooleihin. Tätä kutsutaan roolien kartoitukseksi (role-mapping). Toisessa vaiheessa konteksti ajetaan, eli suoritetaan siihen ohjelmoitu algoritmi. Kontekstit voivat luoda myös sellaisen kokonaisuuden, että se voi esiintyä toisessa kontekstissa jossain roolissa. [Reenskaug & Coplien, 2009]

Rahan siirto oli pankkiesimerkissä käyttötapaus ja siitä on luotu konteksti (koodikatkelma 3). Oleellisesti käyttötapaus koostui kolmesta vaiheesta: käyttäjä määrittää tilin, jolta rahaa siirretään; käyttäjä määrittää tilin, johon rahaa siirretään ja millä summalla; käyttäjä varmistaa oikeutensa tehdä siirto ja hyväksyy sen. Esimerkissä käyttötapauksesta on tehty RahanSiirtoKonteksti (koodikatkelma 3), jolla on tietonaan RahaLähde ja RahaReikä -rooliviitteet sekä summa. Konteksti-olioilla on DoIt-metodi, joka aloittaa kyseisen käyttötapauksen suorittamisen. RahanSiirtoKontekstin DoIt–metodi sisältää vain kutsun RahaLähde-olion TransferTo-metodiin, jonka parametreiksi annetaan kontekstiin syötetty summa ja RahaReikä-tyyppinen olio. Konteksti alustetaan ulkopuolelta sen vaatimilla olioilla ja summalla. [Coplien & Bjørnvig, 2010]

```
// Context object
public class TransferMoneyContext
{
   public TransferMoneySource Source {get; private set;}
   public TransferMoneySink Sink {get;private set;}
   public double Amount {get; private set;}
   public TransferMoneyContext() {
   // logic for retrieving source and sink accounts
   }
   public TransferMoneyContext(TransferMoneySource source, TransferMoneySink sink, double amount) {
      Source = source;
      Sink = sink;
      Amount = amount;
   }
   public void DoIt() {
      Source.TransferTo(Sink, Amount);
   }
}
```

Koodikatkelma 3: Esimerkin konteksti, C# [Coplien & Bjørnvig, 2010].

3.7 Yhteenveto

DCI-arkkitehtuuri koostuu datasta, kontekstista, rooleista ja niiden vuorovaikutuksesta. Data kuvaa järjestelmän toimialalogiikan data-oliota, kontekstit ovat karkeasti järjestelmän käyttötapaukset ja vuorovaikutus käsittää järjestelmän roolit. DCI jakaa järjestelmän kahteen osaan: toiminnot ja toimialalogiikka.

Toimialalogiikka kehitetään lean-ajattelulla ja siinä panostetaan arkkitehtuurin kuvaamiseen data-oliolla. Arkkitehtuuriin voi panostaa, sillä toimialalogiikka muuttuu vain vähän järjestelmän aikana, ja siitä järjestelmä saa rungon muuttuvien osien alle.

Toiminnot ovat pitkälti riippuvaisia käyttäjien ajatusmalleista. Järjestelmän toiminnot kuvataan rooleiksi, jotka kommunikoivat keskenään kontekstin sisällä. Jokainen järjestelmän käyttötapaus on konteksti ja käyttötapauksista analysoidaan järjestelmän roolit. Järjestelmän varsinainen kulku tapahtuu kontekstien välillä ja konteksteista tulisi löytyä käyttötapauksien skenaario DoIt-metodissa.

Järjestelmän toimialalogiikka muuttuu paljon hitaammin kuin toiminnot. Toimintoja muutetaan ja räätälöidään monesti asiakaskohtaisesti, mutta toimialalogiikan pohja pysyy usein muuttumattomana. DCI painottaa keskittymään toimintojen suunnittelussa sen käyttäjiin. Käyttäjien ajatusmallit ovat keskeisiä toimintojen roolien ja niiden kommunikaation määrittämisessä.

DCI-arkkitehtuuria on alettu kehittämään 2000-luvulla, ja se sai muotonsa vuonna 2008. DCI on tutkimusaiheena varsin uusi, eikä siitä ole vielä montaa tutkimusta. DCI-arkkitehtuurista tehdyt tutkimukset ovat olleet myös pitkälti teoreettisia ja käytännön esimerkkejä on vähän. Osasyy esimerkkien vähyyteen on ohjelmointikielten ongelmat toteuttaa DCI-arkkitehtuuri, sillä perinteiset kielet kuten Java ja C++ eivät tue sitä. DCI voidaan saada toimimaan C++:ssa moniperinnän avulla, mutta Java ei pysty toteuttamaan sitä muuttamatta kääntäjää. Javaan on olemassa Qi4J-kehys, jonka avulla siihen saadaan erilaisia ominaisuuksia, joilla DCI-arkkitehtuuri voidaan toteuttaa.

Tutkielmaan on tehty malliesimerkkiä laajempi esimerkkipeli, jonka toiminta ja rakenne käydään läpi seuraavassa luvussa.

4 DCI-arkkitehtuurilla toteutettu peli

4.1 Pelin esittely

DCI-arkkitehtuurin perusesimerkkinä käytetään kahden tilin välistä tilisiirtoa. Vastaavia esimerkkejä löytyy myös muita ja siksi tässä tutkielmassa on haluttu esittää DCI-arkkitehtuuri jostain muusta aiheesta. Tutkielman aiheeksi on valittu peli. Pelit vaihtelevat monimutkaisuudeltaan äärimmäisen paljon muutamasta koodirivistä satoihin tuhansiin. Ohjelmistokehityksen näkökulmasta pelit vaihtelevat suuresti myös toiminnollisuuksiltaan. Tapauksen valintaan vaikutti pelin toteutuksen ja toimialan yksinkertaisuus, mutta samaan aikaan on haluttu toteuttaa sellainen peli, jonka toteutuksessa DCI näkyisi. DCI on korkean tason arkkitehtuuri, joten väistämättä yksinkertaisessa ohjelmassa se on ylilyönti. Tässä esimerkissä on kuitenkin tarkoitus keskittyä sen tuomiin etuuksiin ylläpidollisessa mielessä ja vähemmän koodirivien määrään tai toiminnalliseen nopeuteen.

Tapauksen peli on yksinkertainen yhdistetty graafi, jossa on solmuja ja särmiä. Solmut sisältävät jonkin käsitteen, kuten nimen tai asian, ja solmut ovat yhdistetty toisiinsa särmillä. Särmä kahden solmun väillä viittaa relaatioon solmujen käsitteiden välillä, esimerkiksi solmu ”Harry Potter” ja ”Daniel Radcliffe” voisivat olla yhdistettynä, sillä Radcliffe näytteli kyseistä hahmoa Harry Potter -elokuvissa. Toisaalta ”Harry Potter” ja ”Kirja” voisi olla yhdistettynä, sillä Harry Potter on alun perin kirjasarja. Kahden solmun välillä voi olla vain yksin särmä, mutta tietty solmu voi olla yhdistettynä useampaan muuhun solmuun. Peliä pelataan siten, että aluksi graafista näkyvät vain aloitussolmut ja niiden välittömät naapurit särmien suhteen siten, että naapureiden solmuista näkyy vain tieto, kuinka monta kirjainta kyseisessä käsitteessä on. Pelaajan on tarkoitus arvata kirjoittamalla peliin käsitteitä. Mikäli pelaajan arvaama käsite löytyy jonkin tunnetun sanan naapurista, kyseinen solmu paljastetaan ja sen muita, aiemmin piilossa olleita, naapureita voi arvata. Pelin idean on alun perin keksinyt Igor Navernіouk ja innoittajana on ollut IKI Ry:n sivulta löytynyt relaatiopeli [Hytönen, 2015].

Tähän tutkielmaan on valittu kyseinen peli, koska sen toimialalogiikka on yksinkertainen graafi, jonka toteutus voidaan tehdä monella tavalla. Koska tapauksessa on kiinnostuttu pelin toiminnollisuudesta ja ylläpidosta eikä toiminnallisesta nopeudesta, voidaan graafin toteutus tehdä helpolla tavalla siten, että data-puolen tutkimiseen ei mene paljon työtä. Tapaus ei ole siis optimoitu graafin suhteen, mutta käsittelemällä sitä rajapintojen kautta ei optimointi toisaalta ole myöskään estetty.

Seuraavaksi käydään läpi pelin kehitysympäristö, tarkemmat vaatimukset ja toiminnot. Näiden jälkeen esitellään pelin DCI-arkkitehtuuri. Lopuksi luvussa 5 käydään läpi pelin DCI-arkkitehtuurin arviointi. Pelin lähdekoodi on julkaistu GitHubissa. [GitHub, 2015]

4.2 Ohjelmointiympäristö ja tapauksen vaatimukset

Pelin ohjelmointiympäristönä on käytetty Unity [2015] alustaa, joka on pelien kehittämiseen tarkoitettu monipuolinen ohjelmointialusta. Unity mahdollistaa ohjelmoinnin C#:lla tai muokatulla JavaScriptillä. Tähän tutkielmaan on valittu C# tutkielman tekijän kokemuksen johdosta. C# mahdollistaa DCI-arkkitehtuurin toteutuksen tarpeeksi hyvin tapauksen kannalta, vaikka se ei ole siihen optimoitu. Peli on tehty Windowsille ja sen käyttö on testattu näppäimistöllä ja hiirellä.

Peli on pyritty pitämään yksinkertaisena, joten peliin on jätetty vain puhtaasti pelaamisen mahdollistavat toiminnollisuudet. Pelin pelaamisen toiminnolliset vaatimukset ovat seuraavat: pelin lataaminen, pelin tallentaminen, solmujen arvaaminen ja pelialueella liikkuminen. Näistä vaatimuksista on luotu taulukoissa 8-13 esitetyt käyttötapaukset.

Kuvaus		
Käyttötapaus	Pelin aloittaminen.	
Selostus	Pelaaja käynnistää pelin, jolloin kenttä ladataan näytölle valmiina pelattavaksi.	
Toimijat	Pelaaja, käyttöjärjestelmä ja pelin tiedostot.	
Alkutilanne	Pelin tiedostot on asennettuna pelaajan käyttöjärjestelmään, pelillä on oikeudet lukea ja muokata pelin tiedostoja.	
Jälkitilanne	Peli on pelattavissa näytöllä.	
Skenaario		
1.	Pelaaja käynnistää pelin.	
2.	Pelijärjestelmä lataa pelin kentän tiedoston.	
	2. a	Peli ladataan alkuperäisestä kenttätiedostosta.
	2. b	Peli ladataan viimeksi tallennetusta kenttätiedostosta.
3.	Kenttä näytetään pelaajalle.	
4.	Peli on pelattavassa tilassa.	
Huomioita		
2. Mikäli pelistä on tallennettu kenttätiedosto, niin pelin tulee ladata se, eikä alkuperäistä kenttätiedostoa.		

Taulukko 8: Aloittamisen käyttötapaus.

Kuvaus	
Käyttötapaus	Pelin tallentaminen.
Selostus	Pelin voi tallentaa missä tahansa vaiheessa pelin aikana, kun pelaajalla on mahdollisuus antaa komentoja.
Toimijat	Pelaaja, käyttöjärjestelmä ja peli.
Alkutilanne	Peli on ladattuna näytölle pelattavassa muodossa, peli odottaa käyttäjän komentoja, pelillä on oikeudet muokata sen tiedostoja.
Jälkitilanne	Pelin tilanne on tallennettuna tiedostoon, josta peli voidaan palauttaa täysin samaan pelitilanteeseen kuin tallennuksessa.
Skenaario	
1.	Käyttäjää antaa tallennuskomennon.
2.	Peli tallentaa pelitilanteen tiedostoon.
3.	Peli ilmoittaa tallentumisen lopputuloksesta pelaajalle.
4.	Peli odottaa uusia komentoja.
Huomioita	
3.a Mikäli tallentaminen onnistuu, peli ilmoittaa pelin tallennuspaikan. 3.b Mikäli tallentaminen ei onnistunut, peli pyrkii ilmoittamaan syyn, miksi sitä ei voitu tallentaa.	

Taulukko 9: Pelin tallentamisen käyttötapaus.

Kuvaus		
Käyttötapaus	Arvaaminen.	
Selostus	Pelaaja antaa pelille arvauksen, eli tekstin, jota peli vertaa arvattaviin konsepteihin. Mikäli arvaus on oikea, peli näyttää solmun tekstin ja mahdollistaa sen naapurien arvaamisen, muulloin peli ilmoittaa virheellisestä arvauksesta.	
Toimijat	Pelaaja, peli.	
Alkutilanne	Peli on ladattuna näytölle, peli odottaa pelaajan komentoja.	
Jälkitilanne	Peli on reagoinut tarpeellisella tavalla ja odottaa pelaajan seuraavaa komentoa.	
Skenaario		
1.	Pelaaja kirjoittaa tekstin tekstikenttään ja komentaa peliä tekemään arvauksen tekstillä.	
2.	Peli tarkistaa arvattavissa olevista solmuista, täsmääkö teksti solmuun asetetun tekstin kanssa.	
3.	a	Mikäli teksti löytyy jostain kyseisistä solmuista, solmun teksti näytetään kokonaan ja solmun piilossa olleet naapurit näytetään siten, että niiden teksteistä näkyy vain kirjainten ja sanojen määrä.

	b	Mikäli ei löydy kyseistä tekstiä, peli ilmoittaa siitä pelaajalle.
4.	Peli jää odottamaan uusia komentoja.	
Huomioita		
2. Vain niiden solmujen tekstejä voidaan arvata, joka on suoraan jonkin arvatun solmun naapuri tai pelin alussa näytetyn solmun naapuri.		

Taulukko 10: Arvaamisen käyttötapaus.

Kuvaus	
Käyttötapaus	Pelialueella liikkuminen.
Selostus	Pelin kenttä voi olla suurempi kuin näytölle mahtuva alue, jotta se olisi miellyttävän näköinen. Pelaaja liikuttaa pelin ruutua siten, että ruudun ulkopuolella olevat solmut tulevat näkyviin.
Toimijat	Pelaaja, peli.
Alkutilanne	Peli odottaa pelaajan komentoja ja pelin kenttä on suurempi, mitä mahtuu näytölle.
Jälkitilanne	Peli näyttää sen kohdan kentästä, minne pelaaja on näkymää siirtänyt.

Skenaario	
1.	Pelaaja valitsee suunnan ja nopeuden mihin sekä kuinka nopeasti näkymää siirretään.
3.	Pelin näkymä siirtyy kyseiseen suuntaan.
4.	Peli odottaa käyttäjän komentoja.

Huomioita
1. Pelaaja tekee tämän hiirellä painamalla ja pitämällä hiiren nappia painettuna pelikentän solmuttomalle alueelle sekä liikuttamalla hiirtä. Pelin näkymä siirtyy pelikentällä vastakkaiseen suuntaan kuin hiiren liike. Siirtyminen tapahtuu suhteessa hiiren liikenopeuteen.

Taulukko 11: Pelialueella liikkumisen käyttötapaus.

Kuvaus	
Käyttötapaus	Pelin lopettaminen.
Selostus	Peli suljetaan.
Toimijat	Pelaaja, peli.
Alkutilanne	Peli odottaa käyttäjän komentoja.
Jälkitilanne	Peli on suljettuna.

Skenaario	
1.	Käyttäjä antaa pelin sulkemiskomennon.

2.	a	Mikäli peliä ei ole tallennettu viimeisen muutoksen jälkeen, peli kysyy, halutaanko peli sulkea tallentamatta.
	b	Siirrytään kohtaan 3.
3.	Peli sulkeutuu.	
Huomioita		
2.a Mikäli suljetaan tallentamatta, pelin sulkeminen jatkuu kohdan 2.b mukaan. 2.a Mikäli ei suljeta, sulkeminen keskeytetään ja peli jatkaa uusien komentojen odottamista.		

Taulukko 12: Pelin lopettamisen käyttötapaus.

Kuvaus		
Käyttötapaus	Pelisilmukka.	
Selostus	Pelin silmukka suoritetaan ajoitetusti noin 30 kertaa sekunnissa.	
Toimijat	Peli, pelaaja.	
Alkutilanne	Peli on käynnistetty.	
Jälkitilanne	Peli on mahdollisesti reagoinut käyttäjän toimintaan.	
Skenaario		
1.	Päivitä pelin ajastimet.	
2.	Tarkista missä toiminnassa peli on:	
	a.	Pelaaja on arvaamassa -> päivitä komennot ja siirry pelin arvaamisen käyttötapaukseen (taulukko 10).
	b.	Pelaaja liikuttaa kameraa -> päivitä komennot ja siirry pelikameran liikuttamisen käyttötapaukseen (taulukko 11).
	c.	Pelaaja on lopettamassa peliä -> päivitä komennot ja siirry pelin lopettamisen käyttötapaukseen (taulukko 12).
	d.	Peli halutaan tallentaa -> aloita pelin tallentamisen käyttötapaukseen (taulukko 9).
	e.	Pelaaja ei antanut komentoja.
3.	Pelin toiminta siirretään UnityEnginelle, joka tekee omat päivityksensä kyseiselle ruutupäivitykselle.	
Huomioita		
3. Kun UnityEngine on lopettanut kyseisen ruutupäivityksen, se kutsuu uuden ruutupäivityksen aloitusta, ja peli silmukkaa kutsutaan jälleen managerin kautta alkamaan alusta.		

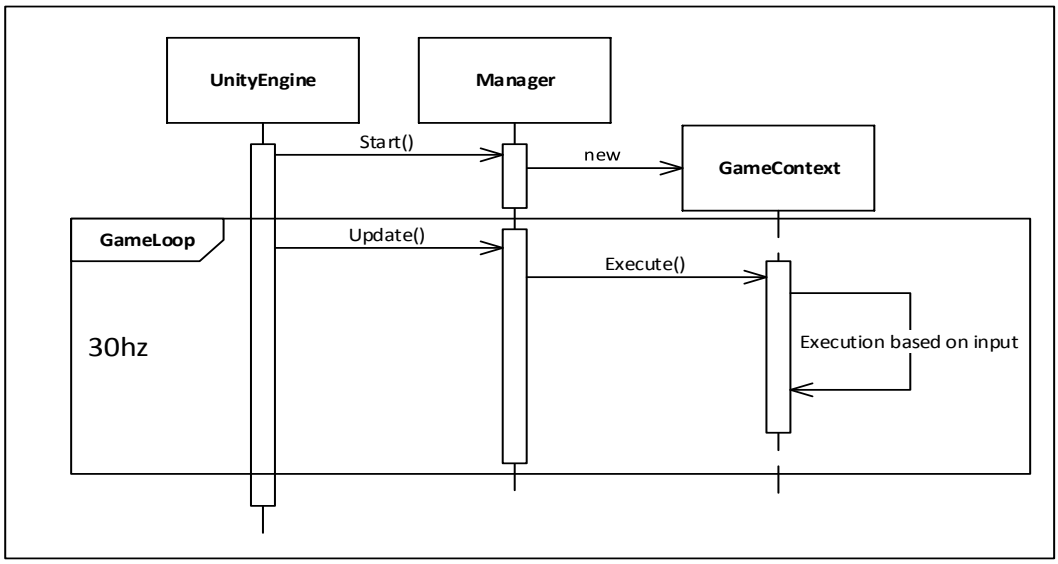
Taulukko 13: Pelisilmukan käyttötapaus.

DCI-arkkitehtuurin muodostaminen jatkuu käyttötapauksien ja toimialalogiikan tarkemmalla analysoinnilla. Toimialasta tulee karkeasti data, käyttötapauksien

skenaarioista kontekstit ja käyttötapauksien perusteella luodaan roolit. Pelin käyttötapauksien ja toimialalogiikan analysoiminen oli kuitenkin pitkä ja iteratiivinen prosessi, joten seuraavassa kohdassa esitellään vain pelin lopulliset versiot DCI-arkkitehtuurin eri osioista.

