Anna Yli-Sipilä

Riippuvuusinjektio

Joustavuutta arkkitehtuuriin löyhillä sidoksilla

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

12.4.2013

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Anna Yli-Sipilä Riippuvuusinjektio: Joustavuutta arkkitehtuuriin löyhillä sidoksilla 51 sivua + 5 liitettä 12.4.2013
Tutkinto	insinööri (AMK)
Koulutusohjelma	tietotekniikka
Suuntautumisvaihtoehto	ohjelmistotekniikka
Ohjaaja(t)	Senior Team Leader Vesa Tuomala yliopettaja Auvo Häkkinen

Insinöörityön tavoitteena oli selvittää riippuvuusinjektion periaate, antaa esimerkkejä sen hyödyntämisestä sekä esitellä toteuttamisen avuksi vapaasti saatavilla olevia kolmannen osapuolen ohjelmistokehyksiä. Työ toteutettiin tutkimalla aiheeseen liittyvää kirjallisuutta ja toteuttamalla esimerkkisovellus, jossa ratkaistiin riippuvuuksista johtuva testattavuuden ongelma riippuvuusinjektiota käyttäen. Työn motivaationa oli testattavuuden parantaminen.

Riippuvuusinjektion periaatetta käsiteltiin yleisesti hyväksi todettujen sovelluskehitysten suunnitteluperiaatteiden kautta. Esitellyt suunnitteluperiaatteet olivat SOLID-akronyymin mukaiset avoin-suljettu-periaate, yhden vastuualueen periaate, Liskovin korvaavuusperiaate, rajapintojen erotteluperiaate ja riippuvuuden kääntöperiaate. Todettiin, että näiden periaatteiden noudattaminen tarkoittaa löyhien liitosten suosimista ja riippuvuusinjektion avulla mahdollistetaan löyhät liitokset.

Riippuvuusinjektion hyödyistä esiteltiin testattavuus, myöhäinen sidonta, laajennettavuus, ylläpidettävyys ja yhtäaikainen kehitys. Näistä esimerkkisovelluksen avulla havainnollistettiin testattavuutta. Toteutustavoista käytiin läpi alustaja-, setteri- ja metodi-injektio sekä ympäröivän kontekstin käsite.

Riippuvuusinjektion toteuttamisella oli huomattava merkitys testattavuuden parantumiseen. Tämän lisäksi esimerkkisovelluksen arkkitehtuuri parani noudattamaan vahvemmin olio-ohjelmoinnille ominaista kapselointia. Esimerkkisovellukseen ei käytetty kolmannen osapuolen ohjelmistokehyksiä, koska niillä ei saatu riittävää lisähyötyä.

Avainsanat	riippuvuusinjektio, inversion of control, testattavuus, suunnitteluperiaatteet

Author(s) Title Number of Pages Date	Anna Yli-Sipilä Dependency Injection: Flexible Architecture with Loose Coupling 51 pages + 5 appendices 12 April 2013
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Vesa Tuomala, Senior Team Leader Auvo Häkkinen, Principal Lecturer

The purpose of this research was to give insight to dependency injection through usage examples and by showcasing generally available third party frameworks. Dependency injection as a design principle in comparison with, and sometimes referred as a synonym to, inversion of control was studied based on literature and discussion found on the subject. An example project was used to show how dependency injection could be used to enable testability. Improving testability was the motivation behind this study.

Generally highly valued design principles were used to clarify the motivation behind dependency injection. These design principles come from the acronym SOLID: Open-Closed Principle, Single Responsibility Principle, Liskov Substitution Principle, Interface Segregation Principle and Dependency-Inversion Principle. It was acknowledged that loose coupling is the key to implementing these design principles and dependency injection therefore a means to gain loose coupling.

Testability, late binding, extensibility, maintainability and parallel development were shown as the benefits of dependency injection. Of those testability was further studied in the example project. Constructor, setter and method injection were detailed as the implementation modes of dependency injection alongside ambient context.

Testability was greatly improved by implementing dependency injection principles. Furthermore, the whole architecture of the example project was amended to enable stricter encapsulation. No third party dependency injection containers were used in the example project as they did not offer sufficient extra value.

Keywords	Dependency Injection, Inversion of Control, testability, design principles

Sisällys

Lyhenteet

API	Ohjelmointirajapinta (engl. Application Programming Interface). Ohjelmointirajapinta on julkinen rajapinta tai kokoelma rajapintoja, joiden kautta sovellukset voivat keskustella keskenään.
DIP	Riippuvuuden käännösperiaate (engl. Dependency-Inversion Principle). Suunnitteluperiaate, jonka mukaan korkeamman tason moduulien ei tule riippua alemman tason moduuleista. Samoin abstraktioiden ei tule riippua konkreettisista toteutuksista [13].
IOC	Hallinnan muutossuunnan vaihtaminen (eng. Inversion of Control). Vastuu instanssien luonnista on käänteinen.
ISP	Rajapinnan erotteluperiaate (engl. Interface Segregation Principle). Rajapintojen pitää olla tarpeeksi pieniä ja erillisiä, jotta toteuttajien ei tarvitse toteuttaa kuin itselleen relevantteja toiminnallisuuksia.
LSP	Liskovin korvaavuusperiaate (engl. Liskov Substitution Principle). Yliluokan A aliluokka B pitää voida korvata yliluokalla A.
OCP	Avoin-suljettu-periaate (engl. Open Closed Principle). Luokkien pitää olla avoimia laajentumaan, mutta suljettuja muuttumaan.
SRP	Yhden vastuualueen periaate (engl. Single Responsibility Principle). Jokaisella luokalla pitää olla vain yksi vastuualue.
TDD	Testilähtöinen sovelluskehitys (engl. Test Driven Development). Sovelluskehitystekniikka, jossa tuotantokoodin toiminnallisuus määritetään testien kautta.

Sanasto

Annotaatio — Annotaatiot ovat tapa lisätä ohjelmistokoodiin metatietoa. Niiden avulla voidaan välittää tietoa kääntäjälle (esimerkiksi @SuppressWarnings(”value = deprecated”)), myös ajonaikaisesti. Annotaatiot merkitään @-notaatiolla.

Injektori — Injektori on riippuvuusinjektiossa käytettävä taho, joka injektoi riippuvuuden sen tarvitsijalle. Kuluttajan ja palvelun tapauksessa injektori on kolmas osapuoli, joka luo palvelun ja välittää sen edelleen kuluttajalle.

Kytkös — Kytkös tarkoittaa sovelluksen luokkakaavioon merkittyä kahden luokan välistä suhdetta. Esimerkiksi luokka A käyttää luokka B:tä, joten niiden välillä on kytkös.

Komponentti

Tässä dokumentissa komponentilla tarkoitetaan jotakin sovelluksen itsenäistä osaa. Se voi olla joko palvelu, kuluttaja tai injektori tai muu selkeästi rajattavissa oleva osa.

Kapselointi — Kapselointi (engl. encapsulation) tarkoittaa olio-ohjelmoinnissa sitä, että yksittäinen luokka käärii sisäänsä omat ominaisuutensa ja pyrkii suojelemaan niitä ulkopuolisilta muutoksilta.

Kuluttaja — Kuluttaja on sovelluksen arkkitehtuurissa palvelua hyväksikäyttävä osapuoli. Riippuvuusinjektio pyrkii vähentämään kuluttajan vastuuta palvelun elinkaaresta.

Laiska initialisointi

Laiska initialisointi tarkoittaa sitä, että olio luodaan vasta, kun sitä ensimmäisen kerran tarvitaan. Vastakohta tälle on niin sanottu innokas initialisointi, jossa oliot luodaan kerralla, vaikka niitä ei välttämättä käytettäisikään.

Löyhä sidos
: Löyhä sidos (engl. loose coupling) tarkoittaa sitä, että yksittäisellä komponentilla on vain vähäinen tieto sovelluksen muista komponenteista ja se pystyy näin ollen toimimaan itsenäisemmin.

Matkija
: Matkijat (engl. mock services) ovat sellaisia palveluimplementaatioita, jotka toteuttavat todellisen palvelun kanssa saman rajapinnan, mutta eivät todellista toiminnallisuutta. Näin esimerkiksi tekstiviestipalvelun sendMessage(String message)-metodi voi palauttaa kutsuttaessa tiedon viestin lähetyksestä ilman, että viesti todellisuudessa lähetetään.

Palvelu
: Palvelu voi olla joko sovelluksen ulkopuolinen kolmannen osapuolen tarjoama rajapinta tai sovelluksen sisäinen itsenäinen kokonaisuus, jota kuluttaja tarvitsee.

Riippuvuus
: Riippuvuus on kahden komponentin välillä oleva suhde, joka kooditasolla ilmenee viittauksina luokkien välillä. Tämä voi esimerkiksi tarkoittaa sitä, että kuluttaja saa tiedon tarvitsemastaan palvelusta omassa alustajassaan. Näin ollen kuluttajaa ei voida luoda ilman palvelua, joten kuluttaja on riippuvainen palvelusta.

Riippuvuusinjektio
: Riippuvuusinjektio on tapa, jolla riippuvuus annetaan sitä tarvitsevalle luokalle. Esimerkiksi jos luokka A on riippuvainen luokasta B, voidaan luokka A injektoida, eli välittää parametrina, luokan B alustajassa.

Service Locator
: Service Locator on suunnittelumalli, joka koostuu kuluttajasta, palvelusta sekä palveluntarjoajasta. Kuluttaja pyytää tarvitsemaansa palvelua palveluntarjoajalta kyseisen palvelun nimellä. Esimerkiksi J2EE-arkkitehtuureissa JNDI-nimipalvelu toimii Service Locator-mallin mukaisesti [9].

Sidosmoduuli

Tässä sidosmoduulilla tarkoitetaan Google Guicen konfiguroinnissa tarvittavaa apuluokkaa, jonka sisälle määritellään sovelluksessa olevat riippuvuudet. Esimerkki sidosmoduulista on luvussa yhdeksän.

1 Johdanto

Ohjelmistokehityksessä päädytään usein tilanteeseen, jossa kahden eri komponentin välille syntyy arkkitehtuuria jäykistävä, vaikeasti testattava ja toiminnallisuusmuutoksia hylkivä riippuvuussuhde. Komponentilla tarkoitetaan tässä sovelluksen arkkitehtuurin kannalta tiettyä toimintaperiaatetta noudattavaa itsenäistä osaa. Riippuvuus taas on suhde komponenttien välillä, mikä ilmenee kooditasolla viittauksina luokkien välillä. Käytännössä tämä tarkoittaa, että toinen luokka vaatii toimintaansa ensimmäisen luokan instanssia. Esimerkki tällaisesta epämielekkäästä riippuvuussuhteesta on palvelun ja kuluttajan välinen suhde.

Kuluttaja tarvitsee tiedon palvelusta hyödyntääkseen tätä. Ohjelmointiteknisellä tasolla tämä tarkoittaa kuluttajassa olevaa viittausta palveluun sekä kuluttajan tekemää palvelun luontikutsua. Vastuu palvelun luonnista, ylläpidosta sekä hävittämisestä siirtyy näin kuluttajalle. Tällainen ratkaisu aiheuttaa palvelussa mahdollisesti tapahtuvien muutosten heijastumisen suoraan kuluttajalle. Rajapintoja hyödyntämällä mahdollistetaan käytettävän palvelun muuttaminen, mutta vastuu palvelusta jää edelleen kuluttajalle. [1.]

Hallinnan muutossuunnan vaihtaminen (engl. Inversion of Control, IoC) on suunnittelumalli, joka noudattaa kuuluisaa Hollywood-periaatetta ”Älä soita meille, me soitamme sinulle”. Riippuvuusinjektio on IoC-mallin eräs toteutus, jossa kuluttajalle syötetään ulkoisen injektorin toimesta kulloinkin haluttu palvelu. Injektori on kuluttajan ja palvelun välillä toimiva taho, joka huolehtii palvelusta ja välittää sen tarvittaessa kuluttajalle. Kuluttajan vastuulla ei näin ollen ole palvelun luonti, eli IoC-mallin mukainen ”soitto”, vaan injektori ”soittaa” kuluttajalle, eli tarjoaa tämän tarvitseman palvelun. Kuluttaja voi tämän jälkeen luottaa siihen, että tarvittava palvelu on käytettävissä. Testattavuuden kannalta tästä seuraa se hyöty, että kuluttajaa voidaan testata, koska riippuvuutena injektoitava palvelu voidaan testitilanteessa korvata esimerkiksi tyhjällä toteutuksella, jolloin kuluttajaa voidaan testata eristyksissä palvelusta. [2.]

Käsitteiden riippuvuusinjektio (engl. Dependency Injection) ja hallinnan muutossuunnan vaihtaminen välille ei alan kirjallisuudessa aseteta selkeää rajaa. Voidaan kuitenkin ajatella, että IoC kuvaa yleisellä tasolla sellaista arkkitehtuuriratkaisua, jossa vastuu tietystä tahosta, esimerkiksi instanssien luonnista, on käänteinen. Riippuvuusinjektion

voidaan täten ajatella olevan yksi IoC-toteutus, koska siinä palveluiden instantiointi ja riippuvuuksien syöttäminen on käänteisessä järjestyksessä. [8.]