4.3 Tapauksen arkkitehtuuri ja DCI

UnityEnginen toiminta perustuu pääosin peliolioihin ja niihin liitettäviin komponentteihin. Unityn pelioliot ovat GameObject-luokasta perittyjen luokkien oliota ja komponentit ovat Component-luokasta perittyjä olioita. Unityn komponentteja ovat muun muassa erilaiset grafiikkaan liittyvät komponentit, kuten 3d-malli ja niiden renderoijat. Komponenteista löytyy MonoBehaviour-luokka, jonka avulla peliolioille voidaan asettaa ajettavaa koodia. Tapauksen kohdalla pelissä on pääosin yksi luokka, Manager, joka on peritty MonoBehaviour-luokasta. Unity kutsuu jokaisen MonoBehaviour-luokan Start-metodia yhden kerran kun kyseisen luokan olio on asetettu päälle. Manager-luokan Start-metodissa luodaan GameContext-olio, joka sisältää käytännön pelilogiikan. Unity kutsuu 30 kertaa sekunnissa jokaisen päällä olevan MonoBehaviourista perityn luokan Update-metodia. Managerin Update-metodissa kutsutaan GameContextin Execute-metodia, ja näin pelin ajo on saatu siirrettyä DCI arkkitehtuuriin. Kuvasta 6 näkyy kyseisen toiminnan sekvenssi. [Unity, 2015]

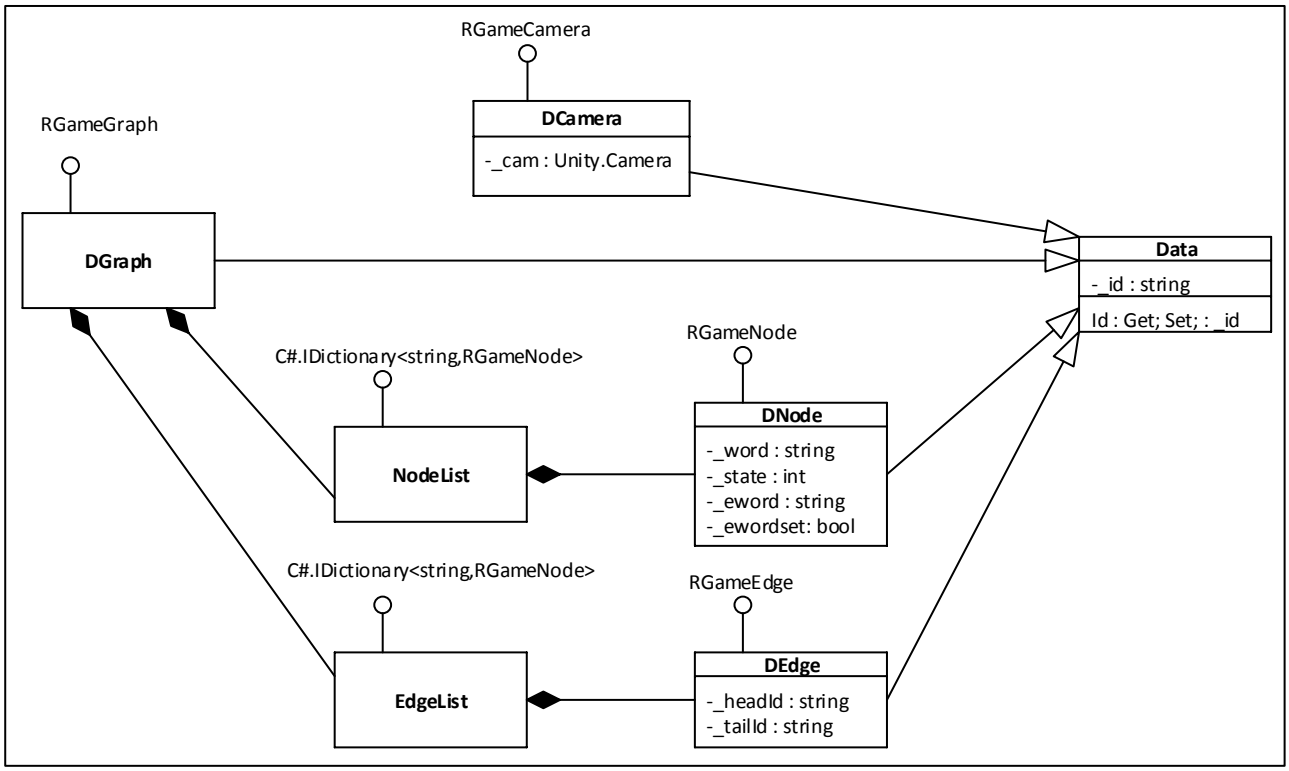


Kuva 6: Ohjelman pelisilmukan ajaminen.

4.3.1 Data

Tapauksen peli, toimialalogiikan kannalta, on graafi. Graafi koostuu solmuista ja niiden välisistä särmistä. Tapauksen graafi on suuntaamaton ja painottamaton. Pelin lopulliset data-luokat on kuvattu luokkakaaviona (kuva 7). Data-luokka toimii varsinaisten DCI-arkkitehtuurin data-olioiden pääluokkana ja määrittää siten jokaiselle Datasta perityille

luokalle tunnisteen (Id). DNode ja DEdge kuvaavat pelin solmut ja särmät. Solmuluokassa on attribuutit _word ja _state, jotka pitävät sisällään kyseiseen solmuun asetetun arvattavan sanan sekä tiedon, onko kyseinen solmu arvattu, arvattavissa vai piilossa. Attribuutti _eword luodaan dynaamisesti _word-attribuutista. Särmäluokassa on tiedot sen alku- ja loppusolmun tunnisteista. NodeList ja EdgeList ovat C#:n Dictionary-luokasta erikoistettuja luokkia, joiden tehtävänä on puhtaasti pitää sisällä graafin solmut ja särmät. DGraph edustaa graafia ja siihen kuuluu lista sen solmuista ja särmistä. DCamera pitää sisällään viitteen UnityEnginen kameraan.



Kuva 7: Pelin data-luokat.

NodeList ja EdgeList sisältävät metodeja jotka liittyvät vain säiliönä toimimiseen, mutta muuten data-osion luokat eivät pidä sisällään pelintoimintaan liittyvää logiikkaa, kuten DCI-arkkitehtuurissa on tarkoitus. DEdge, DNode ja DGraph toteuttavat roolien RGameEdge, RGameNode ja RGameGraph tunnisterajapinnat. Tunnisterajapinnoilla kerrotaan, mitä rooleja kyseiset data-oliot voivat näytellä. Oleellisesti rajapinnoilla varmistetaan se, että oliolla on ne tiedot, joita roolin suoritusta varten vaaditaan. Rajapinta vaatii tiettyjen metodien toteutusta, mutta käytännössä ne ovat vain käsittelijä-tyyppisiä ohjausmetodeja, kuten näkyy koodikatkelmista 4 ja 5.

```
using UnityEngine;
using System.Collections;

public class DGraph : Data, RGameGraph
{
	GameNodeList _nodes;
	GameEdgeList _edges;

	public DGraph()
	{
		_nodes = new GameNodeList();
		_edges = new GameEdgeList();
	}
	#region RGameGraph implementation
	public GameNodeList Nodes {
		get {return _nodes;}
		set {_nodes = value;}
	}
	public GameEdgeList Edges {
		get {return _edges;}
		set {_edges = value;}
	}
	public RGameNode NewNode {
		get {return new DNode();}
	}

	public RGameEdge NewEdge {
		get {return new DEdge();}
	}
	#endregion
}
```

Koodikatkelma 4: Graafin dataluokka.

```
public class DNode : Data, RGameNode
{
	string _word;
	int _state;
	string _eword;
	bool _eSet = false;

	#region RGameNode implementation
	public string Word {
		get {return _word;}
		set {_word = value;}
	}
	public int State {
		get {return _state;}
		set {_state = value;}
	}
	public string EWord {
		get {return _eword;}
		set{
			if(!_eSet){
				_eword = value;
				_eSet = true;
			}
		}
	}
	public bool ESet {
		get{return _eSet;}
	}
	#endregion
}
```

Koodikatkelma 5: Solmun dataluokka.

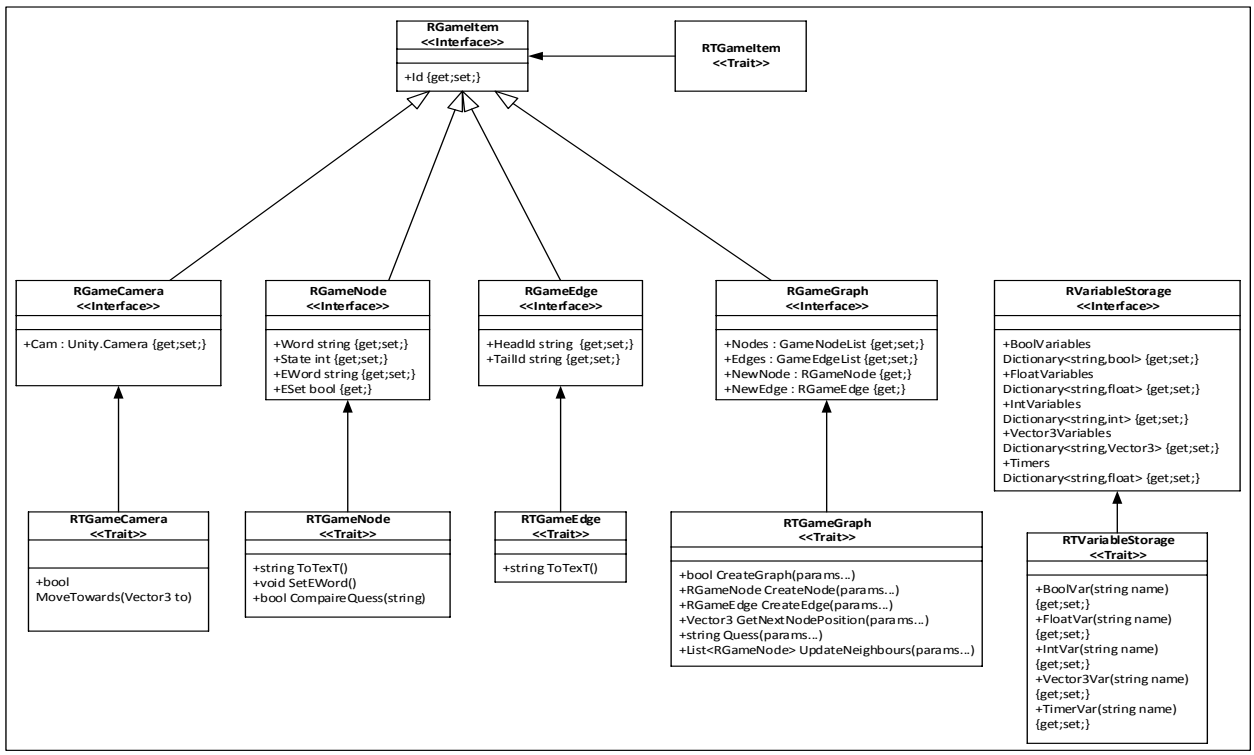
4.3.2 Roolit

Tapauksen peli on kirjoitettu C#:lla, eikä se tue toteutusta sisältäviä rajapintoja. Tästä johtuen roolit luodaan tunnisterajapinta ja roolitoteutus parina. Roolitoteutus on C#:lla toteutettu Extension method -tekniikalla, joka tässä tutkielmassa oletetaan tunnetuksi. Tunnisteet kertovat, mitä roolia data-olio voi esittää, kun taas toteutusroolit sisältävät roolin varsinaiset toiminnot. Teknillisesti rajapinta varmistaa sen, että sitä roolia näyttelevältä oliolta saadaan roolin toiminnoissa vaadittavat tiedot, joita käytetään toteutusroolin metodeissa. [Coplien & Bjørnvig, 2010]

Tapauksen pelistä on löydetty seuraavat roolit: RGameNode, RGameEdge, RGameGraph, RGameCamera ja RVariableStorage (kuva 8). Näistä rooleista RGameNode, RGameEdge, RGameGraph ja RGameCamera ovat pelistä löytyviä olioita, jotka perivät RGameItemin roolin. Jokaista roolia kohden löytyy sen tunniste, joka alkaa kirjaimella R, sekä siihen

kuuluva Extension Method -tiedosto, joka alkaa kirjaimilla RT. Peli on suhteellisen yksinkertainen ja varsinaista toimintalogiikkaa on niukasti. Toiminnollisuuden vähyys näkyy roolien toimintojen vähyydessä ja data-osion samankaltaisuudessa verrattaessa rooleihin.

Roolit RGameNode, RGameGraph ja RGameEdge toimivat pelilogiikan määräämällä tavalla graafin osien rooleissa. RGameGraph toimintoihin kuuluu graafin luominen tekstistä, graafin solmujen ja särmien hallinnoiminen sekä pelin arvaustoiminto. Arvaustoiminto käy läpi mahdolliset solmut ja kysyy solmuroolilta, onko arvattu sana oikein. RGameNode sisältää logiikan arvauksen ja solmussa olevan sanan vertaamiseen. Mikäli vertailualgoritmia halutaan muokata, esimerkiksi tehdä täsmällisen vertaamisen sijaan monipuolisempi edit-distance versio, se voidaan tehdä suoraan muokkaamalla vain solmuroolin roolitoteutusta. Mikäli muutoksia halutaan tehdä solmujen läpikäynti algoritmiin, se tehdään suoraan muokkaamalla graafiroolin roolitoteutusta.

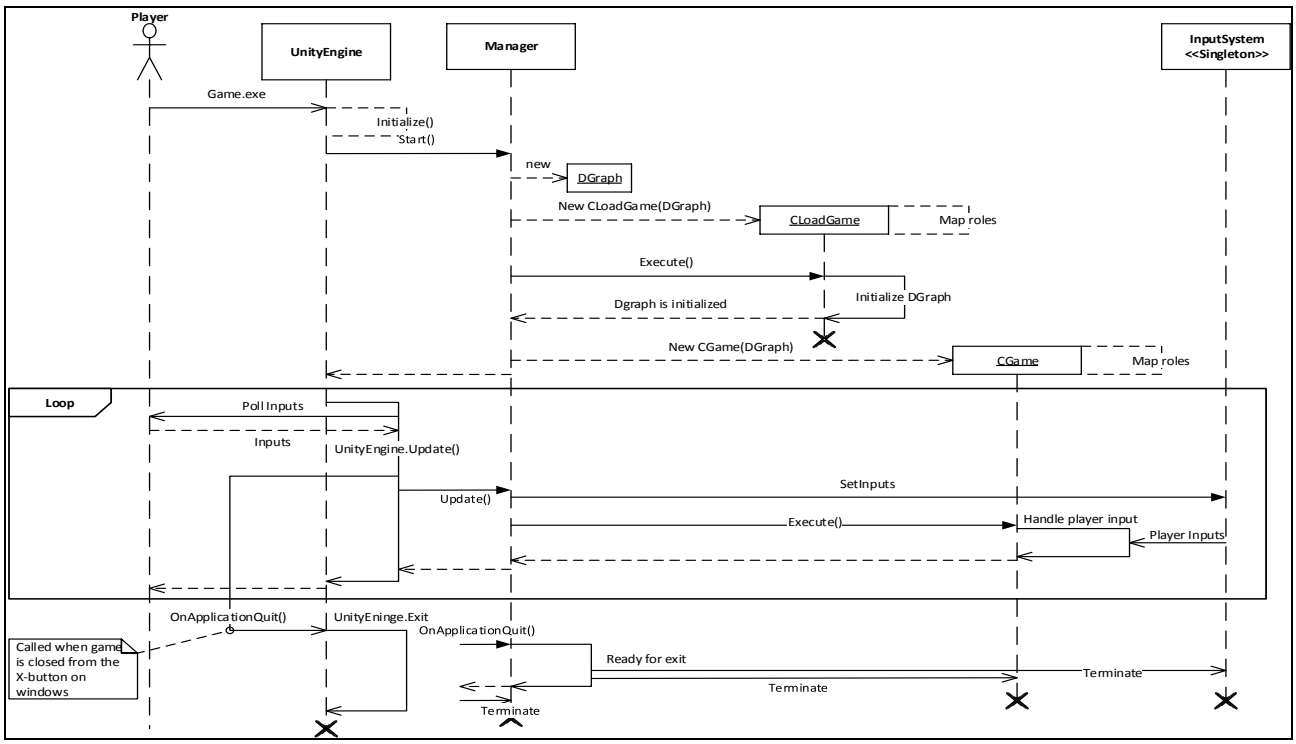


Kuva 8: Pelin roolit.

4.3.3 Kontekstit ja vuorovaikutus

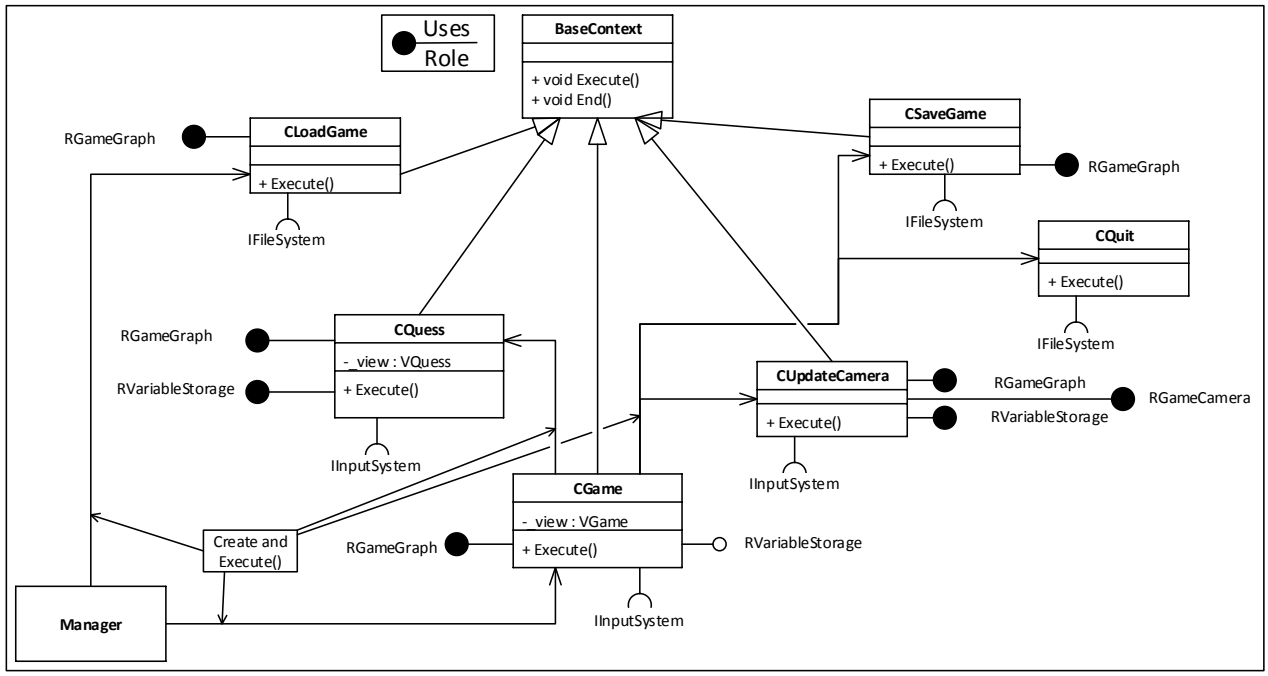
Tapaukseen on määritetty aikaisemmin muutamia käyttötapauksia ja niistä on luotu DCI-arkkitehtuurin mukaiset kontekstiluokat. Kontekstit koostuvat kahdesta osasta: rooleja näyttelevien olioiden ohjaamisesta kontekstissa käytettäviin rooleihin sekä varsinaisesta käyttötapausalgoritmista. Tapauksen C#-toteutuksessa roolien ohjaaminen tapahtuu suoraan kontekstin rakentajassa ja algoritmi metodissa Execute.

Pelissä on kuusi käyttötapausta, josta jokaisesta on tullut suoraan oma kontekstinsa lukuun ottamatta pelin aloitusta. Tapauksen kontekstit ovat BaseContext, CGame, CLoadGame, CSaveGame, CQuess, CQuit sekä CUpdateCamera. BaseContext on yhdistävä yliluokka puhtaasti organisointia varten, eikä sisällä pelin varsinaiseen toimintaan liittyviä toimintoja. CGame pitää sisällään pelin kulun, jonka sisällä kutsutaan muita kontekseja lukuun ottamatta CLoadGame-kontekstia. Pelin aloittamisen käyttötapaus ei ole konteksti, vaan se toteutuu Manager-luokan, CLoadGame- ja CGame-kontekstien avulla. Manager-luokan Start-metodissa kutsutaan CLoadGamea, joka lukee tapauksen pelin tekstitiedostomuodosta RGameGraphia näyttelevään DataGraphiin. Manager määrää tämän jälkeen kyseisen graafin CGamekontekstiin ja kutsuu kyseisen kontekstin Execute-metodia Managerin Update-metodissa. Managerin toiminta näkyy kuvassa 9. CSaveGame tallentaa RGameGraphin tekstiedostoksi ja CQuit-kontekstia kutsutaan pelin lopetuksessa. CQuess pitää sisällään arvaamisen ja sitä kutsutaan, kun pelaaja aktivoi arvaamistekstikentän. RUpdateCamera määrää pelikentällä liikkumisen ja aktivoituu, kun pelaaja tekee hiirellä raahauseleen pelikentässä.

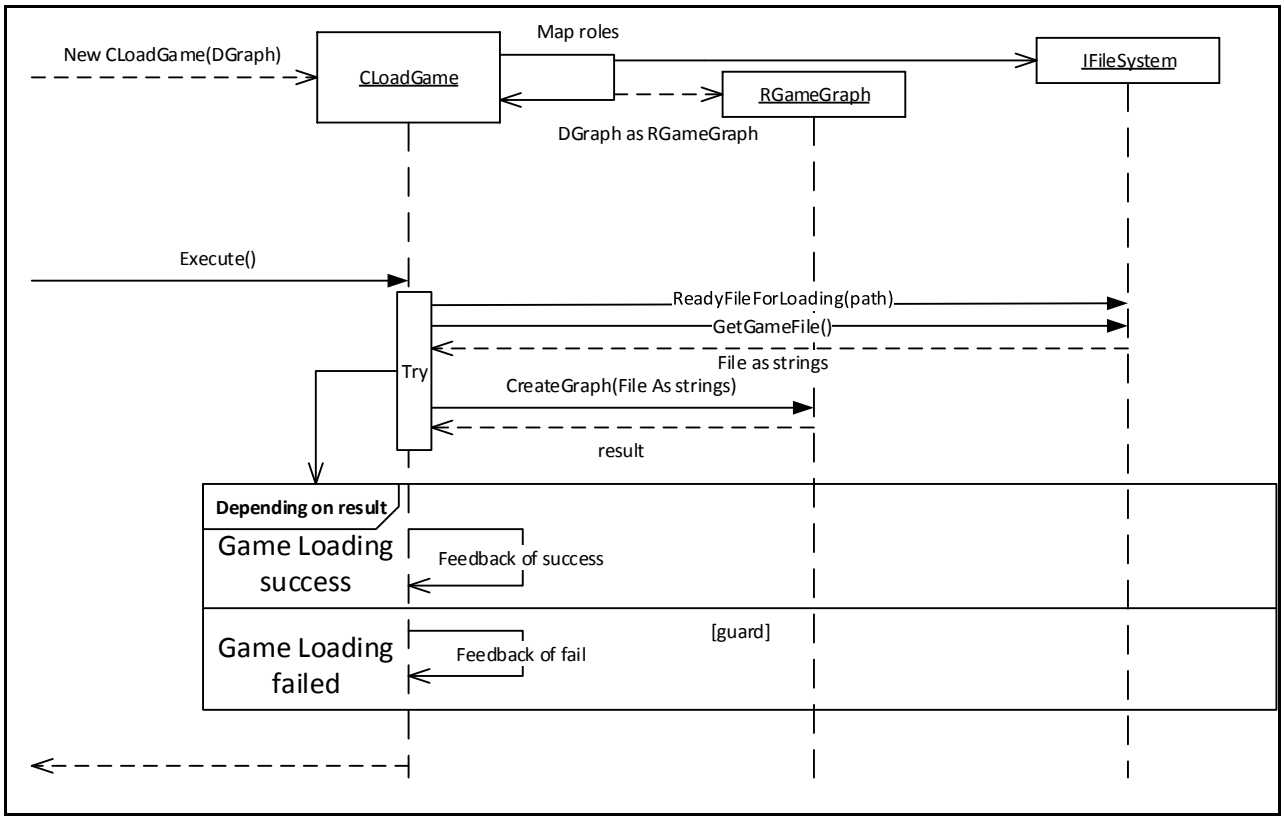


Kuva 9: Managerin toiminnan sekvenssikaavio.

DCI-arkkitehtuurissa kontekstit määrittävät ja ohjaavat roolien välistä vuorovaikutusta. Kontekstien algoritmien tulisi pysyä käyttötapauksien tasolla, vaikkakin hieman teknisemmältä näkökannalta. Varsinainen toimintalogiikka on sijoitettuna rooleihin, jotka kuvattiin alakohdassa 4.3.2. Kuvassa 10 kuvataan kontekstien riippuvuudet rooleihin ja muihin konteksteihin.



Kuva 10: Kontekstit ja roolirajapinnat.



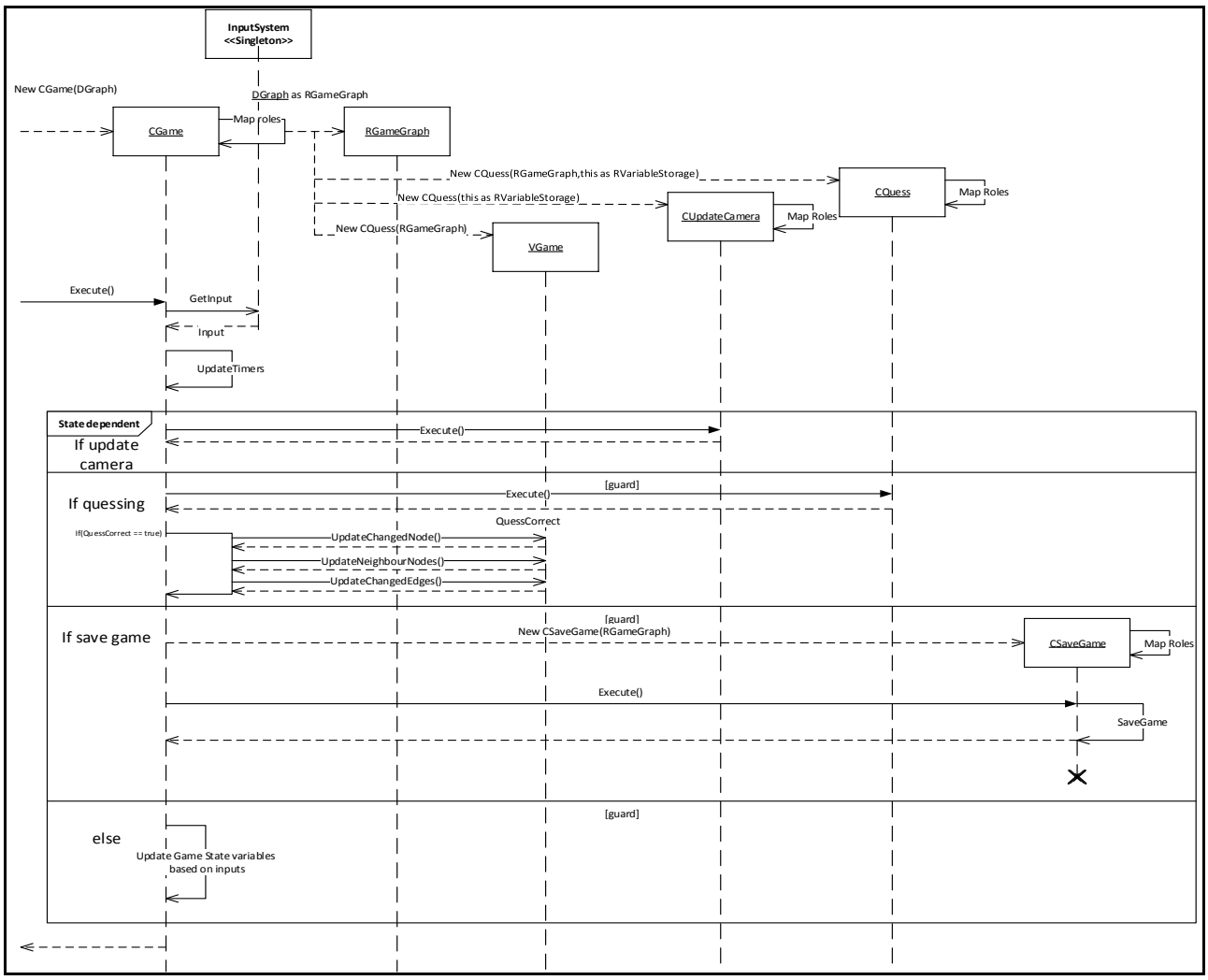
Kuva 11: Kontekstin CLoadGame sekvenssikaavio.

CLoadGame-konteksti (kuva 11) ohjaa IFileSystem-rajapintaa sekä RGameGraph-roolia. CLoadGame-kontekstin rakentajassa ohjataan siihen syötetty olio RGameGraph-rooliin. Tapauksessa rakentajaan syötetään Manager-luokassa alustettu DGraph. Konteksti asettaa

käyttövalmiuteen IFileSystem-rajapinnan, joka käytännössä annetaan myös sen rakentajassa. Koska toteutuksia on useita, eikä IFileSystemin teknillisestä ratkaisusta ole kiinnostuttu tutkielman aihealueessa, se on jätetty pois kuvaajista.

CLoadGame-kontekstin varsinaisessa algoritmissa, eli Execute-metodissa, ladataan tiedosto IFileSystemin palveluilla ja ohjataan String-muotoinen tiedosto RGameGraph-roolin CreateGraph-metodiin. CreateGraph-metodi tallentaa tiedoston mukaisen graafin sitä näyttelevään data-olioon, ja ilmoittaa boolean-arvolla graafin luonnin onnistumisesta. Pelissä ilmoitus tapahtuneesta ohjataan konsoliin.

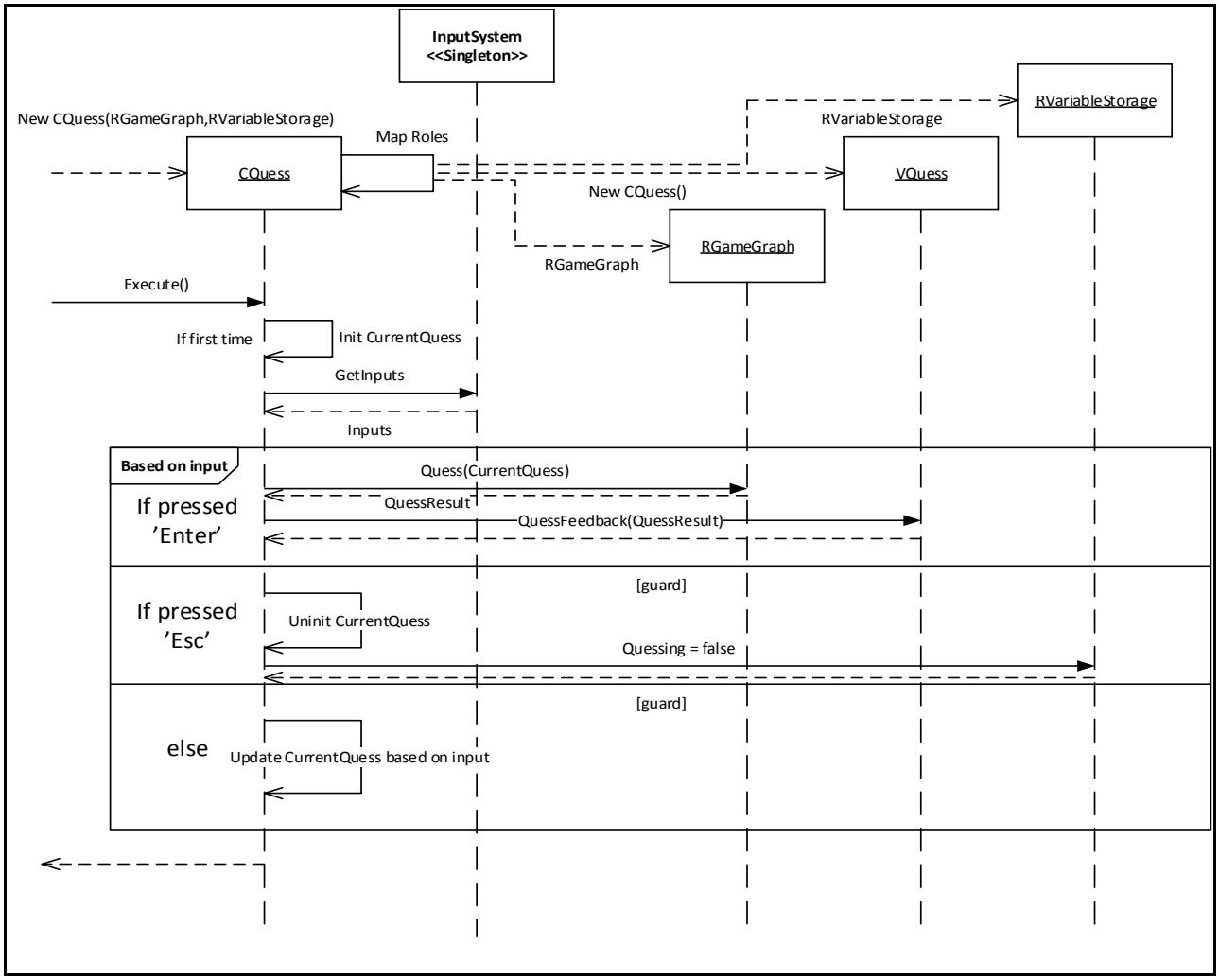
CLoadGame-konteksti alustettiin ja kutsuttiin Managerin Start-metodissa. CLoadGame-konteksti on puhtaasti DCI-arkkitehtuurin mukainen konteksti. Execute-metodinsa jälkeen CLoadGame-konteksti tuhoutuu, jolloin jäljelle jää tekstitiedoston mukainen graafi kontekstille asetetussa DGraphissa. DGraphin syötetään Manager-luokassa CGame-kontekstiin, jonka toiminta näkyy kuvasta 12.



Kuva 12: Kontekstin CGame sekvenssikaavio.

CGame pitää sisällään pelin yleisen kulun ja on toteutettu pelisilmukan käyttötapauksen mukaan (taulukko 13). CGame-kontekstin rakentajalle annetaan CLoadGame-kontekstin täyttämä DGraph, joka asetetaan jälleen näyttelemään roolia RGameGraph. Tapauksen kontekstissa CGame alustetaan valmiuteen kontekstit CUpdateCamera ja CQuess. Toisin kuin puhtaammin DCI-arkkitehtuurin mukainen CLoadGame-konteksti, joka tuhotaan sen Execute-metodin jälkeen, CGame-, CUpdateCamera- ja CQuess-konteksteja tullaan kutsumaan useamman kerran, mistä syystä niitä ei tuhota kutsujen välissä. Toinen merkittävä ero CLoadGame-kontekstin toimintaan verratuna on se, että CGame pitää sisällään näkymän. Näkymiä (view) on sijoitettu muutamaan kontekstiin, ja niitä käydään tarkemmin läpi kohdassa 4.4. Kolmantena erona muihin konteksteihin on, että CGame toimii roolissa RVariableStorage. RVariableStorage-roolia näyttelevä olio toimii muuttujavarastona, joka tässä tapauksessa tarkoittaa pelin toiminnallista tilaa, esimerkiksi ajastimia. Pelin toiminnallista tilaa ei tule sekoittaa pelin toimialalogiikan tilaan eli DCI-arkkitehtuurin data-osioon. CGame-kontekstin toiminta vaatii reagoimista käyttäjän toimintaan, ja pyytää käyttäjän antamia syötteitä InputSystem-singletonilta. InputSystemin toiminta ei varsinaisesti ole tutkielman aiheen kannalta merkittävää, joten sen tarkempaa toimintaa ei tulla käsittelemään.

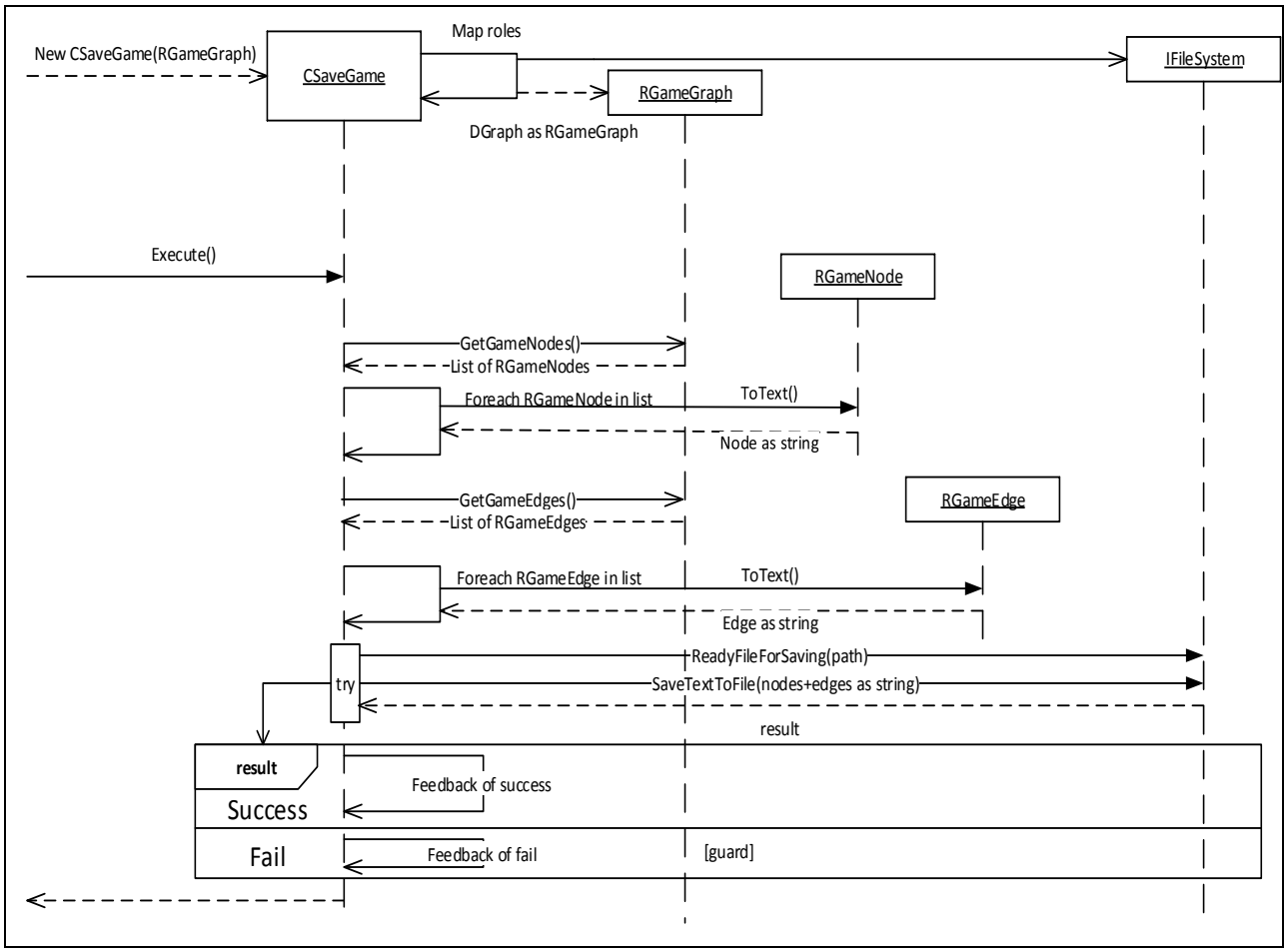
CGame-kontekstin Execute-metodia kutsutaan Manager-luokan Update-metodista (kuvat 5 ja 8). Execute-metodin toiminta alkaa käyttäjän syötteiden hakemisella sekä pelin sisäisten ajastimien päivittämisellä. Tämän jälkeen, riippuen pelin toiminnallisesta tilasta, peliä jatketaan käyttäjän syötteiden tutkimisella tai kutsumalla CUpdateCamera-, CQuess- tai CSaveGame-kontekstin Execute-metodia. Varsinaisessa toteutuksessa on muutama ylimääräinen vaihtoehto, mutta ne liittyvät pelin toimimisen sulavuuteen ja käytännössä vain tarkistavat, tehdäänkö ylipäätään mitään kyseisellä CGame.Execute-metodin kutsuntakerralla. Seuraavaksi käydään läpi muut kontekstit, joiden toiminnan runko seuraa merkittävästi joko CGamen tai CLoadGamen runkoa.



Kuva 13: Kontekstin CQuess sekvenssikaavio.

CQuess-kontekstin (kuva 13) toiminta alkaa, kun pelaaja aktivoi arvaamisen tekstikentän. CQuess aktivoituu, kun pelaaja painaa hiirellä arvaustekstikenttää. CQuess-kontekstin rakentajassa on CGame-kontekstin antama RGameGraph-olio, joka siis oli alun perin DGraph. Lisäksi rakentajassa CGame asetetaan RGameGraph- ja RVariableStorage-rooleihin. CQuess alustaa myös näkymän, joka on käytännössä arvaustekstikenttä.

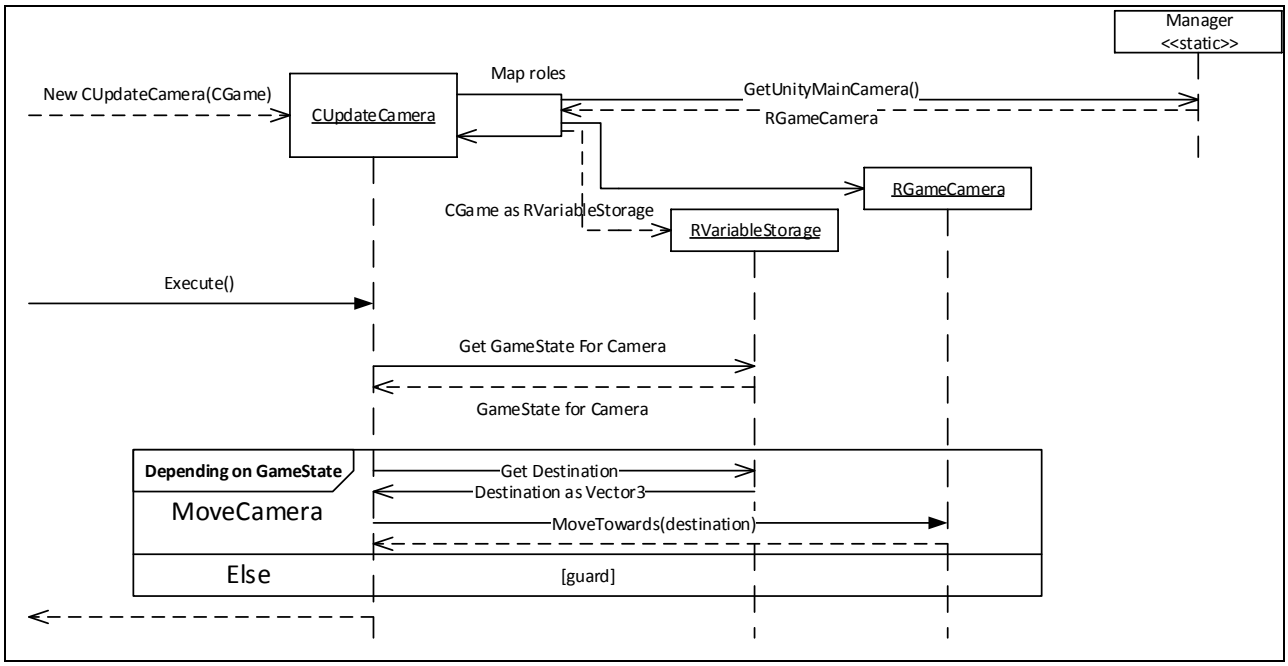
CQuess pyörittää tekstin lukemiseen vaadittavaa silmukkaa sekä siihen liittyvää muuta toimintaa Execute-metodissa. Execute-metodin toiminta alkaa uuden arvauksen alustamisella ja syötteiden hakemisella SystemInputs-singletonilta. Peli reagoi tässä tapauksessa vain näppäimistön komentoihin ja pyrkii tulkitsemaan kaikki näppäinkomennot tekstinä lukuun ottamatta Enter- ja Esc-näppäimiä. Mikäli CQuess tulkitsee pelaajan antaneen Enter-komennon, se kutsuu RGameGraph-olion Quess-metodia annetulla tekstillä. CQuess ohjaa näkymän näyttämään palautetta arvauksen tuloksesta. Jos CQuess tulkitsee pelaajan antaneen Esc-komennon, se lopettaa arvaamisen ja asettaa RVariableStoragen muuttujat siten, että toiminta ohjataan takaisin CGamelle.



Kuva 14: Kontekstin CSaveGame sekvenssikaavio.