Riippuvuusinjektion toteuttaminen tarkoittaa yksinkertaisimmillaan riippuvuuksien siirtämistä luokan sisältä sen rajapintaan. Sen sijaan, että uusia instansseja tarvittavista palveluluokista luotaisiin kuluttajan sisällä, ne voidaan suoraan välittää kuluttajalle. Tämä edistää kehitettävän sovelluksen modulaarisuutta, koska eri osapuolet voidaan selkeämmin irrottaa toisistaan. Tästä seuraa se, että riippuvuusinjektion toteuttaminen tuottaa löyhiä sidoksia. Löyhä sidos tarkoittaa sitä, että yksittäisellä komponentilla on vain vähäinen tieto sovelluksen muista komponenteista ja se pystyy näin ollen toimimaan itsenäisemmin. Löyhät sidokset ovat laajennettavuuden, testattavuuden ja muutosvalmiuden kannalta haluttu ominaisuus.

Tämän työn tarkoituksena on antaa kattava yleiskuva riippuvuusinjektion periaatteesta, hyödyntämisestä, eduista sekä implementoinnin haasteista. Työssä keskitytään erityisesti testattavuuden ylläpitämiseen riippuvuusinjektion avulla. Riippuvuuksien aiheuttamia ongelmia testattavuuteen selkeytetään esimerkkisovelluksen avulla.

2 Riippuvuuksien tuomat haasteet sovelluskehitykselle

Toiminnallisuuskeskeisyys, modulaarisuus ja testattavuus voidaan nostaa sovelluskehityksen kulmakiviksi. Toiminnallisuuskeskeisyys tarkoittaa keskittymistä määritysten mukaisen toiminnallisuuden toteuttamiseen. Modulaarisuus luo helpommin hallittavia ja ymmärrettäviä kokonaisuuksia ja parantaa testattavuutta. Testaaminen taas on ainoa tapa varmistaa, että sovellus toimii, kuten sen pitää. [8.]

Sovelluskehitys tuottaa sovelluksia yhden päämäärän mukaan eli mallintaakseen jotakin todellista toimintoa [8]. Esimerkiksi luvussa kahdeksan esitettävä yksinkertainen sovellus mallintaa toimitusjärjestelmää, joka koostuu lähetettävästä tavarasta, lähettäjästä, toimittajasta ja niitä käyttävästä käyttöliittymästä. Tässä sovelluksessa lähettäjällä on riippuvuus pakkaajaan, joka taas vaihtelee lähetettävän tavaran ominaisuuksien mukaan.

Sovelluksen eri komponenttien väliset kytkökset ovat ristiriidassa aikaisemmin mainittujen sovelluskehityksen kulmakivien kanssa. Kytkökset tarkoittavat sovelluksen luokka-

kaaviossa olevia suhteita eri luokkien välillä. Toiminnallisuuskeskeisyyden, modulaarisuuden ja testattavuuden vaaliminen on haasteellista tilanteessa, jossa käytettävät komponentit ovat tiukasti sidoksissa toisiinsa, eli riippuvuussuhteessa keskenään. Näiden haasteiden voittamiseen on onneksi käytettävissä monia hyväksi havaittuja sovelluksen suunnitteluperiaatteita. Seuraavissa luvuissa käydään tarkemmin läpi näitä suunnitteluperiaatteita. [8.]

3 Suunnitteluperiaatteet (SOLID)

Hyvän sovelluskehityksen viisi tärkeää ohjenuoraa voidaan johtaa akronyymistä SOLID. Nämä ovat vapaasti suomennettuna Yhden vastuualueen periaate (Single Responsibility Principle), avoin-suljettu-periaate (Open-Closed Principle), Liskovin korvaavuusperiaate (Liskov Substitution Principle), rajapintojen erotteluperiaate (Interface Segregation Principle) ja riippuvuuden kääntöperiaate (Dependency-Inversion Principle). Ohjenuorat ovat alun perin Robert C. Martinin toteamia, kun taas akronyymi SOLID on Michael Feathersin käsialaa. Näiden periaatteiden avulla voidaan parantaa sovelluksen testattavuutta ja laajennettavuutta. Jokainen näistä periaatteista on esitelty tarkemmin seuraavissa luvuissa. [15.]

Riippuvuusinjektiota voidaan pitää riippuvuuden kääntöperiaatteen yhtenä toteuttajana. Toisaalta jokaisen viiden periaatteen noudattaminen johdattaa käyttämään riippuvuusinjektiota. Jotta sovellus on laajennettava ja ylläpidettävä, sen osien pitää olla löyhästi sidottuina toisiinsa. Riippuvuusinjektio on tapa, jolla löyhä sidonta mahdollistetaan [10].

3.1 Avoin-suljettu-periaate (OCP)

Avoin-suljettu-periaate on suunnitteluperiaate, jonka mukaan luokkien tulee voida laajentua ilman, että olemassa olevaa toiminnallisuutta joudutaan muuttamaan. Käytännössä tämä tarkoittaa sitä, että uusia ominaisuuksia voidaan lisätä, koska arkkitehtuuri on toteutettu alusta alkaen abstraktioihin nojaten. Koodiesimerkissä 1 on esitetty toteutus, jossa tätä periaatetta ei ole noudatettu ja koodin ylläpidettävyys on siksi vaikeaa. [14.]

```
public class Cook {

    // Prepare a pizza meal.
    public void prepareMeal(PizzaMeal meal) {
        List<Ingredient> ingredients = meal.getIngredients();
        mixAndStir(ingredients);
    }

    // Prepare a salad meal.
    public void prepareMeal(SaladMeal meal) {
        List<Ingredient> ingredients = meal.getIngredients();
        mixAndStir(ingredients);
    }

    private void mixAndStir(List<Ingredient> ingredients) {
        // prepare a meal
    }
}
```

Koodiesimerkki 1. Cook-luokka ja sen kaksi ateriakohtaista prepareMeal()-metodia.

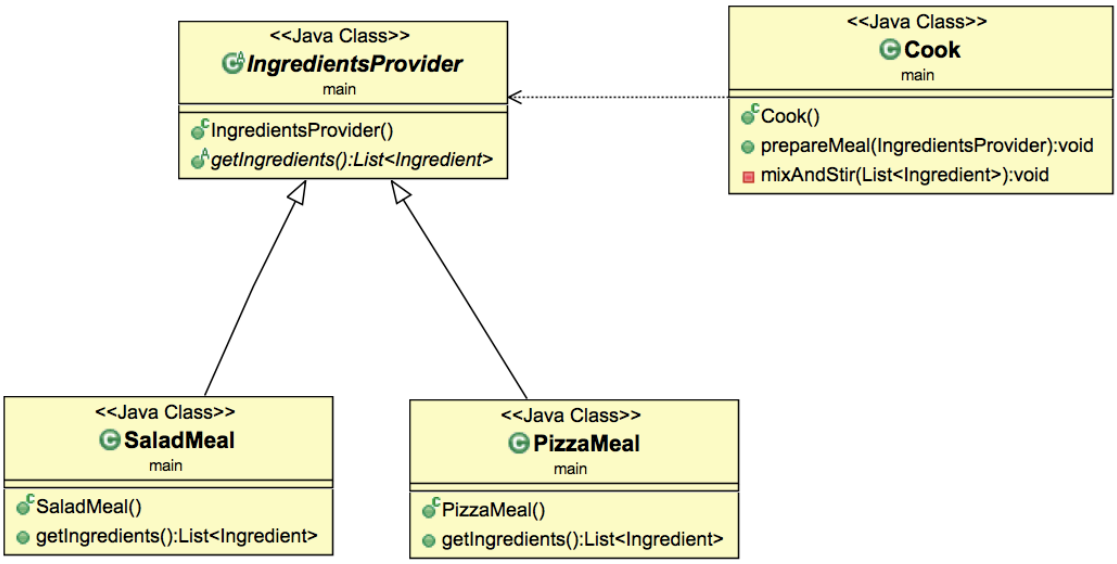
Luokka Cook sisältää kaksi prepareMeal()-metodia, kullekin erilaiselle ateriavaihtoehdolle omansa. Jos ateriavaihtoehtoja tulee tulevaisuudessa lisää, tarkoittaa se, että luokkaan on kirjoitettava uusi metodi kyseiselle aterialle. Tämän lisäksi ateria tarvitsee oman luokan ja toteutuksen getIngeredients()-metodille. Konkreettisten luokkien käyttäminen aiheuttaa näin ollen duplikaattikoodia, eli toistuvaa koodia, ja vaikeuttaa ylläpidettävyyttä. Toteutus ei noudata avoin-suljettu-periaatetta, koska luokan toiminnallisuuden lisääminen vaatii, että olemassa olevaa toteutusta muutetaan.

Koodiesimerkissä 2 on esitetty, miten sovellus voidaan toteuttaa avoin-suljettu-periaatetta noudattaen. SaladMeal ja PizzaMeal toteuttavat abstraktin IngredientsProvider-luokan. Cook-luokassa on enää yksi metodi, joka osaa valmistaa kaikenlaisia aterioita, kunhan ne toteuttavat IngredientsProvider-rajapinnan. Abstraktin luokan IngredientsProvider avulla toteutuksesta saadaan laajennettava ilman, että Cook-luokan sisältöä joudutaan muuttamaan. Uusia aterioita voidaan lisätä, niihin viitata abstraktin luokan määrityksen kautta ja niiden aineksia kysellä toteuttajaluokkien getIngredients()-metodilta.

```
public abstract class IngredientsProvider {
    public abstract List<Ingredient> getIngredients();
}

public class SaladMeal extends IngredientsProvider {

    public List<Ingredient> getIngredients() {
        List<Ingredient> ingredients =
                              new ArrayList<Ingredient>();
        ingredients.add(new Ingredient("lettuce", 5));
        ingredients.add(new Ingredient("tomato", 7));
        return ingredients;
    }
}
public class PizzaMeal extends IngredientsProvider {

    public List<Ingredient> getIngredients() {
        List<Ingredient> ingredients =
                              new ArrayList<Ingredient>();
        ingredients.add(new Ingredient("cheese", 1));
        ingredients.add(new Ingredient("tomato sauce", 1));
        return ingredients;
    }
}
public class Cook {

    public void prepareMeal(IngredientsProvider provider) {
        List<Ingredient> ingredients =
                                provider.getIngredients();
        mixAndStir(ingredients);
    }

    private void mixAndStir(List<Ingredient> ingredients) {
        // prepare a meal
    }
}
```

Koodiesimerkki 2. Avoin-suljettu-periaatteen toteutus.

Kuvassa 1 on esitetty luokkakaavio edellä mainittujen luokista. Luokat SaladMeal ja PizzaMeal toteuttavat IngredientsProvider abstraktin luokan. Cool-luokka käyttää IngredientsProvider-luokan toteuttajaa prepareMeal()-metodissa.



Kuva 1. Luokkakaavio.

Avoin-suljettu-periaatteen noudattaminen parantaa koodin ylläpidettävyyttä, koska toiminnallisuutta voidaan lisätä ilman riskiä olemassa olevan toiminnallisuuden rikkoutumisesta. OCP:n toteuttamiseen voidaan käyttää hyväksi muita tässä luvussa esiteltyjä suunnitteluperiaatteita. Seuraavassa aliluvussa käsitellään näistä ensimmäistä, Yhden vastuualueen periaatetta.

3.2 Yhden vastuualueen periaate (SRP)

Yhden vastuualueen periaatteen mukaan jokaisella luokalla tulee olla vain yksi vastuualue. Ylläpidettävyyden kannalta tämä tarkoittaa sitä, että yhden luokan muutosten ei tule vaikuttaa muihin luokkiin. Tästä johtuu suoraan, että yhden vastuualueen periaate edesauttaa OCP:n toteuttamista. [14.]

Luokan toiminnallisuuden pitää muodostaa koherentti kokonaisuus. Siinä määriteltyjen metodien tulee liittyä toisiinsa niin, että kokonaistoiminnallisuus keskittyy vain yhteen asiaan. Hyvä ja helppo tarkistustapa on miettiä jokaisen metodin kohdalla, sopiiko se

sijaitsemaan sen nimisessä luokassa. Tästä tietysti seuraa se, että luokan nimen tulee kuvata luokan toiminnallisuutta ja olla riittävän yksityiskohtainen. Esimerkiksi luokka OrderHandler ei kuvaa riittävällä tarkkuudella yhtä toiminnallisuutta. Parempi vaihtoehto olisi pilkkoa tällainen luokka pienempiin luokkiin ja antaa niille kutakin alitoiminnallisuutta kuvaavat nimet, kuten LabelAttacher, CostCalculator ja PacketSender. [13.]

Yhden vastuualueen periaatteen noudattaminen parantaa myös testattavuutta. Yksikkötestejä on helpompi kirjoittaa, jos testattava yksikkö on pieni, eikä sillä ole useita riippuvuuksia. Jos metodilla on vain yksi toiminnallisuus, sen testaaminen eristyksissä helpottuu. Seuraavassa luvussa käsitellään abstraktioihin ja periytymiseen keskittyvää suunnitteluperiaatetta Liskovin korvaavuusperiaate.

3.3 Liskovin korvaavuusperiaate (LSP)

Avoin-suljettu-periaatteen toteuttamisen työkaluna käytetään periytymistä, koska periytyminen mahdollistaa abstraktioiden käytön ja abstraktiot taas edelleen mahdollistavat toiminnallisuuden laajentamisen ja korvaamisen ilman, että palvelusta riippuvia luokkia joudutaan muuttamaan. Liskovin korvaavuusperiaate pyrkii edesauttamaan siinä, että periytyminen toteutetaan OCP:tä noudattavalla tavalla.

Liskovin korvaavuusperiaatteen mukaan yliluokan A aliluokka B tulee voida korvata yliluokalla A. Eli toisin sanoen, jos nämä kaksi alkuehtoa pitävät paikkansa: 1) luokka B on luokan A aliluokka ja 2) metodin signatuurissa parametrina on yliluokka A, esimerkiksi DoSomething(A param), metodin toiminnallisuus ei saa rikkoutua, jos sille välitetään luokan B instanssi. Tämä tarkoittaa sitä, että metodissa ei saa olla parametrina annetun luokan tyypin (A tai B) mukaista eriteltyä toiminnallisuutta. Koodiesimerkissä 3 on esitetty LSP:tä rikkova periytyminen.

```
public class Building {
    public enum BuildingType {
        Cottage,
        ApartmentBuilding,
        Hut
    }

    private BuildingType type;

    public Building(BuildingType type) {
        this.type = type;
    }

    public void build(Building building) {
        if (building.type == BuildingType.Hut)
           ((Hut)building).build();
        else if (building.type == BuildingType.Cottage)
           ((Cottage)building).build();
        else if (building.type == BuildingType.ApartmentBuilding)
           ((ApartmentBuilding)building).build();
    }
}


public class Cottage extends Building {
    public Cottage(BuildingType type) {
           super(type);
    }

    public void build() {
           System.out.println("Cottage build");
    }
}
```

Koodiesimerkki 3. Liskovin korvavuusperiaatetta rikkova periytyminen.

Luokan Building build()-metodista nähdään, että periytyminen rikkoo LSP:tä. Jos metodille välitetään yliluokan instanssi, koodin suoritus pysähtyy tyyppimuunnos-

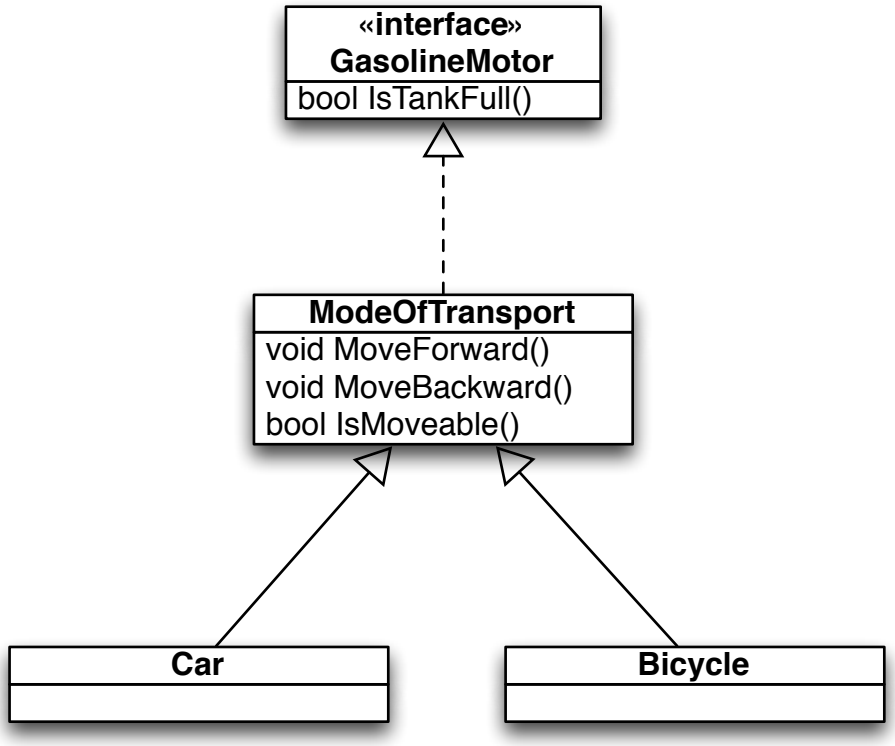
poikkeukseen. Building-luokan instanssia ei voida tyyppimuuttaa aliluokan instansseiksi, eli aliluokan instanssia ei voida korvata yliluokan instanssilla.

Liskovin korvaavuusperiaate johdattelee myös ajattelemaan periytymistä laajemmin kuin kahden eri luokan yhteisten ominaisuuksien kautta. Yleisesti voidaan ajatella, että jos luokka B sisältää luokan A *ominaisuudet*, luokan B tulisi periä luokka A. LSP kiinnittää huomiota luokan *toiminnallisuuteen*. Perintähierarkiaa tulisi miettiä luokan toiminnallisuuden kannalta, eli miten luokkaa käytetään. Esimerkiksi neliö on ominaisuuksiltaan suorakulmio, mutta mikäli suorakulmiota käytetään tavalla, jolla neliötä ei voida käyttää, rikotaan Liskovin korvaavuusperiaatetta. Seuraavassa luvussa käsitellään Rajapintojen erotteluperiaatetta. [13.]

3.4 Rajapintojen erotteluperiaate (ISP)

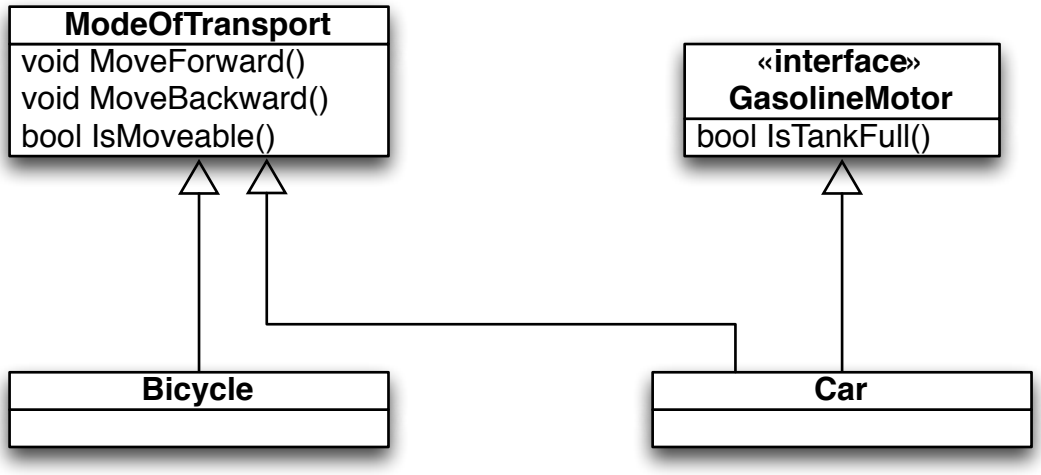
Rajapintojen erotteluperiaatetta voidaan verrata yhden vastuualueen periaatteeseen, sillä se pyrkii erottelemaan rajapinnat koherentteihin kokonaisuuksiin. Tämä tarkoittaa sitä, että yhteen rajapintaan on määritelty vain siihen oleellisesti kuuluva toiminnallisuus. Pilkkomalla rajapinnat useaan osaan, mahdollistetaan se, että toteuttajaluokkien ei tarvitse toteuttaa mitään sellaista, mitä ne eivät toimintaansa tarvitse. Toisinaan muutokset toteuttajaluokissa vaativat muutoksia rajapintoihin. Mikäli rajapinnat ovat eroteltuja, koskevat muutokset vain tämän tietyn rajapinnan toteuttajia.

Kuvassa 2 on esitetty esimerkki, jossa rajapinnan toteuttaja joutuu toteuttamaan itselleen tarpeetonta toiminnallisuutta. Sekä auto että polkupyörä ovat kulkuvälineitä, mutta vain autossa on bensamoottori. Kuvan hierarkian mukaan pyöränkin on toteutettava GasolineMotor-rajapinta, vaikka siinä ei ole sille mitään oleellista toiminnallisuutta. Käytännössä tämä saattaisi tarkoittaa niin sanottua tyhjää metoditoteutusta, joka taas rikkoisi Liskovin korvaavuusperiaatetta, koska Bicycle-luokan käyttäjä ei välttämättä tiedä, ettei IsTankFull()-metodille ole järkevää toteutusta. [13.]



Kuva 2. Car- ja Bicycle-luokat toteuttavat GasolineMotor-rajapinnan.

Kuvan 2 tilanne voidaan muokata noudattamaan ISP:tä erottelemalla GasolineMotor-rajapinta abstraktista luokasta ModeOfTransport. Näin vain Car voi toteuttaa kummankin rajapinnan ja Bicycle vain toisen. Kuvassa 3 on esitettynä vastaava tilanne.



Kuva 3. GasolineMotor-rajapinta on erotettu abstraktista luokasta ModeOfTransport.

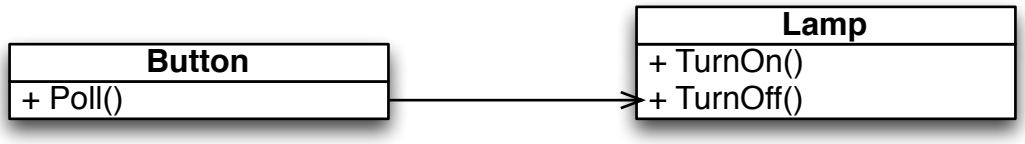
Kuvassa 3 GasolineMotor-rajapinta on erotettu abstraktista luokasta ModeOfTransport. Näin toteuttajaluokat voivat toteuttaa vain itsensä kannalta relevantit osat. Bicycle- ja Car-luokat ovat kummatkin kulkuvälineitä ja toteuttavat siten ModeOfTransport abstraktin luokan. Vain autolla on bensatankki, ja tästä johtuen vain Car-luokka toteuttaa GasolineMotor-rajapinnan.

Huomion arvoista rajapintojen erottelussa on myös se, että pienempiä rajapintoja on helpompi hallita. Samoin esimerkiksi palvelurajapinnat toimivat sopimuksina kuluttajille, eikä niissä selkeyden vuoksi tulisi olla niihin loogisesti kuulumattomia metodeja. Seuraavassa luvussa käsitellään riippuvuuden kääntöperiaatetta.

3.5 Riippuvuuden kääntöperiaate (DIP)

Perinteisessä proseduraalisessa ohjelmoinnissa ylemmän tason toimijat ovat riippuvaisia alemman tason toteutusyksityiskohdista. Tästä johtuen alemman tason muutoksilla on vaikutuksia myös ylöspäin hierarkiassa. Riippuvuuden kääntöperiaate on olio-ohjelmoinnille ominainen tapa vastata näihin ongelmiin abstraktioiden ja riippuvuuksien suunnan muuttamisen kautta. Abstraktioiden käyttö lisää uudelleenkäytettävyyttä ja riippuvuuksien suunnan kääntäminen vähentää koodin herkkyyttä muutokselle.

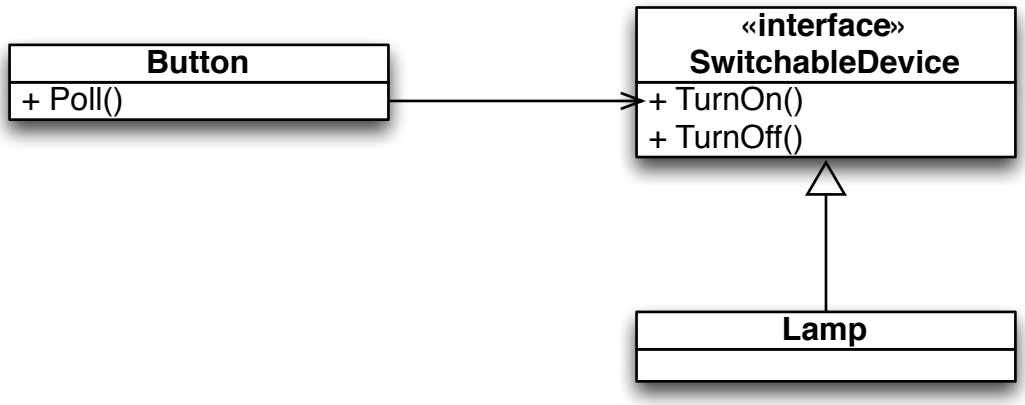
Kuvassa 4 on esitettynä lampun toimintaa kuvaava yksinkertainen arkkitehtuuri. Siinä Button-luokka käyttää Lamp-luokkaa. Lamppu joko sytytetään tai sammutetaan riippuen lampun kulloisestakin tilasta. Painikkeen ja lampun välinen riippuvuus aiheuttaa sen, että mahdolliset muutokset lampun toteutuksessa voivat aiheuttaa muutoksia myös painikkeeseen. Lisäksi uusia päällelaitettavia laitteita ei voida helposti lisätä, koska arkkitehtuurissa käytetään konkreettisia luokkia.



Kuva 4. Button käyttää Lamp-luokkaa ja on täten suoraan riippuvainen siitä.

Kuvassa 5 lampun ja painikkeen välinen riippuvuus on käännetty vastaamaan riippuvuuden kääntöperiaatetta. Button-luokka käyttää SwitchableDevice-rajapintaa, jonka

Lamp edelleen toteuttaa. Tällä mahdollistetaan se, että muutokset konkreettisessa SwitchableDevice-rajapinnan toteuttajassa eivät vaikuta Button-luokan toimintaan ja toisaalta painike voidaan vaihtaa esimerkiksi kytkimeen tai mihin tahansa luokkaan, joka osaa käyttää SwitchableDevice-rajapintaa. Abstraktioiden ja riippuvuuden suunnan käännöksellä mahdollistettiin uudelleenkäytettävyys.



Kuva 5. Button käyttää SwitchableDevice-rajapintaa, jonka Lamp-toteuttaa.

Palvelin-asiakas-arkkitehtuureissa DIP:n toteuttaminen tarkoittaa monesti sitä, että asiakas määrittelee oman pakkauksensa tai C#-kielessä nimiavaruutensa sisällä tarvitsemansa palvelinrajapinnan. Näin palvelimen tulee muuttua silloin, kun asiakkaan tarpeet muuttuvat, ja toisaalta muutokset palvelimen sisäisessä toteutuksessa eivät vaikuta asiakkaan toimintaan. [13.]