CSaveGame-konteksti (kuva 14) on pelin puhtain DCI-arkkitehtuurin mukainen konteksti. CSaveGame alustetaan, kun CGame on saanut komennon tallentaa peli, ja tuhoutuu, kun peli on tallennettu. CSaveGamen rakentaja saa CGamelta jälleen aluperin DGraph-tyyppisen RGameGraphin ja hyödyntää IFileSystemiä.

CSaveGame-kontekstin Execute-metodi alkaa pyytämällä RGameGraphilta listan sen RGameNodeista ja käy läpi kyseisen listan kutsumalla niiden RGameNodesta löytyvää ToText-metodia. ToText-metodi pitää sisällään logiikan siitä, miten kyseinen RGameNode (ja RGameEdge) näytetään tekstirivinä. CSaveGame tallentaa jokaisen solmun tiedot samaan String-muuttujaan. Solmujen jälkeen sama tehdään särmille, eli RGameEdgelle. Lopuksi kontekstissa kyseinen String-muuttuja annetaan IFileSystemille ja jälleen IFileSystemin palaute tallennuksen onnistumisesta ohjataan konsoliin.



Kuva 15: Kontekstin CUpdateCamera sekvenssikaavio.

CUpdateCamera-konteksti (kuva 15) alkaa, kun pelaaja tekee hiirellä raahauseleen pelikentällä ja pelaaja ei sillä hetkellä ole aktivoinut arvaamista. CUpdateCamera alustetaan CGamen alussa ja se hyödyntää kyseistä CGame-kontekstia RVariableStorage-roolissaan. CUpdateCamera pyytää Manager-luokalta pelin aktiivisen kameran ja asettaa sen RGameCamera-rooliin. Kamera on UnityEnginen luokka, ja sen tarkoitus on muuntaa 3d-maailma 2d-projektioksi näytölle.

CUpdateCamera-kontekstin Execute-metodi hakee hiiren sijainnin ja muiden kameran liikkeeseen liittyvien muuttujien tietoja RVariableStoragesta. Mikäli pelikentällä tulee liikkua, konteksti ohjaa tarvittavat tiedot RGameCamera-roolin MoveTowards-metodille. RGameCameran MoveTowards-metodi pitää sisällään kameran liikuttamisen toimintalogiikan. Pelin käytettävyyden kannalta kameran liikuttaminen voidaan toteuttaa myös merkittävästi dynaamisemmin ja monipuolisemmin. Mikäli kyseistä logiikkaa halutaan muuttaa, muutos voidaan tehdä RGameCameran roolitoteutukseen.

4.4 Pelin näkymät

DCI-arkkitehtuuri on tarkoitettu ohjaamaan järjestelmää korkealla tasolla. DCI ei ota kantaa matalan tason toimintojen toteutukseen, kuten käyttäjän syötteiden lukuun tai näkymiin [Reenskaug & Coplien, 2009; Coplien & Bjørnvig, 2010; Reenskaug, 2009]. Pelaajalle näytettävät tulosteet pelin tilasta ja pelaajan syötteet ovat kuitenkin oleellinen osa järjestelmää. Pelissä syötteiden lukeminen on ratkaistu singletonin avulla ja näkymät ovat luokkia, jotka on sijoitettu kontekstien yhteyteen.

Pelaajan syötteiden luku on toteutettu yksinkertaisella syötteiden lukupalvelulla, joka lukee UnityEnginen kaappaamat syötteet. Ne voidaan saada InputSystem-singletonilta. Syötteiden lukemista hyödynnetään useassa kontekstissa ja jokainen konteksti pyytää syötteet itse. Pelin näkymiä on konteksteissa CGame, CQuess sekä CQuit. CGamen näkymä pitää sisällään jokaista pelin solmua ja särmää vastaavat pelioliot. CQuess sisältää tekstikentän ja CQuit pelin lopettamisessa näytettävän dialogin. Kontekstien CGame, CQuess ja CQuit näkymistä vastaavat luokat VQuess, VGame ja VQuit. Näkymien koodeista löytyy palvelut puhtaasti näkymien luontiin, päivittämiseen ja tilan pyytämiseen. Näkymät on yhdistetty johonkin MonoBehaviour-luokasta perittyyn luokkaan, ja nämä luokat ovat liitettynä UnityEnginen pelikentän peliolioihin. CGamen VGame-näkymästä löytyy jokaiselle solmulle oma VNodeMono ja jokaiselle särmälle VEdgeMono, jotka ovat yhdistettynä Node- ja Edge-peliolioihin. CQuess-kontekstin näkymä VQuess on yhdistetty VPanelMonoon ja se on liitetty peliolioon TextPanel. VQuit-näkymää vastaava MonoBehaviour on VExitMono ja se on kiinnitetty ExitButton-peliolioon. Tapauksen syötteiden lukemisen ja näkymien toteutus ei ole ainoa mahdollinen ratkaisu, mutta se ei ainakaan riko DCI-arkkitehtuurin perusideaa erottaa järjestelmän tila toiminnoista, sillä näkymät ja syötteiden lukeminen on yhdistettynä vain konteksti-olioihin.

5 Arviointi

DCI-arkkitehtuuri on syntynyt leanin ja ketterien menetelmien yhdistämiseksi, ja sen tulisi tukea näiden arvoja. Ketterät menetelmät keskittyvät nopeaan sekä iteratiiviseen kehittämiseen, kun taas lean keskittyy tehostamaan järjestelmien kehitystä kokonaisuutena. Tapauksen soveltuvuutta lean-ohjelmistokehitykseen voidaan tutkia tarkastelemalla leanin perusperiaatteita. Leanin periaatteista tässä tutkielmassa keskitytään tarkastelemaan kolmea kohtaa: ylimääräistä koodia, arkkitehtuurin vaatimuksia sekä tuotearvoa. Ketterän kehityksen soveltuvuutta tarkastellaan järjestelmän ymmärrettävyyden näkökulmasta. Ymmärrettävyys on yhteydessä ylläpidettävyyteen [Boehm et al., 1976], joka on monesti vaatimus nopeaan ohjelmistokehitykseen. Ketterän ja leanin arvioinnin jälkeen tapaukseen esitellään muutos, jonka konkreettinen toteutus kuvataan kohdassa 5.3. Luvun lopuksi arvioinnista on yhteenveto kohdassa 5.4. Tapauksen arviointi keskittyy vain taulukossa 14 näkyviin luokkiin ja niiden ominaisuuksiin, sillä ne ovat DCI-arkkitehtuurin toteuttavat osiot ja koska muuten työmäärä laajentuisi hyötyihin nähden merkittävästi. Arvioinnin ulkopuolelle on jätetty Manager, GameNodeList, GameEdgeList sekä InputSystem ja FileSystem.

Data	**Contex**	**Roles**	**View**
Data	BaseContext	RGameCamera	View
DNode	CQuess	RGameEdge	VNodeMono
DGraph	CGame	RGameGraph	VExitButtonMono
DEdge	CLoadGame	RGameItem	VEdgeMono
DCamera	CQuit	RGameNode	VQuess
	CSaveGame	RTGameEdge	VCamera
	CUpdateCamera	RTGameGamera	VGame
		RTGameGraph	VPanelMono
		RTGameNode	
		RTVariableStorage	
		RVariableStorage	

Taulukko 14: Arvioinnin kohteet.

5.1 Lean

Coplien ja Bjørnvig [2010] kirjoittavat paljon leanin käyttämisestä ohjelmistokehityksessä. Suurin osa tästä toiminnasta liittyy koko ohjelmistokehityksen prosessiin. Leanin arvoja voidaan nähdä myös koodin tasolla. Järjestelmän koodin kattavuus kertoo turhan koodin määrästä, tiukat rakenteet johtavat aikaisiin päätöksiin ja järjestelmän palveluiden näkyvyys kertoo järjestelmän arvomaailmasta.

5.1.1 Koodin kattavuus

Lean-jätteitä on seitsemää eri tyyppiä ja ne on esitelty kohdassa 2.5 (taulukko 5). Jätteistä yksinkertaisin arvioitava kohta on ylituotanto, joka tarkoittaa tapauksessa ylimääräisen koodin tai toiminnollisuuden toteuttamista. Tutkimalla koodin kattavuutta eri skenaarioissa saadaan kuva ylimääräisen toimintojen määrästä. Liitteen C avulla voidaan laskea kaikki DCI-osion komponenttien attribuutit ja metodit. Attribuutteja ja metodeja löytyy yhteensä 161 kappaletta ja niitä vastaavan Visual Studion IL-kielen koodirivien määrä on 786.

Kontekstit DCI-arkkitehtuurissa merkitsevät järjestelmän käyttötapauksia, joten käymällä läpi kontekstit on käyty läpi järjestelmän tarjoamat palvelut. Konteksti CGame, eli pelin pääkonteksti, kutsuu kaikkia muita konteksteja, paitsi CLoadGame-kontekstia. Käymällä läpi CGamen luokan koodi rivi riviltä ja käymällä kaikkien ehtolauseiden sisällä, on järjestelmän 161 attribuutista ja metodista käyty 132, eli noin 82 %. Lisäämällä tähän CLoadGame kontekstin läpikäynti, joka käy 20:ssä toisistaan eroavassa metodissa, saadaan luku 152 eli noin 94 % kaikista attribuuteista ja metodeista. Käymällä läpi pelkästään kontekstien koodit, on DCI-osion koodeista käyty läpi noin 94 %. Järjestelmässä on kuusi metodia tai attribuuttia, joita ei käytetä ollenkaan. Jäljelle jääneet kolme metodia tai attribuuttia ovat VEdgeMono.Start, VNodeMono.Start ja VPanelMono.Update, joita UnityEngine kutsuu itse, ja ne ovat puhtaasti näkymän ruutupäivityksiin liittyviä toimintoja. Koodiriveinä vastaavat luvut ovat 741, 26 ja 19 eli noin 94,3 %, 3,3 % ja 2,4 % kaikista attribuuteista ja metodeista.

5.1.2 Arkkitehtuurin vaikutus päätöksiin

Leanissä arvostetaan päätösten tekemistä mahdollisimman myöhään. Yleisesti tällä tarkoitetaan asioiden ja päätösten lukkoon lyömistä vasta silloin, kun se on tarpeellista. Ohjelmistokehityksessä toteutustasolla tämä voidaan nähdä järjestelmän rakenteen jäykkyytenä, eli rakenteen asettamien vaatimusten suurena määränä. Coplien ja Bjørnvig [2010] painottavat, että arkkitehtuurin tulisi olla järjestelmän muoto eikä rakenne. Tämä nähdään DCI-arkkitehtuurin asettamista vaatimuksista konkreettiseen toteutukseen asti. Arkkitehtuurissa on kaksi mielenkiintoista huomiota sen jäykkyyteen liittyen: toimintojen toteutukseen liittyvät vaatimukset ja rajapintojen mahdollistaminen.

Konteksti-luokassa on vain käyttötapauksen algoritmi, eli tarkempi ja kattavampi käyttötapaus. Toimintologiikka löytyy rooleista ja toimialaan liittyvät järjestelmän tila-tiedot löytyvät datasta. Arkkitehtuuri ei määritä kuitenkaan mitään vaatimuksia varsinaisiin toteutuksiin. Kontekstit voivat aloittaa esimerkiksi kokonaan toisen alijärjestelmän toiminnan. Tapauksessa jokainen konteksti on oma osio. CLoadGame ei

sisällä minkään tyyppistä näkymää, kun taas CGame pitää sisällään VGamen, joka on pelin graafin näkymä. Kontekstit määrittävät roolit, joihin sen alustuksessa ohjataan niihin sopivat oliot. Tämän jälkeen kontekstit ovat itsenäisiä, eikä mikään kontekstin tekemä toiminto vaikuta suoraan muihin konteksteihin. Tämä tarkoittaa pitkälti sitä, että vaikka kontekseja voidaan kutsua toisen kontekstin sisällä, ei sisäisen kontekstin logiikka ole riippuvainen sitä kutsuvasta logiikasta. Yleisesti päätösten suhteen tämä tarkoittaa, että toimintoja voidaan kehittää toisistaan riippumatta, niin logiikan kuin käytäntöjenkin suhteen. Hyvää ohjelmointia olisi tietysti pitää komponenttien sisäiset rakenteen ja logiikat yhtäläisinä, mutta arkkitehtuuri ei pakota kyseistä rakennetta. Komponenttien varsinaista toteutusta ei ole päätetty etukäteen, vaikka järjestelmä olisi toteutettu DCI-arkkitehtuurin mukaisesti.

Toinen päätöksiin liittyvä huomio on DCI-arkkitehtuurin rajapintojen mahdollistaminen. DCI-arkkitehtuurin toimintojen ja toimialatietojen erottaminen mahdollistaa myös kyseisten osioiden rajapinnoittamisen luontevasti. Pelin toteutukseen C# vaatii roolien tunnisterajapintojen toteuttamisen data-olioihin, mutta itsessään ne toimivat myös data-olio rajapintoina. Kontekstit ja roolien metodit voidaan asettaa suoraan rajapintojen taakse, joten järjestelmä voidaan määrittää rajapintojen avulla käytännössä kokonaan. Tällöin toteutuksia voi tehdä silloin, kun niitä tarvitaan. Toisaalta rajapinnat mahdollistavat myös muiden järjestelmien integroimisen arkkitehtuuriin. Kontekstin rooleja esittäviä oliota voi olla muut järjestelmät, kunhan ne ovat yhdistettynä roolin tunnisterajapintojen avulla.

5.1.3 Konteksti ja käyttötapaukset

Coplien ja Bjørnvig [2010] huomioivat, ettei ohjelmistolla ole arvoa, mikäli sitä ei käytetä. Järjestelmä on kokoelma palveluita, jotka ovat järjestelmän toimintoja. Järjestelmän käyttäjälle näkyvät palvelut löytyvät käyttötapauksista. Käyttötapaukset ovat yleisesti käytetty tekniikka muodostamaan järjestelmästä käyttäjälle merkittäviä toimintoja. [Coplien & Bjørnvig, 2010] Käyttötapauksista on järjestelmän toiminnan ja siten liiketoiminnan kannalta arvoa.

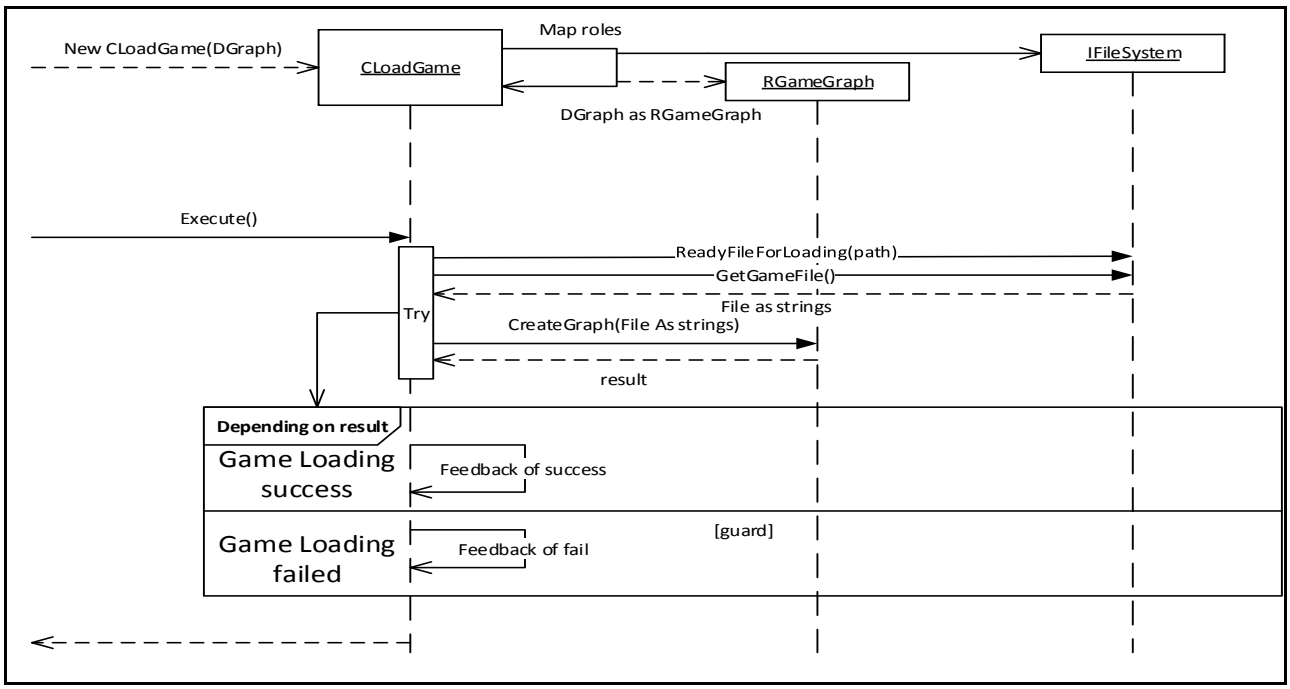
Perinteisessä olio-ohjelmoinnissa käyttötapauksista analysoidaan järjestelmään luokat ja toimijat. Käyttötapauksien tarkoituksena on formalisoida palvelut ja avata tilannetta, joihin järjestelmää kehitetään. DCI-arkkitehtuurissa käyttötapauksista analysoidaan roolit ja niiden toiminnot, ei luokkia. Roolien jälkeen käyttötapauksista tehdään rooleja käyttävä versio, josta tulee käytännössä konteksti. Käyttötapauksen askelluksesta tulee kyseisen kontekstin Execute-metodi.

Pelistä kirjattiin kuusi käyttötapausta: aloittaminen, tallennus, arvaaminen, pelialueella liikkuminen, lopettaminen ja pelisilmukka. Jokaisesta käyttötapauksessa on käyttäjälle arvoa: pelin aloittaminen lataa pelin ja on siten pelattavissa, pelin tallennus tallentaa pelitilanteen, arvaaminen on pelin ydin, pelialueella liikkuminen paljastaa pelaajalle ruudun ulkopuolella olevia pelisolmuja, pelin lopettaminen sulkee pelin ja pelisilmukka tutkii, mitä pelaaja haluaa tehdä.

Tapauksessa on kuusi kontekstia: CGame, CLoadGame, CSaveGame, CQuess, CQuit ja CUpdateCamera. Kontekstit vastaavat käyttötapauksia järjestyksessä: pelisilmukka, pelin aloittaminen, pelin tallentaminen, arvaaminen, lopettaminen ja pelialueella liikkuminen. Kontekstien toiminta on selitetty tarkemmin kohdassa 4.3, mutta niiden velvollisuudet järjestelmässä ovat samat kuin mitä käyttötapauksien kohdalla (taulukot 15 ja 16 sekä kuvat 16 ja 17). Jokaisella kontekstilla on ajettaessa liiketaloudellista arvoa, sillä ne ovat käytännössä samat kuin niitä vastaava käyttötapaus. Loput vertailut näkyvät liitteessä D.

Skenaario		
1.	Pelaaja käynnistää pelin	
2.	Pelijärjestelmä lataa pelin kentän tiedoston	
	2. a	Peli ladataan alkuperäisestä kenttä tiedostosta
	2. b	Peli ladataan viimeksi tallennetusta kenttä tiedostosta
3.	Kenttä näytetään pelaajalle	
4.	Peli on pelattavassa tilassa	

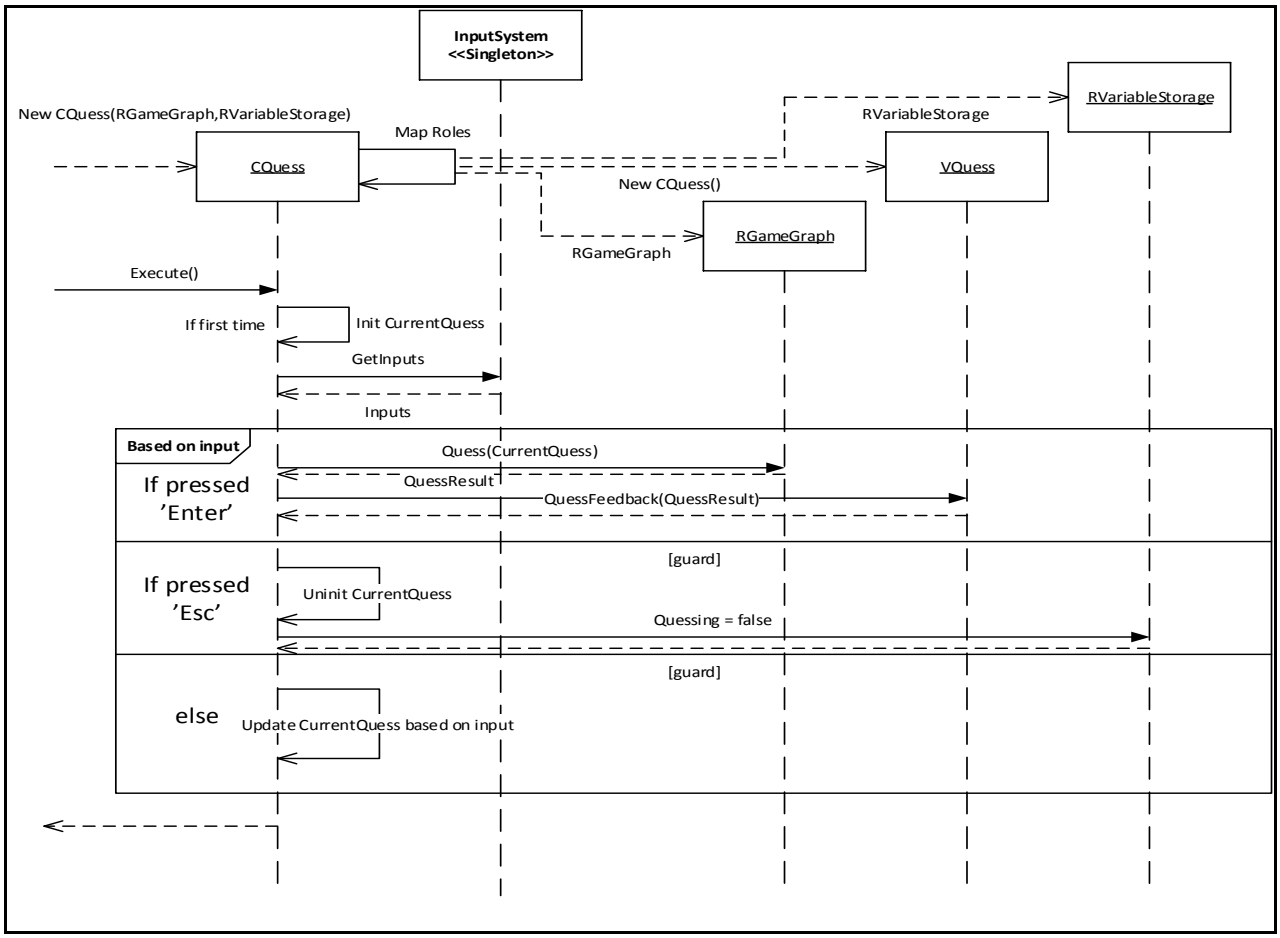
Taulukko 15: Pelin lataamisen käyttötapaus.