Noudattamalla tässä ja edellisissä luvuissa olevia suunnitteluperiaatteita voidaan edistää laajennettavan, modulaarisen ja testattavan koodin syntyä. SOLID:in määrittelemät suunnitteluperiaatteet eivät toki ole ainoita hyvän olio-ohjelmoinnin periaatteita, mutta ne ovat hyvin suosittuja esimerkiksi testilähtöisen (engl. Test Driven Development, TDD) ja ketterän kehityksen parissa. Riippuvuusinjektio ja SOLID:in periaatteet kulkevat käsi kädessä, sillä suunnitteluperiaatteiden noudattaminen johdattaa myös riippuvuusinjektion käyttämiseen. Seuraavassa luvussa keskitytään tarkemmin riippuvuusinjektion tarjoamiin hyötyihin.

4 Riippuvuusinjektion tuomat hyödyt

Tosielämässä kuluttajan ja palvelun suhde ei aina ole pysyvä. Kuluttajan tarpeet muuttuvat ja tarjottavan palvelun on kyettävä vastaamaan niihin. Käytännössä tämä tarkoittaa vastaavan palvelun uudelleenkonfiguroimista tai vaihtamista toiseen palveluun. Ohjelmistoteknisesti eri palveluiden jakama yhteinen, kuluttajan tuntema rajapinta mahdollistaa nämä muutokset ilman, että kuluttajan tarvitsee parhaassa tapauksessa edes olla tietoinen muutoksesta. [8.]

Riippuvuusinjektion suurimmat hyödyt ovat testattavuus, myöhäinen sidonta, laajennettavuus, ylläpidettävyys ja yhtäaikainen kehittäminen. Lisäksi hyötyjä ovat muutoksienhallinnan keskittäminen ja toistuvan koodin vähentäminen. Kuluttajan tarvitsema palvelu, kuten esimerkiksi tekstinkäsittelyohjelman hyödyntämä sanakirja, voidaan vaihtaa kuluttajan tietämättä. Tietyn instanssin luomiseen tarvittavat toimenpiteet, esimerkiksi tietokantapalvelun yhdistämiseen vaadittavat toimet, kirjoitetaan koodin vain kerran. [4; 8; 10.]

Riippuvuusinjektion avulla instanssien tarvitsemat riippuvuudet injektoidaan instansseihin luontihetkellä. Tämä mahdollistaa olemassa olevan todellisen palvelun korvaamisen samaista palvelua mallintavalla matkijalla. Näin palvelua kutsuvia metodeita eli sovelluksen toimintalogiikkaa voidaan testata ilman yhteyttä todelliseen palveluun. Esimerkiksi tekstiviestipalvelua kehitettäessä ei varmastikaan ole toivottavaa lähettää tekstiviestejä jokaisessa testiajossa. [4.]

Yksinkertaisuudessaan riippuvuusinjektion periaate tarkoittaa riippuvuuksien antamista niiden käyttäjälle sen sijaan, että käyttäjä joutuisi niitä luomaan tai kyselemään. Sen toteuttaminen ei vaadi kolmannen osapuolen kirjastoja, mutta niistä voi olla hyötyä. Riippuvuusinjektion vaiheet koostuvat kolmesta perustoiminnallisuudesta: komponenttien välisen riippuvuuden määrittelystä, tämän riippuvuuden toteutuksen konfiguroinnista ja luotujen instanssien elinkaaren hallitsemisesta. Käytettäessä avuksi esimerkiksi kolmannen osapuolen riippuvuusinjektiosäiliöitä komponentit tulee määritellä, ja tämä määrittely tulee kertoa riippuvuusinjektion toteuttajalle, joka tämän jälkeen huolehtii tarvittavien olioiden elinkaaresta. Seuraavissa aliluvuissa käydään tarkemmin läpi riippuvuusinjektion hyötyjä. [8; 10.]

4.1 Testattavuus

Testattavuus tarkoittaa, että sovellusta voidaan yksikkötestata. Yksikkötestaus taas vaatii, että ohjelmayksikköjä voidaan testata eristyksissä. Liittämällä näitä ohjelmayksikköjä löyhästi toisiinsa, mahdollistetaan testattavuuden säilyminen. Toisin sanoen noudattamalla Liskovin korvaavuusperiaatetta, saadaan aikaan arkkitehtuuri, jossa testitapauksissa voidaan käyttää matkijoita tai sijaisia (engl. Test Double) konkreettisten toteuttajien sijaan. [10.]

Liskovin korvaavuusperiaatteen noudattaminen taas ohjaa käyttämään riippuvuusinjektiota. Tämä johtuu siitä, että jotta yliluokan tai rajapinnan viittaukset voidaan korvata eri toteuttajilla, pitää ne välittää jollakin riippuvuusinjektion toteutustavalla, kuten metodiparametrina.

4.2 Myöhäinen sidonta

Myöhäinen sidonta tarkoittaa sitä, että tietyn rajapinnan toteuttaja voidaan korvata toisella, konfiguraatiossa määrätyllä toteuttajalla. Käytännössä tämä tarkoittaa esimerkiksi tilannetta, jossa samalle palvelurajapinnalle on useampi eri toteuttaja ja kulloinkin käytössä olevaa toteuttajaa halutaan vaihdella. Riippuvuusinjektion tapauksessa tämä tarkoittaa kolmannen osapuolen riippuvuusinjektiosäiliön käyttämistä. Tällöin säiliön konfiguraatioon määritellään, mikä konkreettinen rajapinnan toteuttaja välitetään kussakin tapauksessa. Tästä seuraa luonnollisesti sama vaatimus kuin testattavuuden kohdalla: riippuvuudet tulee välittää niiden käyttäjille, jotta päätös niiden konkreettisista toteuttajista voidaan tehdä muualla. Monesti myöhäistä sidontaa pidetään riippuvuusinjektion tärkeimpänä hyötynä. [10.]

4.3 Laajennettavuus

Valtaosalla kehitettävistä sovelluksista on tarpeita laajentua. Uusia ominaisuuksia halutaan lisätä tai kehittää vaihtoehtoisia toiminnallisuuksia, kuten esimerkiksi web- ja mobiilikäyttöliittymät. Laajennettavuus sovelluskehityksessä puhtaimmillaan tarkoittaa sitä, että toiminnallisuutta voidaan lisätä ilman, että olemassa olevaan koodiin pitää tehdä

muutoksia. Riippuvuusinjektio mahdollistaa löyhän liitännän eri komponenttien välillä, ja tämä taas edesauttaa laajennettavuutta.

Esimerkkinä voidaan ajatella viestinkirjoitussovellusta, jonka yksinkertainen toiminnallisuus on kirjoittaa näytölle viestejä. Jos tämän sovelluksen toteutus on tehty laajennettavuuden näkökulma mielessä pitäen, voidaan sovellukseen lisätä esimerkiksi käyttäjän autentikoiminen käyttäen dekoraattori-suunnittelumallia. Dekoraattori-suunnittelumalli on monesti hyödyllinen avoin-suljettu-periaatetta noudattaessa, jolloin luokat pysyvät avoinna laajennuksille, mutta suljettuina muutoksille. Luonnollisesti tämä tarkoittaa abstraktioiden käyttöä. [10.]

4.4 Ylläpidettävyys

Kuten laajennettavuus, myös ylläpidettävyys on vaatimuksena useimmissa kehitettävistä sovelluksista. Niin sanottu spagettikoodi tekee koodipohjasta vaikeasti ylläpidettävän. Tällainen koodi on rakenteeltaan sotkuista ja eri moduulit ovat tiukasti sidoksissa toisiinsa [16]. Yhden vastuualueen periaate pyrkii estämään spagettikoodin syntyä toteamalla, että jokaisella luokalla pitäisi olla vain yksi vastuualue. Tällaisen periaatteen noudattaminen johtaa siihen, että ylläpidettävyys paranee. Toiminnallisuutta on helpompi muuttaa, jos sen sijainti on rajattu ja selkeästi kerrottu. On myös mahdollista, että muutoksia voidaan tehdä, ilman, että olemassa olevaa koodia joudutaan muuttamaan, jos aikaisempi toteutus voidaan korvata uudella toteutuksella vain rajapinnan toteuttajaa vaihtamalla. [10.]

4.5 Yhtäaikainen kehitys

Yhtäaikainen kehitys on tilanne, jossa useampi kehittäjä tai tiimi kehittäjiä työskentelee saman sovelluksen parissa yhtäaikaisesti. Jotta tällainen tilanne sujuisi mahdollisimman hyvin, pitää sovelluksen eri osa-alueiden olla eroteltu toisistaan irrallisiin moduuleihin. Tämä tarkoittaa sitä, että niiden pitää olla löyhästi sidoksissa toisiinsa. Näin toisen tiimin tai kehittäjän ei tarvitse huolehtia kuin omasta osa-alueestaan. [10.]

Yllämainitut riippuvuusinjektion tarjoamat hyödyt ovat kiistatta arvokkaita. Näiden lisäksi muutoksien hallinnan parantuminen ja toistuvan koodin vähentyminen ovat itsessään

jo syitä, joiden vuoksi riippuvuusinjektion käyttöä kannattaa suosia. Seuraavassa luvussa on esitetty tapoja, jolla riippuvuusinjektion voi toteuttaa.

5 Tapoja toteuttaa riippuvuusinjektio

Riippuvuusinjektio lisää yhden abstraktiotason sovelluksen arkkitehtuuriin niin sanotun injektorin kautta. Injektori on kuluttajan ja palvelun välillä toimiva taho, joka huolehtii palvelusta ja välittää sen tarvittaessa kuluttajalle. Toisin kuin Service Locator [9] -suunnittelumallissa, jossa kuluttaja pyytää tarvitsemaansa palvelua palveluntarjoajalta, riippuvuusinjektion periaatteen mukaan kuluttajan ei tarvitse pyytää haluamaansa palvelua eksplisiittisesti vaan se välitetään sille injektorin toimesta [3].

Toteuttaessa riippuvuusinjektio käsin, palvelun ja kuluttajan välillä toimiva injektori on oletettavasti luokka, joka luo sekä palvelun että kuluttajan. Näin luodut palvelut voidaan välittää eteenpäin kuluttajalle. Koodiesimerkissä 4 on esitetty yksinkertainen palvelu-kuluttaja-järjestelmä, jossa Program-luokka toimii injektorina.

```
public interface IService
{
    void Serve();
}

public class Service : IService
{
    public void Serve()
    {
        // serve the client
    }
}
```

```
public class Consumer()
{
    private IService service;

    public Consumer(IService service)
    {
        this.service = service;
    }

    public void Consume()
    {
        service.Serve();
    }
}

public class Program()
{
    public void Run()
    {
        IService service = new Service();
        Consumer consumer = new Consumer(service);
        consumer.Consume();
    }
}
```

Koodiesimerkki 4. Program-luokka toimii injektorina.

Koodiesimerkissä 4 kuluttaja Consumer on riippuvainen palvelusta IService. Riippuvuus annetaan Consumer-luokka alustajan parametrina. Program-luokka luo instanssit IService-rajapinnan toteuttajasta Service ja kuluttajasta Consumer. Samoin se injektoi Consumer-luokan alustajaan referenssin IService-rajapinnan toteuttajaan.

Palvelun injektointi voidaan jaotella kolmeen päätapaan. Nämä ovat alustaja-, setteri- ja rajapintainjektio, ja ne ovat verrattavissa laajemman käsitteen Inversion of Control vastaaviin tyyppeihin (IoC1, IoC2 ja IoC3) [3]. Tässä työssä käsitellään näistä alustaja- ja setteri-injektiota. Setteri-injektio tunnetaan C#-maailmassa ominaisuusinjektiona (engl. property-injection) [10]. Näiden lisäksi esitellään myös metodi-injektio sekä ympäröivän kontekstin käsite. Jokaiselle injektiotavalle on olemassa kolmannen osapuo-

len ohjelmistokehyksiä ja myöhemmissä luvuissa esiteltävistä ratkaisuista PicoContainer sekä Google Guice käyttävät alustajainjektiota. Castle Windsor ja Unity tukevat alusta-, setteri- ja metodi-injektiota [10].

Nimensä mukaisesti alustajainjektiossa kuluttaja saa tarvitsemansa riippuvuudet alustajassa. Toisin sanoen palveluiden on oltava olemassa ennen kuluttajaa, jotta injektori voi injektoida ne kuluttajaan sen luonnin yhteydessä. Setteri-injektiossa taas kuluttaja voidaan luoda ilman palveluita ja tarvittavat palvelut injektoida myöhemmin settereiden kautta. Rajapintainjektio vaatii erityisten injektointi- ja injektorirajapintojen kirjoittamisen. Injektointirajapinta määrittää tarvittavat riippuvuudet, jotka injektorirajapinnan toteuttaja jälleen injektoi. [3.]

Alkeellisimmillaan riippuvuusinjektio voidaan toteuttaa käsin alustajien tai settereiden kautta. Vastaava lähtökohta on riittävä tilanteisiin, jossa kuluttajaan liitettävällä palvelulla ei ole omia riippuvuuksia. Koodiesimerkissä 5 on esitetty tällainen tilanne. Emailer-luokan tarvitsema SpellChecker-instanssi välitetään alustajassa.

```
public Emailer(SpellChecker checker) {
            this.checker = checker;
}
```

Koodiesimerkki 5. Yksinkertainen injektointi.