Kuva 16: Kontekstin CLoadGame sekvenssikaavio.

Skenaario		
1.		Pelaaja kirjoittaa tekstin teksti kenttään ja komentaa peliä tekemään arvaus tekstillä
2.		Peli tarkistaa arvattavissa olevista solmuista, jos teksti täsmää solmuun asetetun tekstin kanssa
3.	a	Mikäli teksti löytyy jostain kyseisistä solmuista, solmun teksti näytetään kokonaan ja solmun piilossa olleet naapurit näytetään siten, että niiden teksteistä näkyy vain kirjainten ja sanojen määrä
	b	Mikäli ei löydy kyseistä tekstiä, peli ilmoittaa siitä pelaajalle
4.		Peli jää odottamaan uusia komentoja

Taulukko 16: Arvaamisen käyttötapauksen askellus.



Kuva 17: Kontekstin CQuess sekvenssikaavio.

5.2 Ketterät menetelmät

Ketterissä menetelmissä kehitysnopeus on tärkeää. Ohjelmiston nopea kehitys vaatii hyvää ymmärrystä ohjelmasta ja muutoksia vastaanottavan ohjelman. Järjestelmän ymmärrettävyydellä tarkoitetaan sitä, kuinka vaikeaa tai helppoa ohjelmistokehittäjän on ymmärtää kyseisen järjestelmän sisäinen toiminta. Ymmärrettävyyteen vaikuttaa järjestelmän monimutkaisuus. Siinä missä ymmärrettävyys on pitkälle subjektiivinen

käsite, siihen liittyvää järjestelmän monimutkaisuutta voidaan tutkia tarkastelemalla järjestelmän koodia. Tapauksesta on tehty kolme eri kompleksisuuteen vaikuttavaa mittausta: yleinen monimutkaisuus, dynaaminen yhtenäisyys sekä kognitiivis-spatiaalinen kompleksisuus.

5.2.1 Yleiset metriikat

Peliin on laskettu yleisistä metriikoista keskiarvo sekä huonoin yksittäinen arvo (taulukko 17). Syklomaattinen kompleksisuus sekä Visual Studion ylläpidettävyysindeksi (maintainability index) ovat funktiokohtaisia.

Ominaisuus	Keskiarvo	Huonoin arvo	
Perintäpuun koko	~1.25	2	CQuess
Koodirivit	~25	166	CGame
Syklomaattinen kompleksisuus	~9.226	31	RTGameGraph.CreateGraph
VisualStudion ylläpidettävyysindeksi	~84.6	34	RTGameGraph.CreateGraph
Monimuotoisuus	~0	5	CGame

Taulukko 17: Visual Studion Code Metrics -työkalun antamista tiedoista. Koko taulukko on liitteessä E.

Visual Studion koodimittaustyökalu (code metrics) on laskenut mittaukset ohjelman koodista. Koodirivien määrä ei ole suora lähdekoodin koodirivimäärä, vaan se perustuu C#-koodista käännettyyn IL-kieleen. [Microsof, 2015] Microsoftin ylläpidettävyysindeksi antaa ”OK” merkinnän mikäli kyseinen metriikka saa arvon, joka on yli 20. Tapauksessa huonoin osa sai 34 ja se sai myös huonoimman CC-arvon.

Syklomaattisen kompleksisuuden tulosrajat ovat seuraavat: 1-10 matalan, 11-20 keskitason, 21-50 korkea ja yli 50 hyvin korkean riskitason metodi. Syklomaattisen kompleksisuuden mukaan järjestelmä on keskimäärin matalan riskitason järjestelmä. Osat, jotka ylittivät mittausrajan 10, olivat CQuess.Execute (15), CGame.Execute (28), CQuit (11), VGame.UpdateNode (11) ja RTGameGraph.CreateGraph (31).

Perintäpuun tulisi olla DCI-arkkitehtuurissa hyvin matala verrattuna perinteiseen olio-ohjelmointiin. Pelin perintäpuun korkeus on kaksi, jota voidaan yleisesti pitää matalana. Koodirivien määrä tulisi yleisesti olla alle 100, jotta luokka pysyy hallittavissa. CGame ja RTGameGraph ovat pelin ainoita osia, joiden rivimäärä nousee yli sadan koodirivin. Järjestelmässä olio-ohjelmoinnin monimuotoisuutta käytettiin yhteensä 12 kertaa. Näistä kaksi liittyi UnityEnginen vaatimaan tilanteeseen (VGame.31 ja VGame.41) ja loput

käyttökerrat olivat InputSystemissä, joiden toteutukset eivät kuulu DCI-arkkitehtuuriin. DCI-arkkitehtuurissa käytetty monimuotoisuus on siis minimaalinen.

Olio-ohjelmoinnin kompleksisuuteen ei ole yksiseliteisiä objektiivisia metriikoita. Yleisesti, jos katsotaan useita eri metriikoita ja niiden yhteistulosta, voidaan varovasti ajatella niiden kertovan järjestelmän varsinaisesta kompleksisuudesta. Pelistä lasketut perinteiset metriikat saavat yleisesti hyvät arvosanat kompleksisuuden suhteen.

5.2.2 Dynaaminen yhtenäisyys

Guptan ja Chhabran [2011] dynaaminen yhtenäisyys voidaan laskea staattisesti luokille tai dynaamisesti oliolle. Mittauksissa on erikoismetodeja, jotka voidaan jättää pois laskuista, sillä yleisesti on todettu, että kyseisen tyyppiset metodit eivät vaikuta yhtenäisyyteen. Erikoismetodit ovat rakentaja, tuhoaja, käsittelijät ja edustajat (delegate). DCI-arkkitehtuurissa data-oliot ovat puhtaasti attribuutteja, mikäli käsittelijät otetaan pois. Roolien tunnisterajapinnat ovat vain käsittelijöitä, joten rooleista jää jäljelle niiden roolitoteutukset. Mittauksissa oletetaan, että luokalla on sekä attribuutteja että metodeja. Oletus ei DCI-arkkitehtuurin suhteen toimi staattisesti, joten luokkien mittaaminen jätetään pois.

Mittauksissa skenaarioita ovat jokaisen kontekstin alustus ja Execute-metodin kutsuminen siten, että jokainen ehtolauseiden vaihtoehto on käyty läpi ainakin kerran. Oliota kyseiseissä tapauksissa ovat konteksti-oliot sekä DGraph, DNode, DEdge ja DCamera. Näistä jokainen on roolissa järjestyksessä RGameGraph, RGameNode, RGameEdge ja RGameCamera. Mittaukset on tehty käsin, koska C# ei tue Guptan ja Chhabran ehdottamaa versiota, joka on tehty Javalle käyttämällä aspektiperustaista ohjelmointia (aspect-oriented programming). Mittausten tulokset näkyvät taulukossa 18. Mittausten kaavat ovat liittessä A.

Skenaariot	Oliot				
LoadGame			Painot		
	$DOC_{LoadGame}$(CLoadGame)	*0.3000*	DC_AM	w1	4
	$DOC_{LoadGame}$(DGraph)	0.0905	DC_MA	w2	3
	$DOC_{LoadGame}$(DNode)	0.0639	DC_MM	w3	2
	$DOC_{LoadGame}$(DEdge)	0.3000	DC_AA	w4	1
Quess					
	DOC_{Quess}(CQuess)	*0.7000*			
	DOC_{Quess}($CQuess_{View}$)	*0.7000*			
	DOC_{Quess}(DGraph)	0.0292			
	DOC_{Quess}(DNode)	0.0250			
CGame					
	$DOC_{GameLoop}$(CGame)	0.4000			
	DOC_{Quess}(DGraph)	0.0714			
UpdateCamera					
	$DOC_{UpdateCamera}$(CUpdateCamera)	*0.4000*			
	$DOC_{UpdateCamera}$(RGameCamera)	0.4333			

Taulukko 18: Dynaamisen yhtenäisyyden tulokset.

Mittaustuloksista (taulukko 18) voidaan huomioida konteksti-olioiden merkittävän suuri yhtenäisyys verrattaessa rooleja käyttäviin olioihin. Erot johtuvat olioiden ominaisuuksien käytön erosta. Skenaarioissa käydään konteksti läpi, mikä on mielekästä siksi, että skenaarioiden ajamisessa on palvelun toiminnallista arvoa. Skenaariot käyvät kontekstien kaiken toiminnan läpi, joten luonnollisesti niiden attribuutteja ja metodeja käytetään paljon. Toisaalta konteksteissa ei myöskään ole paljoa ylimääräistä koodia, sillä niissä on pääosin vain kontekstin ajamiseen liittyvää koodia. Mittaustulokset viittaavat kontekstien itsenäisyyteen eri skenaarioiden välillä. Rooleja käyttävissä oliossa, varsinkin DGraph-oliossa, on usean kontekstin käyttämiä metodeja ja attribuutteja. Luonnollisesti kaikkia kyseisiä ominaisuuksia ei käytetä yhdessä skenaariossa, mutta ne otetaan suhdeluvussa mukaan jakajaan. Roolien käyttäminen eri tavalla eri konteksteissa näkyy siis mittaustuloksissa vähäisenä yhtenäisyytenä.

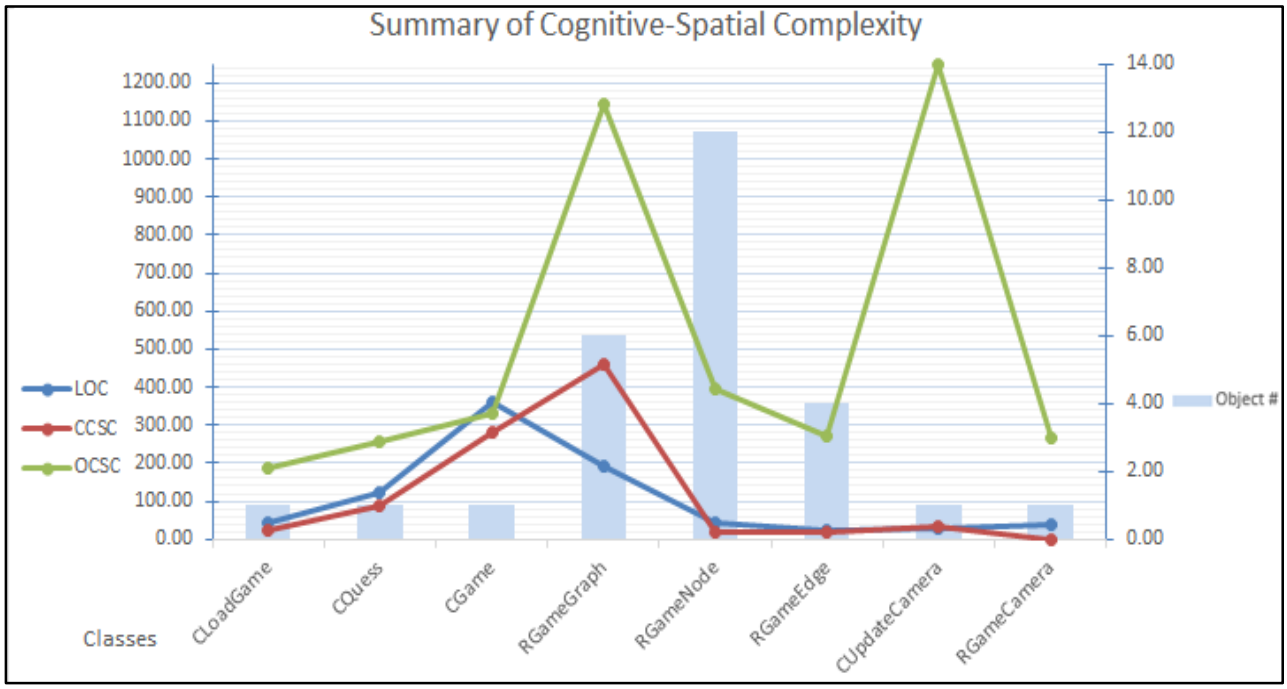
Mielenkiintoinen huomio dynaamisen yhtenäisyyden määritelmässä on, että erikoismetodeja jätetään sen ulkopuolelle. DCI-arkkitehtuurissa data-oliot ovat pitkälti kyseisten erikoismetodien avulla yhdistettyinä rooleihin, joten data-olioiden koodi ei vaikuta laskettuihin yhtenäisyyksiin.

5.2.3 Kognitiivis-spatiaalinen kompleksisuus

Kognitiivis-spatiaalisessa kompleksisuudessa laskettiin eri etäisyysarvoja luokille ja oliolle, joita painotettiin käytön kompleksisuudella. Luokille lasketaan attribuuttien ja metodien käytön etäisyyksiä. Oloille lasketan olion luomisen ja sen jäsenten käytön etäisyyksiä. Kaikkia etäisyyksiä painotetaan sen kontrollirakenteen mukaan. Laskujen yhteenveto on esitetty taulukossa 19 ja kuvassa 18. Tarkemmat laskukaavat ovat liitteessä B.

	CLoad Game	CQuess	CGame	RGame Graph	RGame Node	RGame Edge	CUpdate Camera	RGame Camera
LOC	44.00	121.00	363	193	44.00	25.00	29	37
CACSC	25.00	90.00	129.94	274.35	18.14	18.50	31	0
CMCSC	0.00	0.00	153	186	0.00	0.00	0	0
CCSC	25.00	90.00	282.94	460.35	18.14	18.50	31	0
Object #	1.00	1.00	1	6	12.00	4.00	1	1
ODCSC	78.00	92.00	82	70	112.33	141.75	94	254
OMCSC	108.00	162.00	249	1076.25	282.86	129.50	1154	10
OCSC	186.00	254.00	331	1146.25	395.19	271.25	1248	264

Taulukko 19: Kognitiivis-spatiaalisen kompleksisuuden tulokset.



Kuva 18: Kognitiivis-spatiaalisen kompleksisuuden tulosten kuvaaja.

Guptan ja Chhabran [2009] mukaan kognitiivis-spatiaalinen kompleksisuus kertoo enemmän järjestelmän ymmärtämisen vaikeudesta kuin kognitiivinen toiminnallinen koko (cognitive functional size, CFS) tai olion ja luokan kognitiivinen kompleksisuus (object

spatial complexity ja class spatial complexity, CSC ja OSC). Tapauksesta ei ole kuitenkaan varsinaista vertailukohdetta, sillä mittauksia ei ole tehty vastaaviin ohjelmiin. Mittaustuloksia voi analysoida kuitenkin luokkien kesken.

Merkittävänä huomiona mittauksissa on keskiarvon käyttäminen. Esimerkiksi CUpdateCamera saa korkean OMCSC-arvon verrattuna muihin luokkiin ja niiden arvoihin. Tämä johtuu siitä, että CUpdateCameralla on vain yksi metodi, Execute, jota kutsutaan syvällä CGamen Execute-metodissa. CUpdateCameraa kutsutaan kutenkin vain kerran, ja koko metodin rivimäärä on 11. CUpdateCameran Execute-metodin käyttötilanne mittausten perusteella näyttäisi olevan ymmärtämisen kannalta raskasta, mutta käytännössä se on hyvin suoraviivaista. Rivimäärän käyttö etäisyytenä ja keskiarvojen avulla tasapainottaminen on epävakaa pienillä arvoilla.

Yleisten mittausten kohdalla CGame ja RGameGraph olivat selvästi monimutkaisimpia järjestelmän osia. Kognitiivis-spatiaalisessa mittauksessa CGame ei korreloi tätä monimutkaisuutta. Käytännössä kognitiivis-spatiaalisen mittauksen perusteella CGamen käyttäminen järjestelmässä ei ole ajatukselle raskasta. Käytännössä kaikki rooliluokat saavat merkittävän suuret OCSC-arvot suhteutettuna esimerkiksi koodirivien määrään. OCSC mittaa olioiden ja niiden käytön tilanteita. Rooliluokkia käytetään merkittävästi monipuolisemmin kuin kontekseja, joten niiden kognitiivis-spatiaaliset arvot ovat suuremmat. ODCSC:lla ja OCSC:lla ei ole lineaarista yhteyttä olioiden lukumäärän tai koodirivien määrän kanssa. Yhteyden puuttuminen on mielenkiintoinen huomio, sillä rooliluokkien kognitiivis-spatiaalinen arvo perustuu roolin tunnisterajapinnan ja varsinaisen rooliluokan yhdistelmään, eikä data-osiota käytännössä oteta huomioon. Järjestelmän ymmärtämisen kannalta kognitiivis-spatiaalinen kompleksisuus säilyy samana riippumatta data-oliosta. Lineaarisen riippuvuuden puuttuminen voi merkitä myös sitä, että roolin ymmärtäminen ei vaikeudu, vaikka sen käyttöä lisättäisiin ja hajautettaisiin.

5.2.4 Mittaustulokset

DCI-arkkitehtuuri on kehitetty pitkälti muodostamaan järjestelmästä ymmärrettävämpi kokonaisuus. Tapauksen järjestelmä on mittaustulosten perusteella ymmärrettävä ja ylläpidettävä. Mittauksien tuloksiin pitää kuitenkin suhtautua merkittävällä varauksella. Tapauksen DCI-arkkitehtuuria ei ole varmistettu ulkoisilta tahoilta, eikä yhden tapauksen mittauksilla voi saada kokonaiskuvaa DCI-arkkitehtuurin varsinaisesta toiminnasta. Toisaalta mittaukset eivät ole ristiriidassa väitettyjen hyötyjen kanssa, joten sitä voidaan käyttää kyseisten väitteiden tukena.

Järjestelmän ymmärrettävyys ja ylläpidettävyys ovat hyvin subjektiivisia käsitteitä, joten niiden toteutumista on vaikea mitata. Mittauksissa ei ole myöskään otettu huomioon DCI-arkkitehtuurin eroa perinteisiin olioperustaisiin järjestelmiin verrattuna, joissa luokat koostuvat tiedoista ja toiminnoista. Dynaaminen yhtenäisyyden sekä kognitiivis-spatiaalisen kompleksisuuden määritelmissä oletetaan, että luokat muodostuvat tiedoista ja niitä muokkaavista metodeista. DCI perustuu pitkälti konteksti-olion sisällä tapahtuvaan roolien väliseen toimintaan, eivätkä mittaukset ota järjestelmän kokonaisuutta merkittävästi huomioon. Mittauksien määritelmät tukevat väitettä, että olioiden kommunikaatio on jäänyt perinteisessä olio-ohjelmoinnissa taka-alalle. Kognitiivis-spatiaalinen kompleksisuus laskee olion alkioiden käyttöä, mutta ei esimerkiksi kahden olion käyttöä yhdessä. Dynaaminen yhtenäisyys laskee olioiden alkioiden käyttöä vain olion omissa alkioissa. Mittauksia tulisi kehittää huomioimaan olioiden välinen kommunikaatio ja sen kompleksisuus, jotta DCI-arkkitehtuuria olisi mielekästä mitata objektiivisilla mittareilla.

5.3 Muutosesimerkki

DCI-arkkitehtuurilla tehdyn järjestelmän on tarkoitus mahdollistaa muutoksia helposti. Muutostilanteen havainnollistamiseksi peliin lisätään uusi toiminto. Uusi toiminto on kentän valinta. Alun perin pelissä oli mahdollisuus pelata vain yhtä kenttää nimeltä default.txt. Mikäli halusi muuttaa pelattavaa kenttää, kentän nimeksi piti laittaa default.txt. Uusi toiminto näyttää listan mahdollisista kentistä. Valitsemalla kentän sekä painamalla ”Load” kyseistä kenttää voidaan pelata. Peli tallentaa jatkossa muuttuneen tilanteen kyseiseen kenttään.