Todellisuudessa kuluttajan riippuvuuksilla eli palveluilla voi kuitenkin olla omia riippuvuuksiaan ja tällaisissa tilanteissa riippuvuusinjektion toteuttamiseen kannattaa harkita jotakin kolmannen osapuolen ohjelmistokehystä. Käyttämällä kolmannen osapuolen riippuvuusinjektiosäiliötä eri riippuvuuksien toteuttajat voidaan helposti määrittää säiliön konfiguroinnissa. Näin muutoksia voidaan hallita paremmin, koska rajapinnan toteuttaja voidaan valita vain yhdessä kohtaa koodia. Koodiesimerkissä 6 on esitettynä sähköpostipalvelun luonti, joka havainnollistaa tilannetta, jossa käsin tehtävä injektio ei ole järkevä ratkaisu.

```
public Emailer(SpellChecker checker) {
            this.checker = checker;
            checker.setDictionary(new FinnishDictionary());
            checker.setHyphenation(new Hyphenation());
```

Koodiesimerkki 6. Injetoitavalla palvelulla on omia riippuvuuksia [8].

Koodiesimerkin 6 Emailer-luokka on riippuvainen SpellChecker-luokasta. SpellChecker-luokka taas on edelleen riippuvainen FinnishDictionary- ja Hyphenation-luokista. Jos sovelluksessa luodaan useampia oikolukupalveluita, on jokaiselle niistä tehtävä koodissa esitetty konfigurointi. Tämän konfiguroinnin hallintaan kolmannen osapuolen riippuvuusinjektiosäiliöt tarjoavat apua.

Rajapintainjektio hyödyntää rajapintoja injektion toteuttamisessa. Käytännössä rajapintainjektio muistuttaa setteri-injektiota, sillä riippuvuuden tarvitsevan luokan pitää toteuttaa rajapinta, jonka kautta injektio voidaan toteuttaa. Tällöin riippuvuuden tarjoaja, eli injektori, voi käyttää tätä rajapintaa hyödykseen riippuvuuden täyttämisessä [3]. Seuraavissa luvuissa käydään tarkemmin läpi riippuvuusinjektion eri toteutusperiaatteita, kuten alustaja-, setteri- ja metodi-injektio.

5.1 Alustajainjektio

Alustajainjektiossa tarvittavat riippuvuudet annetaan luokan alustajassa. Tästä seuraa luonnollisesti olettamus siitä, että riippuvuus on saatavissa aina luokan koodia suorittaessa. Alustajainjektion ehdoton etu muihin riippuvuusinjektion toteutustapoihin onkin juuri tämä, riippuvuuden olemassaolon takaaminen. Toki tämä tarkoittaa sitä, että alustajainjektion toteutuksessa on otettava huomioon tilanne, jossa annettu riippuvuus on null-arvoinen. Koodiesimerkissä 7 on esitelty alustajainjektio ja null-arvoon varautuminen.

```
public class Consumer
{
    private readonly IService service;

    public Consumer(IService service)
    {
        if (service == null)
            throw new ArgumentNullException("service");

        this.service = service;
    }
}
```

Koodiesimerkki 7. IService-rajapinnan toteuttaja injektoidaan Consumer-luokkaan.

Alustajainjektion null-arvoon varautuminen on koodiesimerkissä 7 tehty poikkeusten kautta. Mikäli parametrina injektoitu IService-rajapinnan toteuttaja on null-arvoinen, heitetään koodista poikkeus. Näin sovelluksesta aiheutuu ajonaikainen virhe, joka huomataan oletettavammin jo käynnistyksen yhteydessä.

Alustajainjektion toteuttaminen vaatii myös, että luokalla on parametrillinen alustaja. Saattaa kuitenkin olla, että on tilanteita, jossa luokalle on myös määriteltävä parametriton oletusalustaja. Tästä esimerkkinä on Windows Forms-tekniikalla toteutetut käyttöliittymäkomponentit, joiden näyttäminen Visual Studion suunnittelunäkymässä kutsuu aina parametritonta alustajaa. Tällaisissa tilanteissa alustajainjektion toteuttaminen ei ole niin suoraviivaista.

Toisin kuin setteri-injektiossa, riippuvuudella ei ole välttämätöntä olla luokan sisäistä oletusarvoa. Huomion arvoista on kuitenkin, että riippuvuuksien välittäminen alustajassa saattaa aiheuttaa kuormitusta sovelluksen alustamiseen, koska kaikki riippuvuudet pitää luoda heti. Useimmissa tapauksissa alustajainjektio on kuitenkin järkevin vaihtoehto ja myös helpoin toteuttaa. Seuraavassa luvussa esitellään setteri-injektio. [10.]

5.2 Setteri-injektio

Setteri-injektioita voidaan kutsua myös ominaisuusinjektioksi (engl. property-injection), erityisesti .NET-maailmassa. Sillä tarkoitetaan riippuvuuden asettamista joko setterin

tai ominaisuuden (engl. property) välityksellä. Alustajainjektiosta poiketen setteri-injektio ei oletuksena takaa, että riippuvuus on annettu. Käytännössä tämä tarkoittaa sitä, että setteri-injektio sopii tilanteisiin, jossa luokan riippuvuudelle on olemassa oletusarvo, joka voidaan tarvittaessa korvata setteri-injektion välityksellä.

Setteri-injektion toteutus vaatii, että riippuvuus voidaan asettaa joko julkisen ominaisuuden tai setterimetodin kautta. Riippuvuuden olemassaolo voidaan varmistaa oletusarvon asettamisella esimerkiksi luokan alustajassa. Koodiesimerkissä 8 on esitetty setteri-injektion toteutus.

```
public class Consumer
{
    private IService service;

    public IService Service
    {
        get
        {
            if (this.service == null)
            {
                Service = new DefaultService();
            }
            return service;
        }
        set
        {
            if (value == null)
                throw new ArgumentNullException("value");

            if (service != null)
                throw new InvalidOperationException();

            service = value;
        }
    }
}
```

Koodiesimerkki 8. IService-rajapinnan toteuttaja injektoidaan Consumer-luokkaan ominaisuuden kautta.

Koodiesimerkissä 8 IService-rajapinnan toteuttaja injektoidaan Consumer-luokkaan ominaisuuden kautta. Jos ominaisuutta ei ole asetettu ennen sen käyttöä, käytetään oletustoteutusta. IService-rajapinnan toteuttajan asettaminen useammin kuin kerran on estetty poikkeuksella, samoin kuin null-arvon asettaminen.

Setteri-injektion haittapuolia ovat aiemmin mainittu mahdollinen riippuvuuden puuttuminen ja toisaalta se, ettei sen asettaminen tapahdu vain yhdellä ennalta määrätyllä hetkellä, kuten alustajan suorituksen aikana. Tämä tarkoittaa sitä, että on mahdollista, että riippuvuus vaihtuu tahattomasti kesken suorituksen. Tähän voidaan tosin varautua estämällä riippuvuuden asettaminen yhtä kertaa useammin.

Yksi setteri-injektion sovellus on avoin-suljettu-periaatteen toteuttaminen. Tällöin luokan sisäinen toteutus voidaan pitää muuttumattomana ja samalla mahdollistaa sen toiminnan laajentaminen. Luokan sisäinen riippuvuuden oletusarvo mahdollistaa sen, että koodin suorittaminen on mahdollista joka tilanteessa ja toisaalta riippuvuuden asettaminen myöhemmin mahdollistaa sen, että toiminnallisuutta voidaan lisätä. Seuraavassa luvussa esitellään metodi-injektio. [10.]

5.3 Metodi-injektio

Metodi-injektiossa riippuvuus välitetään metodin parametrina. Näin mahdollistetaan se, että jokainen suorituskerta voi olla erilainen ja toisaalta, että suoritusta voidaan ohjata annetun riippuvuuden mukaan. Voidaankin sanoa, että metodi-injektiossa riippuvuus on se konteksti, jonka mukaan metodissa toimitaan. Koodiesimerkissä 9 on esitetty metodi-injektio.

```
public string DoSomething(Dependency value, Context context)
{
    if (value == null)
        throw new ArgumentNullException("value");

    return value.DoSomethingThatReturnsAString();
}
```

Koodiesimerkki 9. Riippuvuus annetaan metodiparametrina.

DoSomething()-metodi ottaa parametrikseen riippuvuudet Dependency ja Context. Null-arvoon varautuminen hoidetaan heittämällä poikkeus. Näin voidaan varmistua, että riippuvuus on annettu.

Riippuvuuden välittäminen metodiparametrina ei välttämättä tarkoita, että välitettyä riippuvuutta käytettäisiin jokaisella suorituskerralla. Esimerkkinä tästä on rajapinta ja sen kaksi erilaista toteuttajaa. Rajapinnassa määritellyn metodin parametreina välitetään arvo ja sitä käsittelevä konteksti. Kaksi eri toteutusta mahdollistaa sen, että toisessa konteksti voidaan jättää huomioimatta ja toisessa sitä voidaan taas hyödyntää. Koodiesimerkissä 9 välitetään metodiin DoSomething() myös Context-olio, mutta sitä ei esimerkin toteutuksessa käytetä. Jos metodi DoSomething() on määritelty rajapinnassa, voivat sen muut toteuttajat tarvittaessa käyttää Context-olion sisältöä. Seuraavassa luvussa esitellään ympäröivän kontekstin käsite. [10.]

5.4 Ympäröivä konteksti

Ympäröivä konteksti tarkoittaa tietyn rajatun alueen sisällä saatavissa olevaa julkista tietoa. Rajattu alue voi olla esimerkiksi Javassa tiettyyn pakkaukseen kuuluvat luokat tai mahdollisesti koko sovellus. Koko sovelluksesta puhuttaessa tämä julkinen tieto on oletettavasti saatavilla staattisen kentän tai globaalin muuttujan kautta.

Koska globaalien muuttujien käyttöä tulisi välttää [17], ympäröivää kontekstia tulisi käyttää myös harkiten ja harvoin. On kuitenkin tilanteita, joissa sen käyttö on tarpeellista. Tällaisia tilanteita ovat tilanteet, joilla on monialaisia vaikutuksia (engl. cross-cutting concerns), joissa koko sovellusaluetta lävistää tietty konteksti. Staattinen kenttä mahdollistaa näin sen, ettei riippuvuutta pidä erikseen välittää jokaiselle luokalle, vaan se on aina saatavilla.

Julkisen staattisen kentän käyttö aiheuttaa myös setteri-injektiossa havaitun ongelman. Mikäli kentän asettamista ei ole estetty, voidaan riippuvuus milloin tahansa korvata toisella. Näin ei voida olla varmoja siitä, että riippuvuuden tila on yhtenäinen koko suorituksen ajan. [10.]

Riippuvuusinjektion eri toteutustavat ovat keinoja välittää riippuvuuksia niitä tarvitseville osapuolille ilman, että riippuvuuksia pitää kysyä. Riippuvuuksien välittämisen lisäksi riippuvuusinjektion periaatteisiin kuuluu välitettyjen riippuvuuksien elinkaaresta ja näkyvyysalueesta huolehtiminen. Tämä tarkoittaa sitä, kuinka kauan ja kuinka laajalla

alueella injektoitavat oliot ovat olemassa. Nämä ovat aspekteja, joiden vuoksi kolmannen osapuolen riippuvuusinjektiosäiliöiden käyttöä kannattaa harkita, sillä ne tarjoavat selkeitä ja helppokäyttöisiä keinoja elinkaaren ja näkyvyysalueen hallintaan. Seuraavassa luvussa käsitellään tarkemmin elinkaarta ja näkyvyysaluetta.

6 Olion olemassaolon dimensiot

Olion olemassaolo rakentuu kahdesta eri dimensiosta: elinkaari (engl. lifecycle) ja näkyvyysalue (engl. scope). Elinkaari määrittää kaikki ne vaiheet, joita olio käy olemassaolonsa aikana läpi. Esimerkiksi musiikkisoittimen vaiheita voisivat olla *toisto*, *tauko* ja *pysäytys*. Olion näkyvyysalue taas määrittää sen, mikä instanssi oliosta kulloinkin on käytössä. Esimerkiksi tietokantayhteydestä vastaavasta luokasta on usein tarpeellista olla olemassa vain yksi instanssi. Näin varmistetaan arkkitehtuurin kannalta osiltaan se, että kantayhteydet ovat synkronisia. Seuraavissa luvuissa tarkastellaan tarkemmin olion elinkaarta ja näkyvyysaluetta.

6.1 Elinkaari

Olion elinkaari koostuu kaikista niistä vaiheista, joita se olemassaolonsa aikana käy läpi. Elinkaaren voidaan ajatella olevan kontekstiriippuvainen ja näin ollen kahden eri olion läpikäymät vaiheet eivät välttämättä ole samat. Jokaiselle oliolle on kuitenkin yhteistä kaksi vaihetta: konstruktio ja dekonstruktio. Olion elinkaari alkaa, kun sen alustajaa kutsutaan, ja päättyy, kun olion muistivaraus vapautetaan.

Luokkainstanssien luonti saattaa olla raskas operaatio suorituskyvyn tai muistinkulutuksen kannalta. Olioiden elinkaarten suunnittelu on siksi tärkeä osa ohjelmiston arkkitehtuurin suunnittelua. Esimerkiksi voidaan ottaa sovellus, jonka avulla voidaan katsella valokuvia. Tällainen toiminto vaatii lukuvirtojen avaamista ja tiedoston käsittelijöiden luomista. Jokainen tiedostonkäsittelijä varaa muistia, ja siksi on tärkeää, että ne myös suljetaan tehokkaasti.