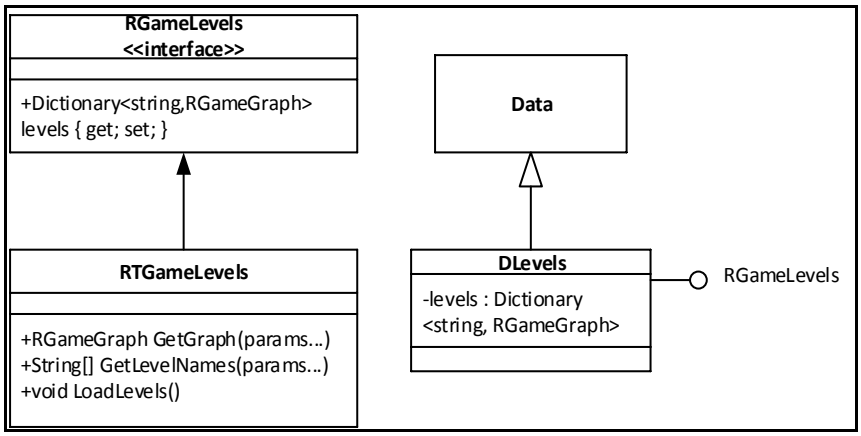
Uuden toiminnon lisääminen alkoi ajattelemalla, mitä kyseisellä toiminnolla halutaan saada aikaan. Toiminnon tavoite oli varsin selkeä, eli valita mahdollisista pelattavista kentistä pelaajan haluama kenttä ja aloittaa sen pelaaminen. Pelin lataamisen ja siten pelaamiseen siirtyminen vaatii toimiakseen tiedon kentän tiedoston nimestä. Uuden toiminnon tulisi siten mahdollistaa kentän tiedoston nimen palauttaminen Manager-luokan tasolle. Valittavien kenttien näyttäminen niiden nimien listana ja kentän valitseminen klikkaamalla sitä tuntui luontevimmalta ratkaisulta. Muutos vaati selkeästi näkymän, joten seuraavaksi toimintoon tehtiin kevyt luonnos näkymästä. Visuaalinen toteutus ei kuulu DCI-arkkitehtuurin osioon, joten sitä ei käydä tässä tarkemmin läpi. Karkean näkymän luominen toiminnosta auttoi myös vahvistamaan ajatusta sen logiikasta. Kun toiminnosta oli saatu tarkempi kuva, myös sen integroimisesta tehtiin korkean tason kartoitus. Integroimisesta saatiin seuraavat huomiot: CLoadGame lataa nimen BaseFileSystemistä, joten sen alustaminen valitulla kentällä tapahtuu ennen CLoadGamen kutsumista; peli

alkaa CLevelSelection-kontekstin ajamisella, mutta sen jälkeen pelin toiminta pysyy samanlaisena.

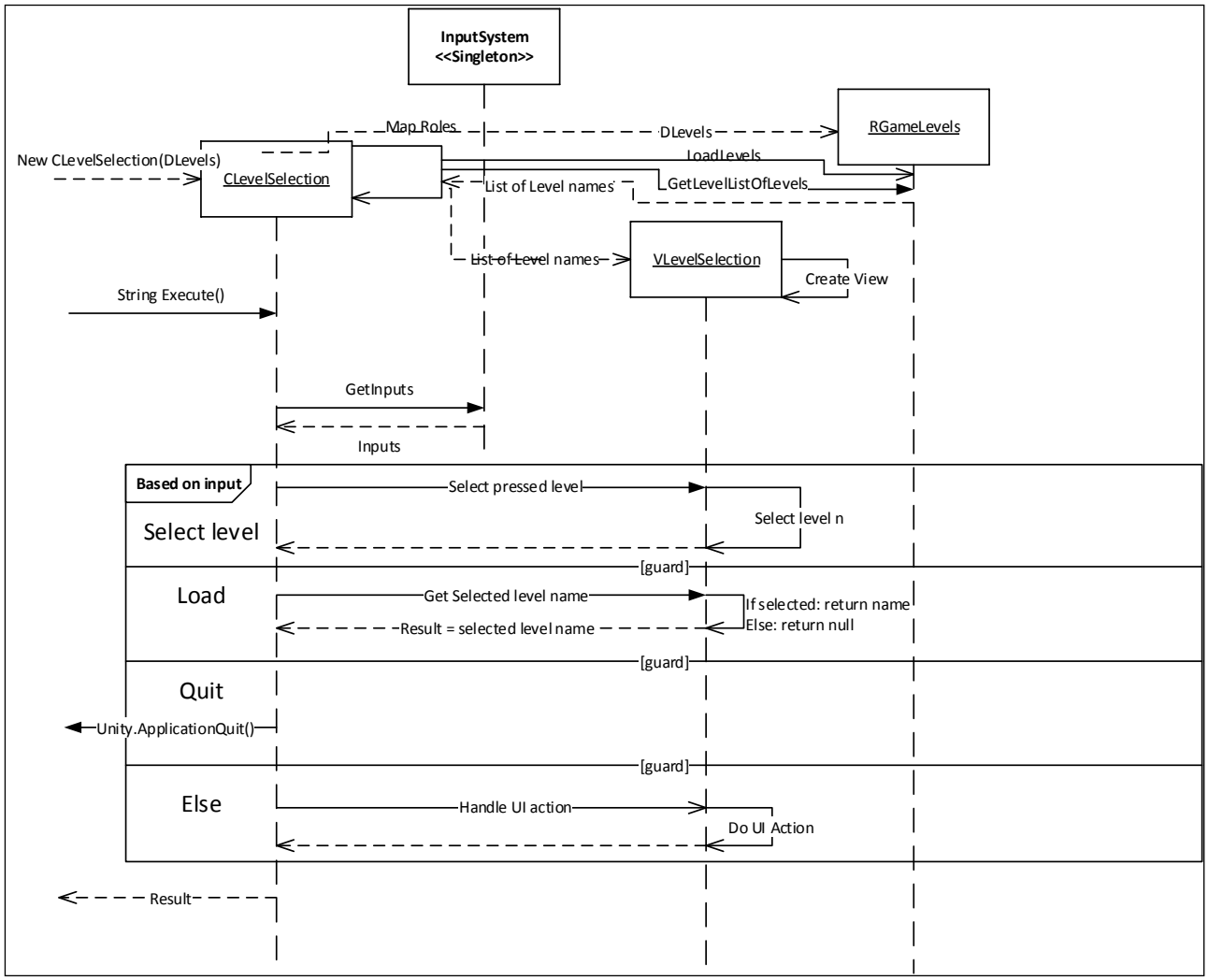
Toiminnon kehitys jatkui sen formalisoinnilla, eli käytännössä käyttötapauksen luomisella, joka on kuvattu taulukossa 20. Käyttötapausta analysoimalla kenttälistasta luotiin rooli RGameLevels ja siihen liittyvä data-olio DLevels. Luokkakaavio uusiin komponentteihin näkyy kuvasta 19. RGameLevels vaatii toimiakseen paikan, jonne pelikenttien nimet tallennetaan. Tästä syystä DLevels on toteutettu Dictionary<string,RGameGraph>-muodossa. RGameLevels-roolin toimintoihin kuuluvat nimilistan lataaminen ja sen palauttaminen. Seuraavaksi käyttötapauksesta luotiin konteksti. Kontekstin sekvenssikaavio on esitelty kuvassa 20.

Kuvaus		
Käyttötapaus	Pelikentän valinta.	
Selostus	Kun pelaaja käynnistää pelin, peli näyttää pelaajalle listan mahdollisista kentistä, joista pelaaja voi valita haluamansa ja pelata sitä.	
Toimijat	Peli, pelaaja.	
Alkutilanne	Peli on käynnistetty.	
Jälkitilanne	Pelaaja on lopettanut koko pelin, tai aloittanut pelaamaan valittua kenttää.	
Skenaario		
1.	Peli tarjoaa näkymän mahdollisista kentistä.	
2.	Pelaaja valitsee kentän tai lopetuksen.	
	a.	Pelaaja valitsee kentän ja aloittaa sen (taulukko 8).
	b.	Pelaaja valitsee lopettamisen.
Huomioita		
2b. Pelaajalle ei voi ehdottaa tallentamista, joten pelin lopetuksen käyttötapausta ei sovelleta kentän valinnassa.		

Taulukko 20: Kentän valitsemisen käyttötapaus.



Kuva 19: Muutososion data ja rooli luokkakaaviona.



Kuva 20: Kontekstin ClevelSelection sekvenssikaavio.

Uuden toiminnon kehityksessä testaamisen tarve oli äärimmäisen pieni. Suurin osa testaamisesta tapahtui näkymän kehityksessä, mutta DCI-arkkitehtuurin osion testaamista ei merkittävästi tarvinnut. Muutoksen DCI-arkkitehtuurin komponentit ovat todella

yksinkertaisia, joten koodin lukeminen niiden oikeellisuuden varmistamiseksi oli riittävä. Ajonaikaista testaamista suoritettiin eri määrillä tiedostoja pelikenttäkansiossa. Tapauksessa voidaan olettaa, että kenttiä on aina vähintään 1 ja korkeintaan 100. Kontekstien moduulimaisuudesta johtuen testaamista voitiin suorittaa suoraan pelin alkuperäisellä rungolla siten, että muiden kontekstien kutsut otettiin pois käytöstä. Kaikilla raja-arvoilla peli toimii oikein.

Muutoksen integroiminen vaati muutoksia Manager- ja CLoadGame-luokkiin. CLoadGame-luokasta poistettiin kaksi riviä koodia, jotka liittyivät BaseFileSystemin alustamiseen. Manager-luokan koodi muuttui yhteensä noin 16 rivin verran. Manager-luokan muutokset näkyvät koodikatkelmista 6 ja 7. Muutoksesta huomaa, että vanhan Manager-luokan Start-metodi on lähes sellaisenaan uuden luokan Update-metodin startLevelLoading-lippumuuttujan alla. Vanha Update löytyy osittain gameLoaded-lippumuuttujan alta. Update-metodin ensimmäinen osio käsittelee CLevelSelection-kontekstin ajamista ja Start sen alustusta. CLoadGame-kontekstista poistettiin riveiltä 23 ja 24 koodi ”string filename = Manager.DEFAULT_LOAD_NAME; bfs.SetRelativePath(filename);”. Tämä löytyy nyt uuden Managerin Update-metodista. Manager.DEFAULT_LOAD_NAME on korvattu CLevelSelection-kontekstin palauttamalla valitun kentän tiedoston nimellä.

```
…
private CGame gameContext = null;
// Use this for initialization
void Start () {
	Application.targetFrameRate = 30;
	instance = this;
	DGraph dg = new DGraph();
	Dictionary<string,Vector3> nodePositions = new Dictionary<string, Vector3>();
	CLoadGame lgc = new CLoadGame(dg,nodePositions,BaseFileSystem.GetInstance());
	lgc.Execute();
	gameContext = new CGame(dg,nodePositions,InputSystem.GetInstance());
}
void Update()
{
	gameContext.Execute();
	printMessages();
}
…
```

Koodikatkelma 6: Manager-luokan vanha logiikka.

```
...
private CGame gameContext = null;
private CLevelSelection levelSelection = null;
// Use this for initialization
void Start () {
        Application.targetFrameRate = 30;
        instance = this;
       DLevels lvls = new DLevels();
       levelSelection = new CLevelSelection(InputSystem.GetInstance(), lvls);
}
void Update()
{
    if(!startLevelLoading && !gameLoaded)
    {
        string filePath = levelSelection.Execute();
        if (filePath != null)
        {
           startLevelLoading = true;
           BaseFileSystem.GetInstance().SetRelativePath(filePath);
           GameObject.Find("LevelSelection").SetActive(false);
        }
    }
    else if(startLevelLoading)
    {
        DGraph dg = new DGraph();
        Dictionary<string, Vector3> nodePositions = new Dictionary<string, Vector3>();
        CLoadGame lgc = new CLoadGame(dg, nodePositions, BaseFileSystem.GetInstance());
        lgc.Execute();
        gameContext = new CGame(dg, nodePositions, InputSystem.GetInstance());

        gameLoaded = true;
        startLevelLoading = false;
    }
    else if(gameLoaded)
    {
        gameContext.Execute();
    }
         printMessages();
}
...
```

Koodikatkelma 7: Manager-luokan uusi logiikka.

Muutosesimerkistä näkee yhden tilanteen, miten uuden toiminnon lisääminen etenee DCI-arkkitehtuurissa. Muutoksen työmäärä keskittyi pääosin halutun toiminnan suunnitteluun ja näkymän toteutukseen. Vain pieni osa kokonaistyömäärästä meni integroinnin huomioimiseen. Teoriassa uuden käyttötapauksen luomisessa ei ole väliä, toteutetaanko se järjestelmän varsinaisessa kehitysvaiheessa vai esimerkiksi käyttäjien pyynnöstä julkaisun jälkeen. Mikäli uusi käyttötapaus ei muuta olemassa olevia käyttötapauksia, ei muutoksia muihin konteksteihin pitäisi tapahtua. Muutosesimerkissä ei tehty muutoksia muihin käyttötapauksiin, joten toteutuksessakaan ei muihin konteksteihin tarvinnut koskea. BaseFileSystemin alustuksen poistaminen CLoadGamesta ei muuttanut kontekstia ja alustus olisi ollut järkevää tehdä Manager-luokassa alun perinkin.

Muutoksen kontekstia, rooleja ja dataluokkia kehittäessä tulisi huomioida olemassa oleva koodi. Esimerkin muutoksessa jo olemassa olevat data-luokat, roolit ja kontekstit eivät voineet auttaa uuden kontekstin toteutuksessa, joten jokainen jouduttiin tekemään erikseen. DLevels-luokan lista käyttää kuitenkin RGameGraph-roolia hyödyksi, joten

kyseinen data-luokka voi pitää sisällään ladattujakin graafeja. Jatkokehityksen suhteen useita pelejä voisi olla muistissa ladattuna yhtä aikaa käyttämällä valmista DLevels-luokkaa data-oliona.

Muutoksen toteuttaminen tapahtui pitkälti siten, miten DCI-arkkitehtuurissa sen on kuviteltu tapahtuvan. Peliin tehtiin kuitenkin vain yksi muutos, joten sitä ei voi yleistää edes kyseisen järjestelmän tasolla. Muutoksen toteuttaminen tapahtui kuitenkin hyvin puhtaasti, joten olisi mielenkiintoista tutkia DCI-arkkitehtuurin käyttötapauksien toteuttamista ennen ja jälkeen järjestelmän julkaisun. Muutoksen perusteella niiden työmäärät eivät eroa välttämättä paljoa ja siten ainakin uuden käyttötapauksen lisäämisen suhteen DCI-arkkitehtuuri näyttäisi tukevan ketterän kehityksen vaatimaa nopeutta.

5.4 Yhteenveto arvioinnista

Coplien ja Bjørnvig [2010] ovat esittäneet DCI-arkkitehtuurin ratkaisuksi ketterän kehityksen ja leanin yhdistämiseksi. DCI-arkkitehtuurin kehittäjä Reenskaugin [2009] mukaan DCI-arkkitehtuurin avulla voidaan luoda ymmärrettävää koodia. Ymmärrettävyys on syvässä yhteydessä ylläpidettävyyteen, joka puolestaan nähdään tärkeäksi järjestelmän ominaisuudeksi [Boehm et al., 1976].

Lean-ajattelumallin suhteen DCI-arkkitehtuuri toteuttaa lean-ajattelua ainakin kolmella tapaa: toteutuksessa on vähän ylimääräistä koodia, se mahdollistaa päätösten tekemisen myöhään sekä siitä löytyy suoraan järjestelmän toiminnan arvoa. Toteutuksessa ennen muutoksia oli noin 3,3 % koodia, jota ei käytetty. Kahden kontekstin alustuksen ja Execute-metodin avulla järjestelmän koodista voitiin suorittaa noin 94 %. Tapauksen suhteen turhaa koodia ei syntynyt paljoa ja koodin käyttäminen kohdistui suoraan kontekstien suorittamiseen. Myöhäistä päättämistä DCI-arkkitehtuuri tukee kahdella tavalla: se mahdollistaa luontaisen rajapinnoittamisen, eikä se ota kantaa komponenttien lopulliseen toteutukseen. Rajapintojen avulla komponenttien rakenne voidaan määrätä, mutta toteutusta voidaan siirtää siihen tilanteeseen, jossa kyseistä osaa käytetään. Kontekstit hyödyntävät rooleja käyttötapauksien suorittamiseksi, mutta roolien varsinainen toteutus jää kehittäjien vastuulle. Viimeisenä asiana leanin suhteen kontekstit ovat käyttötapauksien ilmentymä koodissa. Coplien ja Bjørnvig [2010] ovat todenneet, että järjestelmällä ei ole käyttäjälle arvoa, mikäli sitä ei käytetä. Käyttötapaukset ovat juuri ne palvelut, joiden vuoksi järjestelmää käytetään. Käyttötapauksien löytyminen koodista kontekstina määrittää sen, että kontekstin suorittaminen tarkoittaa käyttötapauksen suorittamista ja siten sillä on suoraan järjestelmän toiminnallista arvoa.

Ketterää kehitystä DCI tukee mahdollistamalla iteratiivista kehitystä ja pitämällä arkkitehtuurin komponentit ylläpidettävinä. Ylläpidettävyyden suhteen tarkastellaan ymmärrettävyyttä, jota tutkittiin järjestelmän kompleksisuudella. DCI-arkkitehtuuri tukee iteratiivista kehitystä aiemmin mainittujen rajapintojen avulla. Lisäksi se tukee kontekstien toisistaan riippumatonta kehitystä. Mikäli järjestelmän käyttötapaukset eivät ole riippuvaisia toisistaan, niitä voidaan kehittää yhtä aikaa erillisesti. Kontekstien riippumattomuus säilyi, kun tapaukseen esiteltiin muutos. Järjestelmän tietojen ja toimintojen selkeä erottaminen mahdollistaa toimintojen kehittämisen ja muokkaamisen rikkomatta järjestelmän tietoja tai muita komponentteja. Järjestelmän ymmärrettävyyttä arvioitiin kompleksisuudella, jossa tutkittiin perintäpuun kokoa, syklomaattista kompleksisuutta, Visual Studion ylläpidettävyysindeksiä, dynaamista yhtenäisyyttä sekä kognitiivis-spatiaalista kompleksisuutta. Perintäpuun koko ja monimuotoisuuden käyttö minimaalista, syklomaattinen kompleksisuus oli hyvin matala tai melko matala ja Visual Studion ylläpidettävyysindeksin mukaan kaikki komponentit olivat hyvin ylläpidettävissä. Dynaamisen yhtenäisyyden ja kognitiivis-spatiaalisen kompleksisuuden metriikat kertovat järjestelmän sisäisistä kompleksisuuteen liittyvistä ominaisuuksista, mutta vertailukohteiden puuttuessa niiden tulokset eivät kerro paljoa pelin varsinaisesta kompleksisuudesta. Oleellisesti tulokset eivät ole ristiriidassa hyvän ymmärrettävyyden kanssa.

Metriikoiden käyttäminen järjestelmän arvioinnissa on ongelmallista. DCI-arkkitehtuurista ei ole tämän tutkielman kirjoittamisen aikaan laajoja esimerkkitapauksia, joten sen suora vertaaminen esimerkiksi työmäärän suhteen ei ole mahdollista. Metriikat kertovat järjestelmän sisäisestä toiminnasta, mutta ongelmaksi tulee niiden määrittäminen. Metriikoita ei ole optimoitu DCI-arkkitehtuurin kaltaiseen rakenteeseen, mikä näkyy esimerkiksi olioiden keskinäisen kommunikaation puuttumisena mittauksista. Reenskaug ja Coplien [2009] painottavat, että olioiden keskeinen kommunikaatio on jäänyt huomioimatta luokkapohjaisessa olio-ohjelmoinnissa, ja metriikoiden määritelmät [Gupta & Chhabra, 2011; Gupta & Chhabra, 2009] tukevat tätä väitettä. DCI-arkkitehtuuri näyttää tuovan leanin ja ketterän kehityksen ideologian mukaisia arvoja järjestelmään. Yleisten mittauksien perusteella tapaus on ymmärrettävä, eikä ole siten ristiriidassa Reesenkaugin [2009] väitteiden kanssa. DCI-arkkitehtuurista on kuitenkin vähän tutkittua tietoa, eikä tällä hetkellä ole metriikoita, joita olisi optimoitu DCI-arkkitehtuurin tyyliseen rakenteeseen kooditasolla. DCI-arkkitehtuuri vaikuttaa lupaavalta vaihtoehdolta perinteisiin olio-ohjelmointirakenteisiin, mutta sen tarkempaan arviointiin vaaditaan enemmän tapaustutkimuksia.

6 Yhteenveto

Tutkielmassa perehdyttiin Reenskaugin [2009] luomaan DCI-arkkitehtuuriin, joka on luotu olio-ohjelmoinnin kompleksisuuden vähentämiseksi. DCI-arkkitehtuuri yhdistää ketterän kehityksen ja lean-ajattelun uudenlaiseksi olio-ohjelmointitavaksi, jossa järjestelmä jaetaan toimintoihin ja toimialalogiikkaan. [Coplien & Bjørnvig, 2010] Jaottelulla saavutetaan nopeasti muuttuvien osien erottaminen hitaasti muuttuvista osista, minkä avulla järjestelmän ylläpidettävyyttä kohennetaan. Järjestelmässä toimialalogiikka kuvataan perinteisemmällä luokkaohjelmoinnilla, jossa luokkien metodit ovat puhtaasti toiminnoista riippumattomia. Toiminnot kuvataan konteksteilla ja rooleilla, joiden avulla järjestelmän toiminta saadaan vastaamaan paremmin ihmisen ajatusmallia.

Tutkielmassa käsiteltiin DCI-arkkitehtuuria sen mukaisesti luodun esimerkkipelin avulla. Peliä arvioitiin ketterän kehityksen ja lean-ajattelun näkökulmista. Ketterän kehityksen arvioinnissa keskityttiin järjestelmän kompleksisuuteen ja lean-ajattelussa järjestelmän kokonaisuuteen. Pelin kompleksisuutta arvioitiin perinteisillä kompleksisuusmittareilla ja kahdella olio-ohjelmointiin liittyvällä kompleksisuusmittarilla. Lean-ajattelusta järjestelmää arvioitiin koodin liiketaloudellisen arvon, koodin kattavuuden ja koodin rakenteiden jäykkyyden näkökulmista.

Mittausten tulokset viittasivat siihen, että pelin DCI-arkkitehtuuri tuki lean-ajattelua ja ketterää kehitystä. Monet olio-ohjelmoinnin metriikat kuitenkin olettavat perinteisen luokkakoostumuksen, jota DCI-arkkitehtuuri ei täytä. Perinteisesti luokka koostuu sekä tiedoista että metodeista, mutta DCI-arkkitehtuurissa ne muodostetaan dynaamisesti yhdistämällä roolit ja data-oliot kontekstin ajaksi. Tutkielmassa käytettyjä mittareita ei ole optimoitu DCI-arkkitehtuurin rakenteeseen, joten niillä saadut tulokset ovat kyseenalaisia.

DCI-arkkitehtuurilla oli mahdollista tehdä oikea peli, eikä sen käytöstä seurannut merkittäviä ongelmia. DCI-arkkitehtuurin kompleksisuuteen ei ole tämän tutkielman kirjoittamisen hetkellä tunnettuja mittareita, joten teoreettinen arviointi on ongelmallista. DCI-arkkitehtuurilla tulisi toteuttaa pieniä ja keskisuuria ohjelmistoprojekteja, joista voidaan saada reaalimaailman tietoa tapaustutkimuksilla.

DCI-arkkitehtuurissa kontekstien ja roolien kompleksisuus ovat riippuvaisia järjestelmän käyttötapauksista ja data-osio on riippuvainen toimialan kompleksisuudesta. DCI-arkkitehtuurin tapaustutkimuksissa tulisi keskittyä esimerkiksi siihen, toteutuvatko nämä riippuvuudet reaalimaailman järjestelmissä ja vaikuttaako siihen järjestelmän suuruus. Mikäli riippuvuudet kohtaavat reaalimaailman toteutuksissa, DCI-arkkitehtuuri voi myös hyötyä reaalimaailman kompleksisuuden vähentämisestä. Mielenkiintoista olisi nähdä, voiko liiketoiminnan kompleksisuutta vähentää optimoimalla DCI-arkkitehtuuria ja toteuttamalla arkkitehtuuriin tehdyt optimoinnit liiketoiminnassa.

Viiteluettelo

[Agile Manifesto, 2001] Agile Manifesto, Manifesto for Agile Software Development. Available as http://www.agilemanifesto.org/. Checked 4.12.2015.

[Seibel & Armstrong, 2009] Peter Seibel and Joe Armstrong, *Coders at Work: Reflections on the Craft of Programming*, Apress, 2009, 205-240.

[Buckley, 2015] Kerry Buckley, Manifesto for Half-Arsed Agile Software Development. Available as http://www.halfarsedagilemanifesto.org/. Checked 4.12.2015.

[Boehm et al., 1976] B. W. Boehm, J. R. Brown and M. Lipow, Quantitative evaluation of software quality. In: *Proc. 2nd International Conference on Software Engineering*, IEEE, 1976, 592-605.

[Cardelli, 1996] Luca Cardelli, Bad Engineering Properties of Object-Oriented Languages. *ACM Computing Surveys* **28**, 4 (Dec 1996), 150.

[Chhabra & Gupta, 2009] Jitender Kumar Chhabra and Varun Gupta, Evaluation of Object-Oriented Spatial Complexity Measures. *ACM SIGSOFT Software Engineering Notes* **34**, 3 (May 2003), 1-5.

[Coplien, 2012] James O. Coplien, Objects of the People, By the People, and For the People. In: *Proc. 11th Annual International Conference on Aspect-Oriented Software Development Companion*, ACM, 2012, 3-4.

[Coplien & Bjørnvig, 2010] James Coplien and Gertrud Bjørnvig, *Lean Architecture for Agile Software Development*, John Wiley & Sons Ltd, 2010.

[Gamma et al., 1994] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GitHub, 2015] Miikka Kähkönen, DCI-arkkitehtuurin peliesimerkki. Saatavilla osoitteesta https://github.com/MiikkaKahkonen/DCI-peliesimerkki.git. Tarkistettu 4.12.2015.

[Gupta & Chhabra, 2009] Varun Gupta and Jitender Kumar Chhabra, Object-Oriented Cognitive-Spatial Complexity Measures. *World Academy of Science, Engineering and Technology* **3**, 2 (2009), 122-129.

[Gupta & Chhabra, 2011] Varun Gupta and Jitender Kumar Chhabra, Dynamic cohesion measures for object-oriented software. *Journal of Systems Architecture* **57**, 2011, 452-462.

[Hayata et al., 2012] Tomohiro Hayata, Jianchao Han and Mohsen Beheshti, Facilitating Agile Software Development with Lean Architecture in the DCI Paradigm. In: *Proc. 9th International Conference on Information Technology: New Generations*, IEEE, 2012, 343-348.

[Hytönen, 2015] Kimmo Hytönen, Relaatiot. Saatavilla osoitteesta http://hyotynen.iki.fi/relaatiot/. Tarkistettu 14.12.2015.

[Kay, 2003] Alan Kay, On the Meaning of "Object-Oriented Programming". Available as http://www.purl.org/stefan_ram/pub/doc_kay_oop_en. Checked 4.12.2015.

[Koskimies, 2000] Kai Koskimies, *Oliokirja*, Talentum, 2000.

[Microsoft, 2015] Microsoft Developer Network, Code Metrics Values. Available as https://msdn.microsoft.com/en-us/library/bb385914.aspx. Checked 4.12.2015.

[Nielsen, 2010] Jakob Nielsen, Mental Models. Available as http://www.nngroup.com/articles/mental-models. Checked 4.12.2015.

[Poppendieck & Poppendieck, 2003] Mary Poppendieck and Tom Poppendieck, *Lean Software Development: An Agile Toolkit*, Addison-Wesley, 2003.

[Reenskaug, 2007] Trygve Reenskaug, The Case for Readable Code. *Computer Software Engineering Research*, Nova Science Publishers, 2007, 3-8.

[Reenskaug, 2009] Trygve Reenskaug, The Common Sense of Object Oriented Programming. Available as http://folk.uio.no/trygver/2009/commonsense.pdf. Checked 4.12.2015.

[Reenskaug & Coplien, 2009] Trygve Reenskaug and James O. Coplien, The DCI Architecure: A New Vision of Object-Oriented Programming. Available as http://www.artima.com/articles/dci_vision.html. Checked 4.12.2015.

[Shao & Wang, 2003] Jingqiu Shao and Wang, Yingxu, A new measure of software complexity based on cognitive weights. *Canadian Journal of Electrical and Computer Engineering* **28**, 2 (April 2003), 69-74.

[Simola, 2012] Juha Simola, Hajautettu työajanseuranta DCI-arkkitehtuuria käyttäen. Diplomityö, Tampereen teknillinen yliopisto, 2012.

[Sommerville, 2011] Ian Sommerville, *Software engineering*, 9th ed. Addison-Wesley, 2011.

[Strategos, 2015] Strategos inc, Toyota Production System (TPS) & Lean. Available as http://www.strategosinc.com/toyota_production.html. Checked 4.12.2015.

[Unity, 2015] Unity Official Documentation. Available as http://docs.unity3d.com/Manual/index.html. Checked 4.12.2015.

Liitteet

A. Dynaaminen yhtenäisyys

Guptan ja Chhabran [2011] dynaaminen yhtenäisyys voidaan laskea staattisesti luokille tai dynaamisesti oliolle. Mittauksissa on erikoismetodeja, jotka voidaan jättää pois laskuista. Erikoismetodit ovat rakentaja, tuhoaja, käsittelijät ja edustajat (delegate). Mittauksissa oletetaan, että luokalla on sekä attribuutteja että metodeja. Oletus ei DCI-arkkitehtuurin suhteen toimi staattisesti, joten luokkien mittaaminen jätetään pois. Dynaaminen yhtenäisyys lasketaan siis vain olioille. Olioiden attribuuttien ja metodien väliset relaatiot määritetään taulukon 21 kaavoilla.

Attribute-Method Write Relation (AM) : $r_w^R(e_i^R \cdot e_j^R) = 1 \ \wedge \ e_i^R \in A^R(o) \ \wedge e_j^R \in M^R(o) \wedge o \ \in O(c) \wedge c \ \in C$
Method-Attribute Read Relation (MA) $r_R^R(e_i^R \cdot e_j^R) = 1 \ \wedge \ e_i^R \in M^R(o) \ \wedge e_j^R \in A^R(o) \wedge o \ \in O(c) \wedge c \ \in C$
Method-Method Call Relation (MM) $r_C^R(e_i^R \cdot e_j^R) = 1 \ \wedge \ e_i^R \in M^R(o) \ \wedge e_j^R \in M^R(o) \wedge o \ \in O(c) \wedge c \ \in C$
Attribute-Attribute Reference Relation (AA) $r_{RF}^R(e_i^R \cdot e_j^R) = 1 \ \wedge \ e_i^R \in A^R(o) \ \wedge e_j^R \in A^R(o) \wedge o \ \in O(c) \wedge c \ \in C$

Taulukko 21: Dynaamisen yhtenäisyyden relaatioiden laskukaavat [Gupta & Chhabra, 2011].

AM tarkoittaa tilannetta, jossa jokin olion o metodi m kirjoittaa kyseisen olion attribuuttiin a. MA on tilanne, jossa olion o metodi lukee kyseisen olion attribuuttia a. MM on tilanne, jossa olion o metodi kutsuu toista olion o metodia. AA on tilanne, jossa olion metodin sisällä käytetään sen kahta eri attribuuttia. Ominaisuuksien laskuun liittyy painoarvo, jolla kyseisten ominaisuuksien summa kerrotaan. Painot tutkielmassa ovat 4, 3, 2 ja 1 järjestyksessä ominaisuuksille AM, MA, MM ja AA. Painojen arvot tulisi määrittää järjestelmän oma asiantuntija, mutta oleellisesti painoarvojen suuruuksien tulee olla järjestyksessä AM > MA > MM > AA. [Gupta & Chhabra, 2011]

Oliolle lasketaan dynaaminen yhtenäisyys aina jossain skenaariossa. Skenaario on yksinkertaisesti käyttötilanne, esimerkiksi ohjelman aloitus. Tutkielmassa kontekstien läpikäynnit olivat skenaarioita. Olioille lasketaan jokaisesta skenaariosta taulukon 22 mukaiset arvot.

m = |A(o)| eli olion o kaikkien attribuuttien määrä. n = |M_n(c)| eli olion o kaikkien normaalien metodien määrä.
$DC\_AM_x(o) = \begin{cases} 0; jos\ m = 0\ \vee n = 0 \\ \dfrac{\sum_{i=1}^{m}\sum_{j=1}^{n} r_W^R\left(e_i^R \cdot e_j^R\right)}{m \times n} \end{cases}; missä\ e_i^R \in A(0) \wedge e_j^R \in M_n(o) \wedge o \in O(c)$
DC_AM on skenaariossa x ilmentyneiden AM-relaatioiden suhde mahdollisiin AM-relaatioihin.
$DC\_MA_x(o) = \begin{cases} 0\ ; jos\ n = 0\ \vee m = 0 \\ \dfrac{\sum_{i=1}^{m}\sum_{j=1}^{n} r_R^R\left(e_i^R \cdot e_j^R\right)}{n \times m} \end{cases}; missä\ e_i^R \in\ M_{n(o)} \wedge e_j^R \in A(o) \wedge o \in O(c)$
DC_MA on skenaariossa x ilmentyneiden MA-relaatioiden suhde mahdollisiin MA-relaatioihin.
$DC\_MM_x(o) = \begin{cases} 0\ ; jos\ n = 0 \\ \dfrac{\sum_{i=1}^{m}\sum_{j=1 \wedge j\neq i}^{n} r_C^R\left(e_i^R \cdot e_j^R\right)}{n \times (n-1)}; missä\ e_i^R \in\ M_n(o) \wedge e_j^R \in M_n(o) \wedge o \in O(c) \\ 1\ ; jos\ n = 1 \end{cases}$
DC_MM on skenaariossa x ilmentyneiden MM-relaatioiden suhde mahdollisiin MM-relaatioihin.
$DC\_AA_x(o) = \begin{cases} 0\ ; jos\ m = 0 \\ \dfrac{\sum_{i=1}^{m-1}\sum_{j=1-1}^{m} r_{RF}^R\left(e_i^R \cdot e_j^R\right)}{n \times m \times (m-1)/2}; missä\ e_i^R \in\ A(o) \wedge e_j^R \in A(o) \wedge o \in O(c) \\ 1\ ; jos\ m = 1 \end{cases}$
DC_AA on skenaariossa x ilmentyneiden AA-relaatioiden suhde mahdollisiin AA-suhteisiin.
$DOC_x(o) = \dfrac{w1 * DC\_AM_x(o) + w2 * DC\_MA_x(o) + w3 * DC\_MM_x(o) + w4 * DC\_AA_x(o)}{w1 + w2 + w3 + w4}$
$DOC_x(o)$ on olion o kokonaisyhtenäisyys skenaariossa x.

$DOC(o_i) = \dfrac{\left(\sum_{i=1}^{\lvert X\rvert} DOC_x(o_i)\right)}{\lvert X\rvert}$	$DOC(o_i)$ on olion o kokonaisyhtenäisyys kaikkien skenaarioiden kesken.

Taulukko 22: Dynaamisen yhtenäisyyden laskukaavat [Gupta & Chhabra, 2011].

B. Kognitiivis-spatiaalinen kompleksisuus

Guptan ja Chhabran [2009] kognitiivis-spatiaalinen kompleksisuus voidaan laskea sekä luokille että olioille. Luokan kompleksisuus (class cognitive-spatial complexity, CCSC), muodostetaan tutkimalla luokan attribuuttien ja metodien käyttöä luokan metodien sisällä. Olion kompleksisuus (object cognitive-spatial complexity, OCSC) muodostetaan tutkimalla olion muodostamista ja sen alkioiden käyttöä.

CCSC lasketaan summaamalla attribuuttien kompleksisuus (attribute cognitive-spatial complexity, ACSC), ja metodien kompleksisuus (method cognitive-spatial complexity, MCSC). ACSC lasketaan erikseen jokaiselle luokan attribuutille ja itse luokan ACSC on näiden tulosten keskiarvo. Metodien suhteen tehdään sama. OCSC lasketaan summaamalla olion alustusten kompleksisuus (object definition cognitive-spatial complexity, ODCSC) ja olion alkioiden käytön kompleksisuus (object member usage cognitive-spatial complexity, OMUCSC). Etäisyyttä laskettaessa alimmalla tasolla se painotetaan kontrollirakenteen painoarvolla ja mikäli kontrollirakenteet ovat sisäkkäin, ne summataan. Laskukaavat löytyvät taulukosta 23. [Gupta & Chhabra, 2009]

Kontrollirakenne (BCS)	Paino	Huomioitavaa
Sekvenssi	1	Yksinkertainen komentorivi
Yhden tapauksen ehtolause	2	if-else
Usean tapauksen ehtolause	3	Case, sisäkkäiset if-else if
Iteraattorit	3	for, while jne.
Funktiokutsu	2	Vain käyttäjän määrittämät kutsut
Rekursio	3	
Rinnakkaisuus ja Keskeyttäjät	4	
$Distance(i,k)$	i on attribuutti, metodi tai olion alkio ja k on rivinumero. Distance(i,k) on i:n aiemman käyttötilanteen rivinumeron, tai jos aiempaa ei ole, niin sen määritelmän rivinumeron, ja rivinumero k:n absoluuttinen etäisyys koodiriveinä. Mikäli rivit eivät ole samassa tiedostossa, Distance(i,k) = (etäisyys käsiteltävän tiedoston i:n ensimmäisestä käytöstä tiedoston alkuun) + (etäisyys i:n määrittämisen tiedostosta i:n määrittämisen riviin).	
$ACSC(i,k) = W_k * Distance(i,k)$	i on attribuutti ja k rivinumero. ACSC(i,k) on attribuutin i käyttötilanne rivillä k painotettuna	

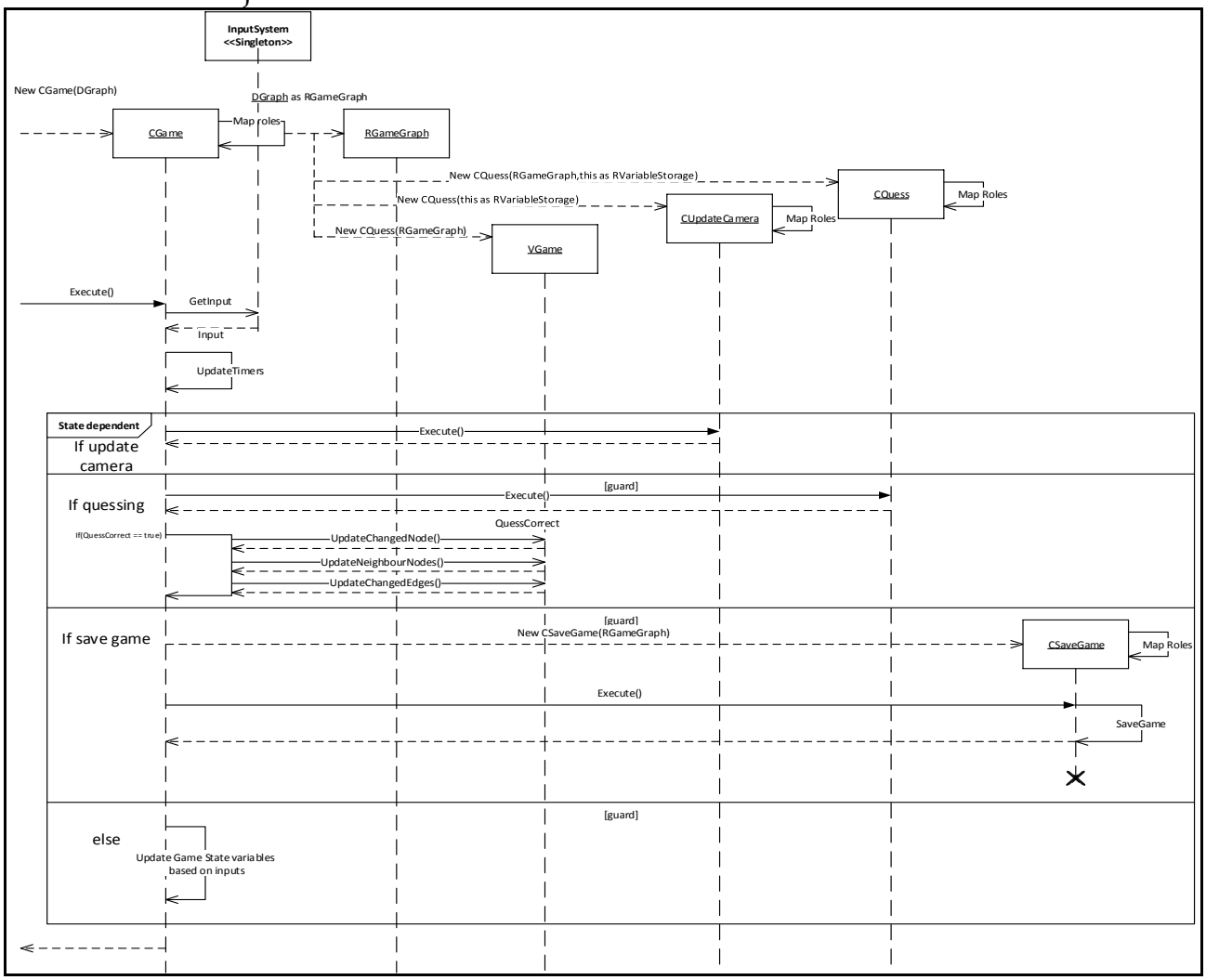
	rivin k käytön kontrollirakenteen (BCS) painoarvolla Wk.
$ACSC_i = \left(\frac{\sum_{k=1}^{p} ACSC(i,k)}{p}\right)$	Attribuutin i kompleksisuus on sen käyttöjen keskiarvo.
$\text{CACSC} = \frac{(\sum_{i=1}^{q} ACSC_i)}{q}$	Luokan attribuuttien monimutkaisuus on sen attribuuttien monimutkaisuuksien keskiarvo.
$MCSC(i,k) = W_k * Distance(i,k)$	Metodin i käytön kompleksisuus rivillä k. Metodin aikaisemman kutsun tai määritelmän etäisyys rivistä k kerrottuna käytön kontrollirakenteen painoarvo. Jos rivit ovat eri tiedostoissa, sama toiminta kuin attribuutin kohdalla metodin suhteen.
$MCSC_i = \frac{\sum_{k=1}^{m} MCSC\ (i,k)}{m}$	Metodin i kompleksisuus on sen käyttöjen kompleksisuuksien keskiarvo.
$CMCSC = \frac{\sum_{i=1}^{n} MCSC_i}{n}$	Luokan metodien kompleksisuus on sen metodien kompleksisuuksien keskiarvo.
$CCSC = CACSC + CMCSC$	Luokan kompleksisuus on sen attribuuttien kompleksisuuden ja metodien kompleksisuuden summa.
$ODCSC(i) = W_k * Distance(i,k)$	Olion luomisen kompleksisuus ODCSC(i) on sen luomisen kontrollirakenteen painoarvo kerrottuna sen etäisyydellä luomisrivistä k olion luokan määritelmään.
$OMUCSC(i,k) = W_k * Distance(i,k)$	Olion alkion i käytön kompleksisuus OMUCSC on sen käyttörivin k etäisyys aiemmasta käytöstä tai määritelmästä painotettuna sen kontrollirakenteen painoarvolla.
$OMUCSC_i = \frac{(\sum_{k=1}^{m} OMUCSC(i,k))}{m}$	Olion alkion i kompleksisuus on sen käyttöjen kompleksisuuksien keskiarvo.
$OMCSC = \frac{\sum_{i=1}^{n} OMCSC_i}{n}$	Olion alkioiden kompleksisuus on sen alkioiden kompleksisuuksien keskiarvo.
$OCSC = ODCSC + OMCSC$	Olion monimutkaisuus on sen luomisen ja käytön monimutkaisuuksien summa.

Taulukko 23: Kognitiivis-spatiaalisen kompleksisuuden laskukaavat [Gupta & Chhabra, 2009].