Niin sanottu laiska initialisointi saattaa taas olla tarpeellinen tilanteessa, jossa olion luonti on raskasta ja luonti halutaan näin suorittaa vasta kun olion instanssia tarvitaan ensimmäisen kerran. Toisaalta on myös tilanteita, jossa luokkainstansseja halutaan

luoda innokkaasti (engl. eager), eli siten, että instanssit luodaan, vaikka niitä ei heti käytettäisikään.

Olion olemassaoloon vahvasti liittyvä toinen käsite elinkaaren rinnalla on näkyvyysalue. Kolmannen osapuolen riippuvuusinjektio-ratkaisut tarjoavat mahdollisuuden konfiguroida olioiden elinkaaria ja näkyvyysaluetta. Seuraavassa luvussa käsitellään näkyvyysaluetta tarkemmin. [8.]

6.2 Näkyvyysalue

Olion olemassaoloon liittyy elinkaaren lisäksi myös näkyvyysalue. Näkyvyysalue määrittää sen ajan, minkä sisällä tietty avain viittaa tiettyyn instanssiin. Esimerkiksi istuntokohtainen näkyvyysalue (engl. session) voi määrittää, että tietyn http-istunnon aikana käyttäjän kirjautumisoikeudet ovat muuttumattomat, eli niitä kysyttäessä saadaan aina sama instanssi.

Yleisemmin käytetty näkyvyysalue on *no scope* (Spring-kehyksessä tämä on nimeltään *prototype*), joka tarkoittaa sitä, että jokainen riippuvuus täytetään aina uudella, toisista riippumattomalla instanssilla. Toinen usein käytetty näkyvyysalue on ainokainen (engl. singleton). Ainokainen-näkyvyysalue tarkoittaa sitä, että tiettyä avainta (Spring-kehyksessä bean-id) vastaan palautetaan aina sama instanssi, jos toimitaan saman riippuvuusinjektiosäiliön sisällä.

Suunnittelumalleista puhuttaessa ainokainen-suunnittelumalli perustuu sille, että tietystä oliosta on sovelluksen olemassaoloaikana olemassa vain yksi instanssi. Tämä on perusteltua tilanteissa, joissa monella eri taholla on riippuvuus yhteen ja samaan palveluun, esimerkiksi tietokantayhteyteen ja kutsut tietokantaan halutaan pitää synkronisina. Näin yhteyden ottaminen tietokantaan kulkee hallitusti vain yhtä reittiä. Ainokainen-suunnittelumallissa on kuitenkin myös haittapuolia, kuten sen heikko testattavuus. Koodiesimerkissä 10 on kuvattu riippuvuus ainokainen-instanssiin, joka pakottaa käyttämään tiettyä instanssia DbManager-oliosta. Testattavuuden kannalta tämä tarkoittaa sitä, että testitapauksissa on käytettävä todellista tietokantayhteyttä. Toinen mahdollisuus olisi muuttaa DbManager-luokkaa niin, että se sisältäisi setterin, jossa ainokainen-instanssi voitaisiin korvata. Tämä kuitenkin rikkoisi ainokainen-suunnittelumallin periaatteita.

```
public class UserManagement
{
    private readonly Connection connection;

    public UserManagement()
    {
        connection = DbManager.GetInstance().GetConnection();
    }

    public void AddNewUser(string userName)
    {
        User user = new User(userName);
        connection.AddUser(user);
    }
}
```

Koodiesimerkki 10. DbManager-luokan GetInstance()-metodi palauttaa ainokainen-instanssin DbManager-luokasta. Tältä instanssilta kysytään edelleen kantayhteys GetConnection()-metodin kautta.

DbManager-luokan GetInstance()-metodi palauttaa ainokainen-instanssin DbManager-luokasta. Tältä instanssilta kysytään edelleen kantayhteys GetConnection()-metodin kautta. Tämä tarkoittaa sitä, että testitapauksissa DbManager-instanssia ei voida korvata esimerkiksi matkijalla (eng. mock). Sen sijaan joudutaan aina käyttämään oikeaa instanssia ja näin ollen myös kantayhteyttä. Matkija toteuttaa saman rajapinnan oikean instanssin kanssa, mutta sen metodien toiminnallisuus on eri tai mahdollisesti tyhjä.

Näkyvyysalueiden tapauksessa ainokaisella on eri merkitys. Esimerkiksi Spring-kehyksessä ainokainen-bean tarkoittaa sitä, että tiettyä bean-id:tä vastaan palautetaan aina sama olioinstanssi tietyn injektorin sisällä. Eli siinä, missä suunnittelumallien ainokainen palauttaa koko sovelluksen aikana aina saman instanssin, riippuvuusinjektiokehyksistä puhuttaessa palautetaan aina riippuvuusinjektiosäiliökohtainen instanssi. Spring-kehyksessä siis bean-id on näkyvyysalueeltaan ainokainen. Testattavuuden kannalta tämä tarkoittaa sitä, että testitilanteessa voidaan samaan luokkaan viitata eri bean-id:llä, jolloin ainokainen-instanssi voidaan korvata testin ajaksi.

Riippuvuusinjektiokehysten käyttämisestä saadaan näkyvyysalueiden kannalta se hyöty, että määrittämällä tietty avain tiettyyn näkyvyysalueeseen, kehys vastaa näkyvyysalueen noudattamisesta koko sovelluksen käytön ajan. Näkyvyysalue ja elinkaari liittyvät vahvasti kolmannen osapuolen riippuvuusinjektiosäiliöihin. Seuraavassa luvussa esitellään, mihin tarkoitukseen tällaisia säiliöitä voidaan käyttää sekä käydään läpi muutama tarjolla oleva toteutusvaihtoehto. [8.]

7 Kolmannen osapuolen riippuvuusinjektiosäiliöt

Kolmannen osapuolen riippuvuusinjektiosäiliöllä tarkoitetaan sovelluskirjastoa, joka on kehitetty riippuvuusinjektion tarpeisiin. Riippuvuusinjektion toteuttamisen kannalta säiliöiden käyttäminen ei ole pakollista, mutta niistä voi tarpeen tullen hyötyä suuresti. Riippuvuusinjektion toteuttamiseen suurissa sovelluksissa saattaa liittyä paljon niin sanottua kytköskoodia, jonka avulla sovelluksen eri osat kytketään toisiinsa. Tästä esimerkkinä on halutun rajapinnan toteuttajan valinta. Tämä on sellainen toiminnallisuus, joka voidaan jättää riippuvuusinjektiosäiliön vastuulle. Tärkeää on kuitenkin ymmärtää, että jotta tällaisesta säiliöstä on apua, pitää sovelluksen arkkitehtuurin noudattaa riippuvuusinjektion periaatteita. Toisin sanoen, säiliö itsessään ei takaa riippuvuusinjektion toteutumista.

Yhteistä kaikille riippuvuusinjektiosäiliöille on, että ne tarjoavat ohjelmointirajapinnan (API) ja tarvitsevat konfiguraatiota toimiakseen. Ohjelmointirajapinta tarjoaa metodeja esimerkiksi tietyn rajapinnan toteuttajan luomiseen. Konfiguraatiossa taas määritellään, millä rajapinnan toteuttajalla kukin viittaus täytetään. Säiliöt tarjoavat myös keinoja luvussa kuusi mainittujen olioiden ominaisuuksien elinkaari ja näkyvyysalue hallintaan. Säiliöiden avulla voidaan siis hallita sovelluksen oliograafin luomista ja sen ajonaikaisia muutoksia. [10.]

Seuraavissa luvuissa on esitelty muutamia laajemmin tunnettuja kolmannen osapuolen riippuvuusinjektiosäiliöitä. Näistä C#-kielellä on käytössä Castle Windsor ja Unity. Javalle taas on olemassa Google Guice ja PicoContainer. Näiden lisäksi Spring on olemassa sekä Java- että C#-kielille, mutta sen esittely on sen monipuolisuuden vuoksi jätetty työn ulkopuolelle. Lisätietoja Spring-kehyksestä on saatavilla osoitteessa [18].

7.1 Castle Windsor

Castle Windor on yksi kehittyneimmistä kolmannen osapuolen riippuvuusinjektiosäiliöistä. Se on osa avoimen lähdekoodin projektia Castle Project ja saatavilla sekä .NET- että SilverLight-ympäristöihin [19]. Castle Windsor on monipuolinen IoC-säiliö, jonka käyttö yksinkertaisimmillaan tarkoittaa säiliön konfiguroimista eli haluttujen rajapintojen ja luokkien rekisteröimistä sekä niiden käyttämistä säiliöstä käsin. Käyttäminen tässä tarkoittaa sitä, että Castle Windsor täyttää sovelluksessa olevat riippuvuudet, mikäli se on konfiguroitu niin tekemään. [10.]

Esimerkki Castle Windsorin yksinkertaisesta konfiguroinnista on koodiesimerkissä 11. Itse säiliö luodaan new-avainsanan avulla. Tämän jälkeen säiliöön tulee rekisteröidä tarvittavat rajapinnat ja niiden toteuttajat. Tarvittaessa voidaan rekisteröidä myös konkreettisia luokkia, samoin kuin olemassa olevia instansseja. Rekisteröinti suoritetaan Register()-metodin kautta. Rekisteröinnin jälkeen säiliöltä voidaan pyytää tietyn rajapinnan tai luokan toteuttajaa Resolve()-metodin kautta. Jokainen Castle Windsorin tarvitsema tyyppi pitää rekisteröidä säiliöön [10]. Tämä tarkoittaa sitä, että jos esimerkiksi rekisteröidään rajapinta IVehicle ja sille toteuttaja Car ja Car-luokan alustaja ottaa parametrikseen IMotor-rajapinnan toteuttajan, myös IMotor ja sen toteuttaja pitää rekisteröidä. Tärkeä havainto Castle Windsorin säiliön konfiguroinnista on se, että koodiesimerkissä 11 oleva komento rekisteröi vain komponentin rajapinnalle IVehicle. Toisin sanoen, säiliöstä ei tämän jälkeen voi selvittää Car-luokkaa kutsumalla container.Resolve<Car>(). Car-luokan instanssi saadaan vain IVechicle-rajapinnan kautta.

```
var container = new WindsorContainer();
container.Register(Component
                    .For<IVehicle>()
                    .ImplementedBy<Car>());
IVehicle car = container.Resolve<IVehicle>();
```

Koodiesimerkki 11. Castle Windsor-riippuvuusinjektiosäiliön yksinkertainen konfiguraatio.

Huomion arvoista on, että vaikka säiliöltä voidaan missä tahansa sen näkyvyysalueella pyytää luokkainstansseja, on sen käyttö tällä tavalla riippuvuusinjektion periaatteiden vastaista. Sen sijaan Resolve()-metodia tulisi käyttää vain kokoonpanojuuren (engl. composition root) yhteydessä. Castle Windsoria tulisi sen sijaan käyttää täyttämään

riippuvuuksia niin, että sen toiminta on muulle sovellukselle huomaamatonta. Tämä tarkoittaa sitä, että kun sovellus pyöräytetään käyntiin, Castle Windsor varmistaa, että tarvittavat luokat on instantioitu ja että nämä instanssit ovat olemassa ennalta määrätyn ajan. [19.]

Castle Windsor säiliön konfigurointi on mahdollista tehdä joko XML:n, koodin tai autorekisteröinnin kautta. Esimerkki koodin avulla tehtävästä rekisteröinnistä esiteltiin koodiesimerkissä 11. Etuna siinä on vahva tyypitys ja eksplisiittisyys. Autorekisteröinti taas auttaa tilanteissa, jossa halutaan esimerkiksi rekisteröidä useita saman rajapinnan toteuttajia. Koodiesimerkissä 12 on esitetty esimerkki autorekisteröinnistä. Siinä kaikki saman käännöspakkauksen (engl. assembly) sisällä saman rajapinnan toteuttavat luokat rekisteröidään kerralla. Uusien samat ehdot toteuttavien luokkien lisääminen ei näin ollen aiheuta tarvetta rekisteröintikonfiguraation muuttamiseen. Autorekisteröinnin ohessa voidaan käyttää myös hyväksi nimeämiskonventioita. Rekisteröintikutsuun voidaan lisätä ehto, jolla tarkistetaan, että tietty sana tai kirjainyhdistelmä on osa luokan nimeä. [10.]

```
var container = new WindsorContainer();
container.Register(AllTypes
            .FromAssemblyContaining<Car>()
            .BasedOn<IVehicle>());
```

Koodiesimerkki 12. Castle Windsorin autorekisteröinti.

Vanhin Castle Windsorin konfiguroimistapa on XML. XML:n käytössä on se etu, että konfigurointia voidaan muuttaa ilman, että koodimuutoksia joudutaan tekemään. Koodiesimerkissä 13 on esitetty esimerkki XML-konfiguraatiosta. Komponentit rekisteröidään <castle>-elementin sisällä. Komponentille määritetään tunnistekenttä, toteutettava rajapinta ja sen toteuttaja. Tällainen konfiguraatio voidaan ottaa käyttöön kutsumalla säiliön Install()-metodia ja välittämällä sille parametriksi haluttu konfiguraatiotiedosto. [10.]

```
<castle>
  <components>
    <component id=”vehicles.car”
               service=”IVehicle”
               type=”Car”/>
  </components>
</castle>
```

Koodiesimerkki 13. Castle Windsorin konfiguroiminen XML:n avulla.

Kolmannen osapuolen riippuvuusinjektiosäiliöiden avulla voidaan hallita olioiden elinkaarta ja näkyvyysalueita. Castle Windsorissa elinkaari ja näkyvyysalue määritetään elintyylin (engl. lifestyle) avulla. Käytössä ovat elintyylit, kuten ainokainen (engl. Singleton), ohimenevä (engl. Transient), http-pyyntökohtainen (engl. PerWebRequest) sekä itse määriteltävän muokattavan (engl. Custom) elinkaarimallin. Ainokainen-elinkaarimalli on oletuselinkaarimalli Castle Windsorissa. Elintyylin määrittäminen on osa säiliön konfiguroimista ja se voidaan määrittää joko yhdelle tai useammalle komponentille kerralla. Koodiesimerkissä 14 on esitetty elintyylin määrittäminen käyttämällä koodissa tapahtuvaa konfigurointia. Olioinstanssit voidaan tarvittaessa vapauttaa kutsumalla säiliön Release()-metodia. Tämän jälkeen roskienkeruu huolehtii olion siivoamisesta. [10.]

```
container.Register(
           Classes.FromThisAssembly()
           .BasedOn<IController>()
           .LifestyleTransient());
```

Koodiesimerkki 14. Elintyylin määrittäminen [19].

Koodiesimerkissä 14 jokainen saman käännöspakkauksen luokka, joka toteuttaa IContoller-rajapinnan, rekisteröidään ohimenevällä elintyylillä. Tämä tarkoittaa sitä, ettei Castle Windor huolehdi näiden olioinstanssien olemassaolon pysyvyydestä. Toisin sanoen, jokainen kerta, kun instanssia pyydetään, säiliöstä saadaan uusi instanssi ja vanhat tuhoutuvat, kun niihin ei enää ole viittausta.

Riippuvuusinjektion kannalta Castle Windsor tukee alustaja- ja ominaisuusinjektiota. Samoin tehdasluokkien käyttäminen on mahdollista. Tehdasluokkia voidaan käyttää

hyödyntämällä säiliön konfiguroinnissa UsingFactoryMethod()-funktiota. Ominaisuusinjektio taas on mahdollista rekisteröimällä halutun ominaisuuden tyyppi säiliöön. Tämän jälkeen Castle Windsor osaa palauttaa tälle ominaisuudelle arvon. Seuraavassa luvussa käsitellään toista kehittynyttä riippuvuusinjektiosäiliötä Unity.

7.2 Unity

Unity on Microsofting kehittämä riippuvuusinjektiosäiliö .NET- ja Silverlight ympäristöihin. Se tukee alustaja, property- ja metodi-injektiota [22]. Unity on Castle Windsoriin verraten nuorempi tekijä, ja sen käyttö on yhtä lailla maksutonta. Castle Windsorista poiketen Unityn säiliöstä on mahdollista pyytää instanssia ilman, että sitä on ensin rekisteröity. Tämä tosin vaatii sitä, että kyseisellä luokalla on olemassa parametriton oletusalustaja. Instanssien selvittäminen säiliöstä on mahdollista myös parametrillisten alustajien tilanteessa, mikäli sisimmällä riippuvuudella on parametriton oletusalustaja. [10.]

Koodiesimerkissä 15 on esitetty Unityn konfigurointia. Samoin kuin Castle Windsorissa UnityContainer voidaan luoda new-avainsanalla. Komponenttien rekisteröinti tehdään RegisterType()-metodin kautta. Tämä metodi ottaa tyypeikseen rekisteröitävän rajapinnan ja sen toteuttajan. Resolve()-metodin avulla voidaan pyytää instanssia IVehicle-rajapinnan toteuttajasta. Kuten aikaisemmassa kappaleessa mainittiin, Unity tukee konkreettisten luokkien selvittämistä ilman rekisteröintiä. Säiliöstä voidaan näin kutsua Car-olion selvittämistä ilman, että sitä on ensin rekisteröity.

```
var container = new UnityContainer();
container.RegisterType<IVehicle, Car>();
IVehicle car = container.Resolve<IVehicle>();
```

Koodiesimerkki 15. Unity-riippuvuusinjektiosäiliön konfigurointi.

Koodiesimerkissä 15 Unity-riippuvuusinjektiosäiliö konfiguroidaan koodissa. Säiliöön rekisteröidään IVehicle-rajapinta ja sille toteuttaja Car. Resolve()-metodia kutsumalla voidaan pyytää instanssia IVehicle-rajapinnan toteuttajasta.

Unity tukee sekä XML- että koodikonfigurointia. Koodiesimerkissä 15 on esitetty koodin avulla tehtävä konfigurointi. Huomionarvoista Unityn konfiguroinnissa on se, että kutsumalla RegisterType()-metodia rekisteröidään sekä rajapinta että sen toteuttaja, toisin kuin Castle Windsorissa. Tämä tarkoittaa sitä, että rekisteröinnin jälkeen säiliöstä voidaan selvittää kumpikin komponentti käyttäen Resolve()-metodia. Castle Windsorista poiketen, Unity ei tuo autorekisteröintiä. Koodiesimerkissä 16 on esitetty Unityn konfiguroiminen XML:n avulla. [10.]

```
<unity>
  <namespace name="Vehicles"/>
  <assembly name="Vehicles"/>
  <container>
    <register type="IVehicle" mapTo="Car"/>
  </container>
</unity>
```

Koodiesimerkki 16. Unity-riippuvuusinjektiosäiliön konfigurointi XML:n kautta [10].

XML-konfiguroinnissa komponentit määritellään <unity>-elementin sisällä. Rekisteröintiä varten asetetaan nimiavaruus sekä käännöspakkauksen nimi. Rajapintojen toteuttajien määrittely tehdään <container>-elementin sisällä käyttäen <register>-elementtiä.

Myös Unity tukee useita elintyylimalleja, kuten ohimenevä (engl. Transient) ja säiliön hallitseman (engl. Container Controlled). Ohimenevä-elintyyli on Unity-riippuvuusinjektiosäiliön oletuselintyyli. Se tarkoittaa sitä, että säiliö ei hallinnoi instanssien elinkaarta. Koodiesimerkissä 17 on esitetty ainokainen-elintyylin määrittäminen Unity säiliöön. Ainokainen-elintyyli voidaan määrittää antamalla RegisterType()-metodille parametrina ContainerControlledLifeTimeManager-instanssi.

```
container.RegisterType<Car>(
            new ContainerControlledLifeTimeManager());
```

Koodiesimerkki 17. Ainokainen-elintyylin määrittäminen Unity-säiliöön.

Sekä Unity että Castle Windsor ovat kummatkin loistavia vaihtoehtoja riippuvuusinjektiosäiliötä valittaessa. Tässä työssä esitettyjen ominaisuuksien lisäksi kummatkin tar-

joavat myös muita kehittyneitä toiminnallisuuksia, kuten esimerkiksi niin sanotun välistävetäjän (engl. interceptor), jonka avulla voidaan toteuttaa esimerkiksi lokalisaatiota tai virheenkäsittelyä. Seuraavassa luvussa esitellään lyhyesti Java-kielelle yleisesti käytettyjä riippuvuusinjektiosäiliöitä.

7.3 Java: Google Guice ja PicoContainer

Google Guice on Googlen kehittämä Java-kielinen ohjelmistokehys, jonka tarkoituksena on tarjota annotaatioita käyttävä monipuolinen ratkaisu riippuvuuksien poistamiseen. Guice toimii alustajainjektointiperiaatteen mukaan [1]. Se pyrkii vähentämään annotaatioiden avulla tehdasluokkien ja new-avainsanan käytön tarvetta. Keskeisin annotaatio on `@inject`, jonka avulla Guice päättelee, mitä riippuvuuksia injektoidaan. [4.]

Yksinkertaisimmillaan Guice vaatii sidosmoduulin kirjoittamisen sekä `@inject` –annotaation lisääminen kuluttajan alustajaan. Tämän jälkeen Guice liittää sidosmoduulissa määritellyn palvelun kuluttajaan, kun kuluttaja luodaan Guicen injektoriedustajan avulla. [1.]

Guicen tapaan myös PicoContainer toimii alustajainjektion periaatteen mukaan, mutta se mahdollistaa myös setteri-injektion käytön. PicoContainer on saatavissa Javan lisäksi myös .NET, Ruby ja PHP-kielille [5]. PicoContainerin rakennetta voidaan yksinkertaistaen kuvata kokoelmarakenteen, kuten Javan Hashtable, avulla. Jokainen riippuvuussuhteen osapuoli eli sekä palvelu että kuluttaja lisätään tähän kokoelmarakenteeseen. Lisäysvaiheessa määritellään haluttu riippuvuus eli toteuttajaluokka. Kuluttaja voidaan luoda kutsumalla kokoelmarakenteesta haluttua luokkaa [6].

Riippuvuusinjektion toteuttaminen ei vaadi kolmannen osapuolen riippuvuusinjektiosäiliön käyttöä ja toisaalta säiliön käyttöönottaminen ei takaa riippuvuusinjektion toteutumista. Seuraavassa luvussa on esitelty niin sanottu ”Poor Man’s DI” eli käsin tehty riippuvuusinjektion toteutus käyttäen tehdasluokkia.

7.4 Vaihtoehto: tehdasluokat

Tehdasluokka on kolmannen osapuolen sovelluksista riippumaton tapa toteuttaa riippuvuusinjektio. Se noudattaa riippuvuusinjektion pääperiaatetta, eli riippuvuuksien siirtämistä sovelluksen ytimestä sen reunoille, jossa niitä on helpompi hallita muutostilanteissa [1]. Tehdasluokka toimii nimensä mukaisesti tehtaana, jonka tehtävänä on luoda instansseja tietystä luokasta. Tehtaan käyttäjän ei tarvitse tietää tehtaan toiminnasta muuta kuin tuotettavan olion toteuttava rajapinta. Tämä mahdollistaa saman rajapinnan eri toteuttajien luomisen dynaamisesti. Tässä luvussa käydään läpi esimerkki, jossa tehdasluokat ovat apuna riippuvuusinjektion toteutuksessa. [7.]

Koodiesimerkissä 18 on esitetty tehdasluokkia käyttävä toteutus. Tässä toteutuksessa Program-luokka toimii injektorina. IConnection-rajapinnan toteuttajasta luodaan instanssi kutsumalla ConnectionFactory-tehdasluokan CreateConnection()-metodia. Samoin IDeliveryMedium-rajapinnan toteuttajasta luodaan instanssi kutsumalla DeliveryMediumFactory-tehdasluokan CreateDeliveryMedium()-metodia. Nämä instanssit välitetään edelleen Sender-luokan alustajalle sen luonnin yhteydessä. Sender-luokan Send()-metodia voidaan tämän jälkeen käyttää viestin lähettämiseen.

```
public class Program
{
    static void Main(string[] args)
    {
        IConnection connection =
                    ConnectionFactory.CreateConnection(true);
        IDeliveryMedium deliveryMedium =
               DeliveryMediumFactory.CreateDeliveryMedium(false);
        Sender sender = new Sender(connection, deliveryMedium);
        sender.Send("Send this social media message");
    }
}
```

Koodiesimerkki 18. ConnectionFactory ja DeliveryMediumFactory toimivat tehdasluokkina.

Haluttu IConnection- ja IDeliveryMedium-rajapinnan toteuttaja voidaan määrittää tehdasluokassa. Tehdasluokkien CreateConnection()- ja CreateDeliveryMedium()-metodit ottavat parametreikseen boolean-arvon, jonka perusteella luodaan haluttu instanssi.

Koodiesimerkissä 19 on esitetty tehdasluokkien sisältö. ConnectionFactory-luokan CreateConnection()-metodi ottaa sisäänsä boolean-arvon, jonka perusteella luodaan joko MobileConnection tai WiFiConnection. DeliveryMedium-luokan CreateDeliveryMedium()-metodi ottaa sisäänsä parametrin, jonka perusteella luodaan joko MailMedium tai SoMeMedium. Näin voidaan määritellä koodiesimerkissä 20 esitellyn Sender-luokan Send()-metodin käyttämä viestin lähetystapa. Viesti voidaan lähettää joko mobiili- tai wifi-verkossa käyttäen sähköpostia tai jotain sosiaalisen median kanavaa.

```
internal class ConnectionFactory
{
    public static IConnection CreateConnection(bool mobile)
    {
        if (mobile)
            return new MobileConnection();
        else
            return new WiFiConnection();
    }
}


internal class DeliveryMediumFactory
{
    public static IDeliveryMedium CreateDeliveryMedium(bool mail)
    {
        if (mail)
            return new MailMedium();
        else
            return new SoMeMedium();
    }
}
```

Koodiesimerkki 19. Tehdasluokat ConnectionFactory ja DeliveryMediumFactory.

Koodiesimerkin 20 Sender-luokka on riippuvainen IConnection- ja IDeliveryMedium-rajapintojen toteuttajista. Nämä riippuvuudet annetaan alustajainjektion kautta. Sender-luokka käyttää näitä riippuvuuksia Send()-metodissa lähettääkseen parametrina saadun viestin eteenpäin. Sender on tietämätön siitä, minne (sähköpostiin vai sosiaalisen median kanavalle) ja miten (mobiili- tai wifi-verkossa) viesti lähetetään.

```
internal class Sender
{
    private readonly IConnection connection;
    private readonly IDeliveryMedium deliveryMedium;

    public Sender(IConnection connection,
                          IDeliveryMedium deliveryMedium)
    {
        this.connection = connection;
        this.deliveryMedium = deliveryMedium;
    }

    public bool Send(string message)
    {
        return deliveryMedium.Deliver(connection, message);
    }
}
```

Koodiesimerkki 20. Sender-luokka on riippuvainen IConnection- ja IDeliveryMedium-rajapintojen toteuttajista.