C. Esimerkkipelin arvioitavan osuuden luokat, attribuutit ja metodit.

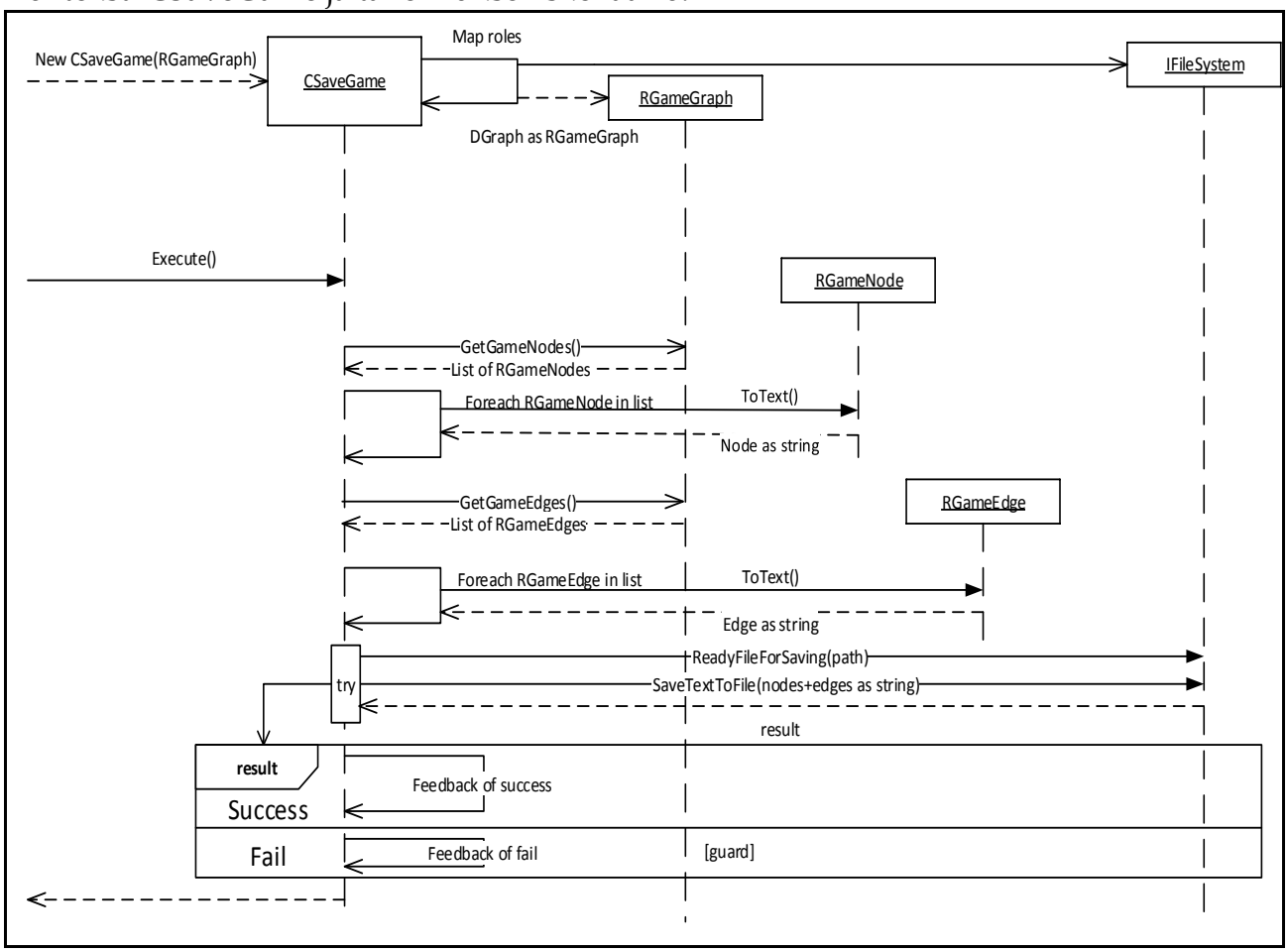
D. Kontekstien ja käyttötapauksien skenaarioiden loput vertailut

Konteksti CGame ja aloituksen skenaario.



Skenaario		
1.	Päivitä pelin ajastimet	
2.	Tarkista missä toiminnassa peli on:	
	a.	Pelaaja on arvaamassa -> päivitä komennot ja siirry pelin arvaamis Use Caseen (K3)
	b.	Pelaaja liikuttaa kameraa -> päivitä komennot ja siirry pelikameran liikuttamis Use Caseen (K4)
	c.	Pelaaja on lopettamassa peliä -> päivitä komennot ja siirry pelin lopetus Use Caseen (5)
	d.	Peli halutaan tallentaa -> aloita pelin tallennus Use Case (K2)
	e.	Pelaaja ei antanut komentoja
3.	Pelin toiminta siirretään UnityEnginelle, joka tekee omat päivityksensä kyseiselle framelle	

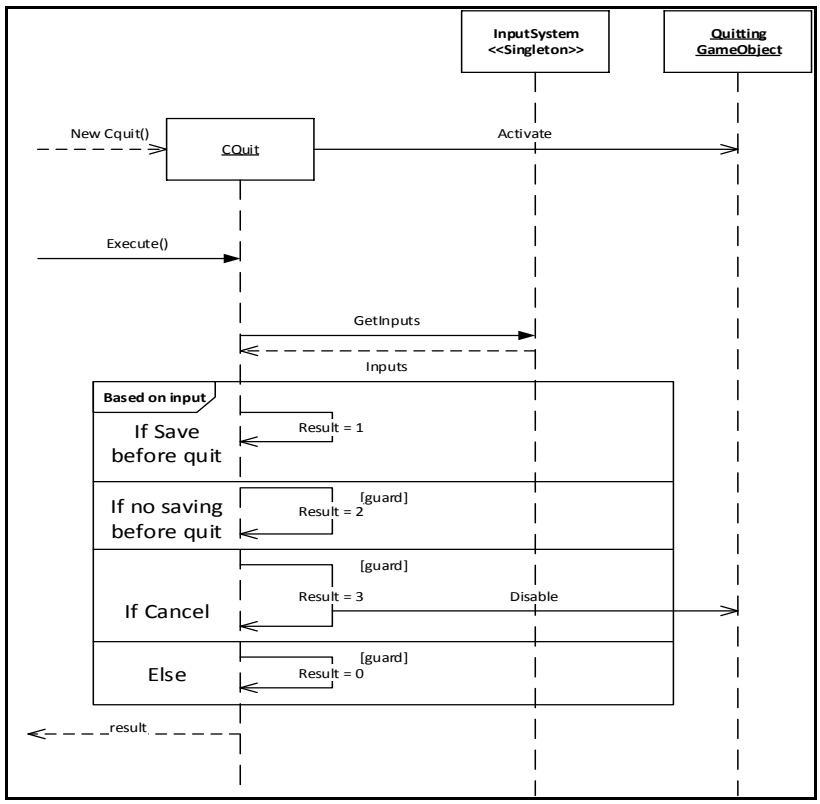
Konteksti CSaveGame ja tallennuksen skenaario.



Skenaario	
1.	Käyttäjää antaa tallennuskomennon
2.	Peli tallentaa pelitilanteen tiedostoon
3.	Peli ilmoittaa tallentumisen lopputuloksesta pelaajalle
4.	Peli odottaa uusia komentoja

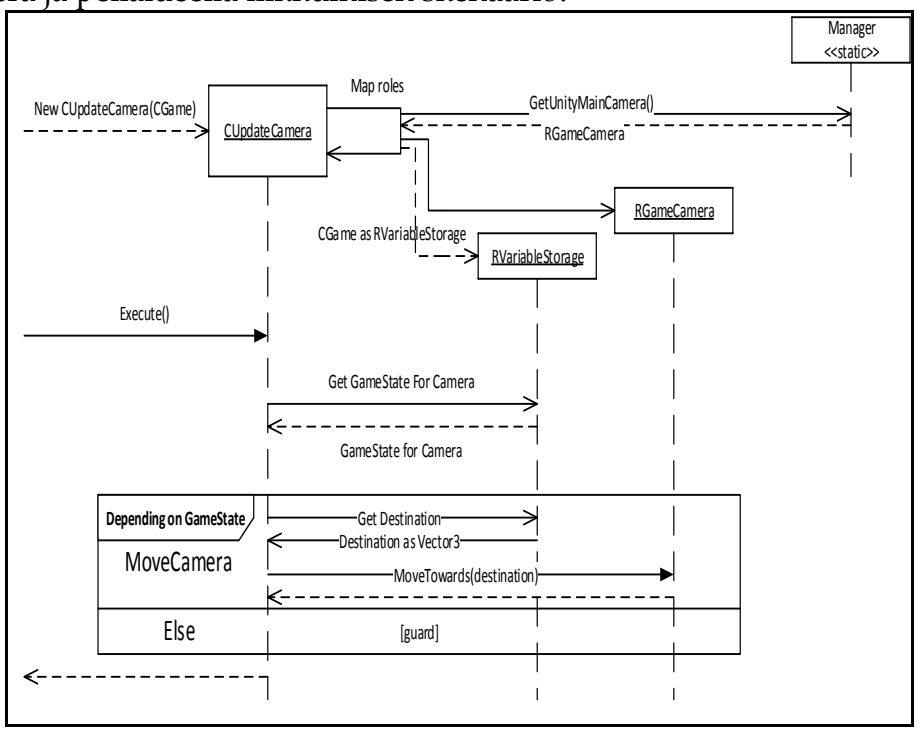
Konteksti CQuit ja lopetuksen skenaario.

Skenaario		
1.		Käyttäjä antaa pelin sulkemiskomennon
2.	a	Mikäli peliä ei ole tallennettu viimeisen muutoksen jälkeen, peli kysyy, halutaanko peli sulkea tallentamatta
	b	Peli aloittaa sulkeutumisen
3.		Peli käyttää UnityEnginen sulkemiskomentoa, joka huolehtii tarvittavista sulkeutumiseen liittyvissä asioista
4.		Peli on suljettu



Konteksti CUpdateCamera ja pelialueella liikkumisen skenaario.

Skenaario	
1.	Pelaaja valitsee suunnan ja nopeuden mihin ja kuinka nopeasti näkymää siirretään
2.	Pelin näkymä siirtyy kyseiseen suuntaan
3.	Peli odottaa käyttäjän komentoja



E. Yleisten metriikoiden tulostaulukot

Luokkat

Type	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code(Il)
VPanelMono	80	6	5	8	8
VNodeMono	82	8	5	7	14
VExitButtonMono	100	1	5	1	1
VEdgeMono	83	9	5	6	17
CQuess	47	16	2	12	61
DNode	92	9	2	3	14
DGraph	92	7	2	9	13
DEdge	94	5	2	3	7
DCamera	94	3	2	4	5
VQuess	68	8	2	6	19
CGame	59	48	2	33	166
CLoadGame	67	4	2	8	15
VCamera	100	1	2	1	1
CUpdateCamera	73	3	2	10	8
CSaveGame	57	7	2	18	24
CQuit	52	17	2	14	39
VGame	59	26	2	23	60
RTGameNode	69	3	1	4	17
RTVariableStorage	75	20	1	4	35
RTGameGraph	53	48	1	22	106
RTGameGamera	70	2	1	5	5
BaseContext	90	3	1	1	3
NodePair	86	3	1	1	7
View	100	1	1	0	1
Data	91	5	1	0	8
RTGameEdge	65	1	1	2	8
RGameGraph	100	6	0	5	1
RGameItem	100	2	0	0	1
RGameNode	100	7	0	1	1
RVariableStorage	100	10	0	2	1
RGameEdge	100	4	0	1	1
RGameCamera	100	2	0	2	1

Metodit

Type	Member	Maintainability Index	Cyclomatic Complexity	Class Coupling	Lines of Code (Il)
VPanelMono	Update() : void	74	3	2	4
VPanelMono	SetColor(Color) : void	76	2	6	3
VPanelMono	VPanelMono()	94	1	1	1
VNodeMono	Start() : void	83	1	3	2
VNodeMono	Tm.get() : TextMesh	73	2	3	5

VNodeMono	GetPosition() : Vector3	84	1	4	2
VNodeMono	SetPosition(Vector3) : void	92	1	4	1
VNodeMono	SetWord(string) : void	94	1	1	1
VNodeMono	EnableRenderer(bool) : void	82	1	3	2
VNodeMono	VNodeMono()	94	1	1	1
VExitButtonMono	VExitButtonMono()	100	1	1	1
VEdgeMono	Start() : void	83	1	3	2
VEdgeMono	Lr.get() : LineRenderer	73	2	3	5
VEdgeMono	EnableRenderer(bool) : void	94	1	1	1
VEdgeMono	GetStart() : Vector3	91	1	1	2
VEdgeMono	SetStart(Vector3) : void	84	1	2	2
VEdgeMono	GetEnd() : Vector3	91	1	1	2
VEdgeMono	SetEnd(Vector3) : void	84	1	2	2
VEdgeMono	VEdgeMono()	94	1	1	1
CQuess	CQuess(RGameGraph, InputSystem, RVariableStorage)	66	1	5	8
CQuess	Execute() : string	39	15	12	53
DNode	Word.get() : string	91	1	0	2
DNode	Word.set(string) : void	95	1	0	1
DNode	State.get() : int	91	1	0	2
DNode	State.set(int) : void	95	1	0	1
DNode	EWord.get() : string	91	1	0	2
DNode	EWord.set(string) : void	80	2	0	3
DNode	ESet.get() : bool	91	1	0	2
DNode	DNode()	94	1	1	1
DGraph	DGraph()	79	1	3	3
DGraph	Nodes.get() : GameNodeList	91	1	1	2
DGraph	Nodes.set(GameNodeList) : void	95	1	1	1
DGraph	Edges.get() : GameEdgeList	91	1	1	2
DGraph	Edges.set(GameEdgeList) : void	95	1	1	1
DGraph	NewNode.get() : RGameNode	89	1	2	2
DGraph	NewEdge.get() : RGameEdge	89	1	2	2
DEdge	HeadId.get() : string	91	1	0	2
DEdge	HeadId.set(string) : void	95	1	0	1
DEdge	TailId.get() : string	91	1	0	2
DEdge	TailId.set(string) : void	95	1	0	1
DEdge	DEdge()	100	1	1	1
DCamera	DCamera(Camera)	87	1	2	2
DCamera	Cam.get() : Camera	91	1	1	2
DCamera	Cam.set(Camera) : void	95	1	1	1

VQuess	VQuess()	64	3	5	8
VQuess	UpdateText(string) : void	95	1	1	1
VQuess	ColorFeedback(int) : void	64	4	2	10
CGame	CGame(RGameGraph, Dictionary<string, Vector3>, InputSystem)	59	1	8	13
CGame	Execute() : void	33	28	30	71
CGame	GetRandomNodePosition() : Vector3	69	1	2	6
CGame	UpdateTimers() : void	52	7	2	19
CGame	InitVars() : void	44	1	2	42
CGame	BoolVariables.get() : Dictionary<string, bool>	91	1	1	2
CGame	BoolVariables.set(Dictionary<string, bool>) : void	95	1	1	1
CGame	FloatVariables.get() : Dictionary<string, float>	91	1	1	2
CGame	FloatVariables.set(Dictionary<string, float>) : void	95	1	1	1
CGame	IntVariables.get() : Dictionary<string, int>	91	1	1	2
CGame	IntVariables.set(Dictionary<string, int>) : void	95	1	1	1
CGame	Vector3Variables.get() : Dictionary<string, Vector3>	91	1	2	2
CGame	Vector3Variables.set(Dictionary<string, Vector3>) : void	95	1	2	1
CGame	Timers.get() : Dictionary<string, float>	91	1	1	2
CGame	Timers.set(Dictionary<string, float>) : void	95	1	1	1
CLoadGame	CLoadGame(RGameGraph, Dictionary<string, Vector3>, BaseFileSystem)	76	1	5	4
CLoadGame	Execute() : void	61	3	8	11
VCamera	VCamera()	100	1	1	1
CUpdateCamera	CUpdateCamera(RVariableStorage)	74	1	7	4
CUpdateCamera	Execute() : void	74	2	6	4
CSaveGame	CSaveGame(RGameGraph, BaseFileSystem, Dictionary<string, Vector3>)	76	1	5	4
CSaveGame	Execute() : void	50	6	18	20
CQuit	CQuit(InputSystem)	59	6	7	11
CQuit	Execute() : int	47	11	12	28
VGame	VGame(RGameGraph, Dictionary<string, Vector3>)	50	5	18	20
VGame	UpdateNode(RGameNode) : void	64	4	4	8
VGame	UpdateEdge(RGameNode, RGameNode) : void	50	11	5	21
VGame	UpdateNodes(List<RGameNode>) : void	78	2	4	3

VGame	UpdateEdges(List<Node Pair>) : void	77	2	4	3
VGame	GetNodePositions() : Dictionary<string, Vector3>	70	2	6	5
RTGameNode	SetEWord(this RGameNode) : void	68	1	3	5
RTGameNode	ToTexT(this RGameNode) : string	61	1	3	10
RTGameNode	CompaireQuess(this RGameNode, string) : bool	84	1	2	2
RTVariableStorage	BoolVar(this RVariableStorage, string) : bool	75	2	3	4
RTVariableStorage	BoolVar(this RVariableStorage, string, bool) : void	77	2	3	3
RTVariableStorage	FloatVar(this RVariableStorage, string) : float	75	2	3	4
RTVariableStorage	FloatVar(this RVariableStorage, string, float) : void	77	2	3	3
RTVariableStorage	IntVar(this RVariableStorage, string) : int	75	2	3	4
RTVariableStorage	IntVar(this RVariableStorage, string, int) : void	77	2	3	3
RTVariableStorage	Vector3Var(this RVariableStorage, string) : Vector3	75	2	4	4
RTVariableStorage	Vector3Var(this RVariableStorage, string, Vector3) : void	77	2	4	3
RTVariableStorage	TimerVar(this RVariableStorage, string) : float	75	2	3	4
RTVariableStorage	TimerVar(this RVariableStorage, string, float) : void	77	2	3	3
RTGameGraph	CreateGraph(this RGameGraph, List<string>, Dictionary<string, Vector3>) : bool	34	31	19	62
RTGameGraph	CreateNode(this RGameGraph, string, string, CGame.DCINodeState) : RGameNode	69	1	6	7
RTGameGraph	CreateEdge(this RGameGraph, string, string) : RGameEdge	70	1	4	6
RTGameGraph	GetNextNodePosition(this RGameGraph) : Vector3	69	1	4	6
RTGameGraph	Quess(this RGameGraph, string) : string	66	5	10	7
RTGameGraph	UpdateNeighbours(this RGameGraph, string) : List<RGameNode>	52	9	12	18

RTGameGamera	MoveTowards(this RGameCamera, Vector3) : bool	70	2	5	5
BaseContext	Execute() : void	89	1	1	1
BaseContext	Done() : void	89	1	1	1
BaseContext	BaseContext()	100	1	0	1
NodePair	NodePair(RGameNode, RGameNode)	81	1	1	3
NodePair	Head.get() : RGameNode	91	1	1	2
NodePair	Tail.get() : RGameNode	91	1	1	2
View	View()	100	1	0	1
Data	Id.get() : string	91	1	0	2
Data	Id.set(string) : void	95	1	0	1
Data	GetNextId() : string	78	1	0	3
Data	Data()	100	1	0	1
Data	Data()	95	1	0	1
RTGameEdge	ToTexT(this RGameEdge) : string	65	1	2	8
RGameGraph	Nodes.get() : GameNodeList	100	1	1	1
RGameGraph	Nodes.set(GameNodeList) : void	100	1	1	1
RGameGraph	Edges.get() : GameEdgeList	100	1	1	1
RGameGraph	Edges.set(GameEdgeList) : void	100	1	1	1
RGameGraph	NewNode.get() : RGameNode	100	1	1	1
RGameGraph	NewEdge.get() : RGameEdge	100	1	1	1
RGameItem	Id.get() : string	100	1	0	1
RGameItem	Id.set(string) : void	100	1	0	1
RGameNode	Word.get() : string	100	1	0	1
RGameNode	Word.set(string) : void	100	1	0	1
RGameNode	State.get() : int	100	1	0	1
RGameNode	State.set(int) : void	100	1	0	1
RGameNode	EWord.get() : string	100	1	0	1
RGameNode	EWord.set(string) : void	100	1	0	1
RGameNode	ESet.get() : bool	100	1	0	1
RVariableStorage	BoolVariables.get() : Dictionary<string, bool>	100	1	1	1
RVariableStorage	BoolVariables.set(Dictionary<string, bool>) : void	100	1	1	1
RVariableStorage	FloatVariables.get() : Dictionary<string, float>	100	1	1	1
RVariableStorage	FloatVariables.set(Dictionary<string, float>) : void	100	1	1	1
RVariableStorage	IntVariables.get() : Dictionary<string, int>	100	1	1	1
RVariableStorage	IntVariables.set(Dictionary<string, int>) : void	100	1	1	1
RVariableStorage	Vector3Variables.get() : Dictionary<string, Vector3>	100	1	2	1

RVariableStorage	Vector3Variables.set(Dictionary<string, Vector3>) : void	100	1	2	1
RVariableStorage	Timers.get() : Dictionary<string, float>	100	1	1	1
RVariableStorage	Timers.set(Dictionary<string, float>) : void	100	1	1	1
RGameEdge	HeadId.get() : string	100	1	0	1
RGameEdge	HeadId.set(string) : void	100	1	0	1
RGameEdge	TailId.get() : string	100	1	0	1
RGameEdge	TailId.set(string) : void	100	1	0	1
RGameCamera	Cam.get() : Camera	100	1	1	1
RGameCamera	Cam.set(Camera) : void	100	1	1	1