Palveluiden tehdasluokat ConnectionFactory ja DeliveryMediumFactory käärivät sisäänsä haluttujen palveluiden luonnin. Tämä mahdollistaa sen, että vastuu palveluista on siirtynyt tehdasluokkiin. Esitetystä ratkaisusta käy ilmi tehdasluokkien heikkous riippuvuusinjektion toteuttajana. Kirjoittavien luokkien määrä kasvaa ja vaikka abstraktiotaso nouseekin mahdollistaen samalla joustavuuden eri palveluiden instantioinnissa, syntyy samalla kokonaisuuksia, jotka tulee kääntää kerralla. Todellisessa sovelluksessa tämä saattaa johtaa huomattavaan määrään kerralla käännettäviä luokkia. [1.]

Seuraavassa luvussa käydään läpi laajempi esimerkki riippuvuusinjektion toteutuksesta. Tässä esimerkissä testattavuutta pyritään parantamaan käyttäen hyväksi riippuvuusinjektion periaatteita.

8 Esimerkkisovellus: toimituksen tilausjärjestelmä

Tässä luvussa esitellään esimerkkisovellus, joka havainnollistaa testattavuuden vaatimuksia riippuvuusinjektion kannalta. Esimerkkisovellus mallintaa yksinkertaista toimi-

tuksen tilausjärjestelmää. Sovelluksessa käyttäjä voi valita lähetettävän tuotteen ja sen kohdemaan ja painaa tämän jälkeen toimituksen vahvistuspainiketta. Painikkeen painamisen jälkeen sovellus näyttää prosessipalkin ja kertoo, onnistuiko toimituksen tilaus. Kuva käyttöliittymästä on tämän työn liitteenä. Seuraavassa luvussa käydään tarkemmin läpi sovelluksessa olevaa riippuvuutta ja sen aiheuttamaa ongelmaa testattavuuteen.

8.1 Riippuvuus testattavuuden esteenä

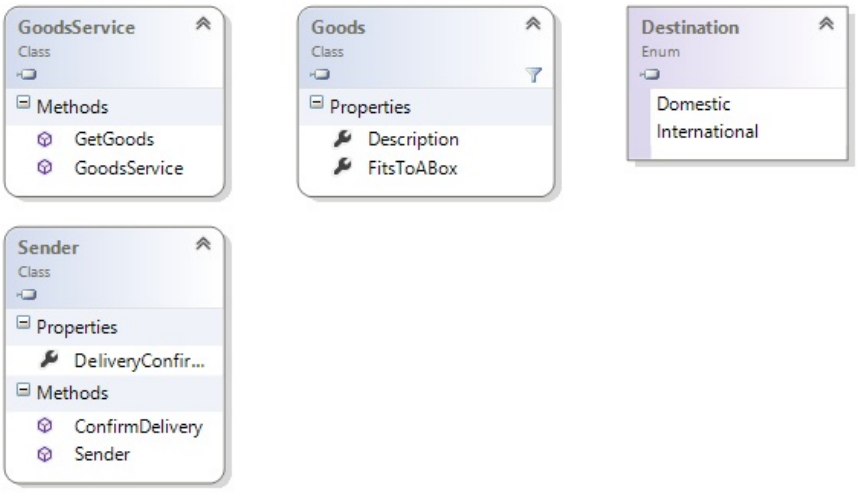
Riippuvuusinjektion kannalta sovelluksen ongelma liittyy lähetettävien tuotteiden pakkaamiseen. Kun käyttäjä painaa Confirm-painiketta, Sender-luokalle välittyy lopulta kutsu vahvistaa toimitus. Tämä tilauksen vahvistus sisältää sekä kutsun toimituksen tilaukseen joltakin GoodsDeliverer-rajapinnan toteuttajalta että kutsun lähetettyjen tavaroiden pakkaamiseen. Näitä Packer-rajapinnan toteuttajia, CartonPacker ja ContainerPacker, käytetään sovelluksessa mallintamaan riippuvuutta ulkoiseen palveluun.

Sovelluksen käyttäjälle näytetään ilmoitus tuotteiden onnistuneesta pakkauksesta. Tämä ilmoitusikkuna kuvastaa sellaista riippuvuutta ulkoiseen palveluun, joka vaikeuttaa testattavuutta. Sovelluksen näyttämälle tilauksen vahvistusviestille on olemassa kaksi yksikkötestiä. Niiden ajaminen itsenäisesti on käytännössä mahdotonta, koska jokainen ajokerta vaatii interaktiota käyttöliittymän kanssa. Pakkauksen ilmoitusikkuna piirtyy ruudulle testejä ajaessa ja yksittäisen testin suoritus jää odottamaan ilmoitusikkunan kuittausta. Todellisessa käyttötilanteessa ilmoitusikkuna voisi olla esimerkiksi pakkauspalvelun tilaus kolmannen osapuolen toimittajalta, jolloin jäätäisiin odottamaan tietoa pakkauspalvelun tilauksen onnistumisesta.

Jotta tilauksen vahvistusviestin sisältöä voidaan testata eristyksissä, eli ilman näytölle tulevaa pakkauksen ilmoitusviestiä, pitää pakkauspalvelun toiminnallisuus korvata mieluiten sellaisella tyhjällä toteutuksella, joka ei piirrä graafista käyttöliittymää. Käytännössä tämä tarkoittaa matkijoiden käyttöä. Jotta Packer-rajapinnan toteuttaja voidaan testissä korvata tyhjällä toteutuksella, pitää se välittää Sender-luokalle sen sijaan, että se luotaisiin Sender-luokan sisällä. Seuraavassa luvussa esitellään sovelluksen luokkakaavio.

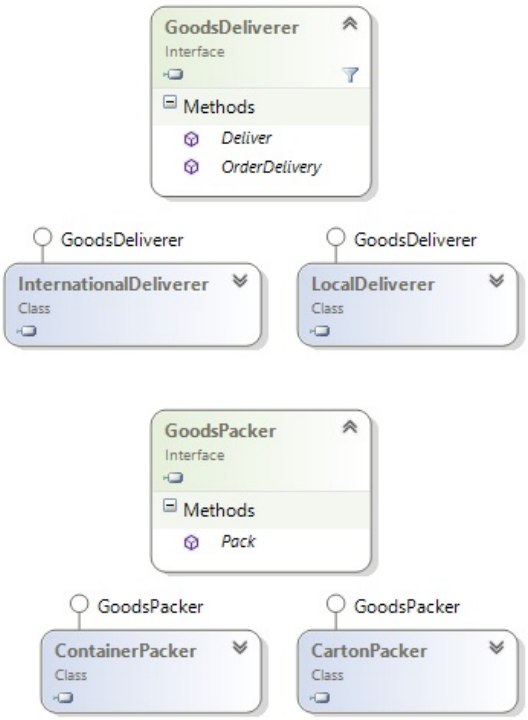
8.2 Esimerkkisovelluksen toimintalogiikkaprojektin luokat

Toimituksen tilausjärjestelmä koostuu kolmesta eri projektista: toimintalogiikka-, käyttöliittymä- ja testiprojektista. Käyttöliittymäprojekti DeliveryApplicationUI käyttää hyväkseen toimintalogiikkaprojektin DeliveryApplication-luokkia. Toimintalogiikkaprojekti DeliveryApplication sisältää kuvien 6 ja 7 mukaiset luokat.



Kuva 6. DeliveryApplication-projektin apuluokat.

Esimerkkisovelluksessa luokka Goods mallintaa lähetettävää asiaa, esimerkiksi ruokaa, metallia tai maaperää. Goods-luokalla on tieto siitä, mahtuuko se pahvilaatikkoon. Tämän tiedon perusteella Sender-luokka valitsee pakkausmateriaalin. GoodsService-luokka kuvastaa sovelluksessa tietokantaa. Destination-enum luettelee kaksi vaihtoehtoista lähetyskohdetta, jotka ovat kotimaa ja ulkomaat. Näistä riippuu se, toimiiko lähetyksen kuljettajana kotimainen vain kansainvälinen yritys. Sender-luokka sisältää metodin ConfirmDelivery(), joka sisältää toiminnallisuuden lähetyksen tilaamisesta ja pakkaamisesta. Kuvassa 7 on kuvattu pakkaukseen ja kuljetukseen liittyvät luokat ja niiden periytymishierarkia.



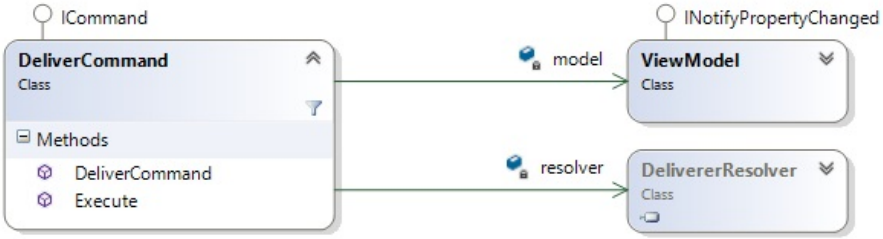
Kuva 7. Toimituksen tilausjärjestelmän pakkaukseen ja toimitukseen liittyvät luokat.

GoodsDeliverer-rajapinta määrittelee toimittajan sekä sille metodit Deliver() ja OrderDelivery(). Esimerkkisovellukseen on toteutettu vain OrderDelivery()-metodi. Tämä on metodi, jota Sender kutsuu ConfirmDelivery()-metodin sisällä tilatakseen kuljetuksen. GoodsPacker-rajapinta määrittää pakkaajan ja sille metodin Pack(). Myös tätä metodia kutsutaan Senderin ConfirmDelivery()-metodista. ContainerPacker- ja CartonPacker-luokat toteuttavat GoodsPacker-rajapinnan. Sender valitsee pakkaajaksi CartonPacker-toteuttajan, mikäli pakattava tuote (Goods-luokan edustaja) mahtuu pahvilaatikkoon ja muussa tapauksessa ContainerPacker-luokan edustajan. Seuraavassa kappaleessa esitellään käyttöliittymäprojektin luokat.

8.3 Esimerkkisovelluksen käyttöliittymäprojektin luokat

Esimerkkisovelluksen käyttöliittymä on toteutettu Windows Presentation Fountation-tekniikkaa [23] käyttäen ja sen arkkitehtuuri noudattaa Model-View-ViewModel-mallia. Käytännössä tämä tarkoittaa sitä, että graafinen käyttöliittymä koostuu käyttöliittymäkomponenttimäärityksistä, painikkeet, tekstikentät ja niin edelleen, jotka on määritelty MainWindow.xaml-tiedostossa. Käyttöliittymän interaktiologiikka taas on sijoitettu ViewModel-luokkaan. ViewModel-luokka on nimensä mukaisesti mallinnus näkymästä.

Sitä kautta koodissa voidaan käsitellä käyttäjän syötteitä. Kolmas osa MVVM-mallia eli Model koostuu esimerkkisovelluksessa monesta eri luokasta. Tähän voidaan ajatella kuuluvan DeliverCommand-luokan, joka sisältää Confirm-painikkeen takaisen logiikan sekä kaikkien toimintalogiikkaprojektin luokkien, koska ne muodostavat sovelluksen toiminnallisuuden. Kuvassa 8 on esitetty DeliveryApplicationUI-projektin toiminnallisuuden kannalta tärkeät luokat.



Kuva 8. DeliveryApplicationUI-projektin toiminnallisuuden kannalta tärkeimmät luokat.

Käyttöliittymäprojektin tärkeimmät luokat ovat DeliverCommand ja ViewModel. ViewModel sisältää julkiset ominaisuudet, joiden avulla käyttöliittymäkomponentteja voi käsitellä koodin kautta. DelivererResolver-luokka on esitelty osana käyttöliittymäprojektia, vaikka se todellisuudessa on toimintalogiikkaprojektin luokka. Tämä johtuu siitä, että vain käyttöliittymäprojekti käyttää sitä. DelivererResolver on tehdasluokka, jonka avulla luodaan oikeanlaisia GoodsDeliverer-rajapinnan toteuttajia riippuen lähetyksen kohdemaasta.

DeliverCommand sisältää toiminnallisuuden, joka liittyy Confirm-painikkeen painamiseen. Luokka toteuttaa ICommand-rajapinnan, jonka avulla DeliverCommand-luokan toteuttajan Execute()-metodia voidaan kutsua Confirm-painikkeen painalluksen yhteydessä. Execute()-metodin suorittaminen käynnistää koko tilausprosessin. Sen sisällä luodaan uusi Sender-luokan toteuttaja. Tämän jälkeen kutsutaan Sender-luokan ConfirmDelivery()-metodia. Vielä lopuksi asetetaan käyttöliittymään näkymään tieto onnistuneesta tilauksesta. Tämä tehdään asettamalla Sender-luokan DeliveryConfirmation-ominaisuuden arvo ViewModel-luokan ConfirmationMessage-ominaisuuden arvoksi. Näin käyttöliittymään päivittyy tieto tilauksen onnistumisesta.

8.4 Esimerkkisovelluksen testiprojekti

Eri luokkien väliset riippuvuudet hankaloittavat monesti testausta. Yksikkötestausta on vaikea tehdä, jos luokkaa ei voida testata eristyksissä. Oikeammin riippuvuudet tekevät yksikkötestauksesta helposti integraatiotestausta, jossa luokkien yhteistyötä testataan. Integraatiotestaus on tärkeä testauksen osa-alue ja sitä kannattaa harjoittaa. Yksikkötesteihin verraten integraatiotestit ovat kuitenkin monesti monimutkaisempia ja alttiimpia hajoamaan ohjelman kehittyessä. Tämä johtuu juuri siitä, että integraatiotesteissä testataan montaa eri luokkaa yhdessä, ja näin ollen muutos yhdessä luokassa saattaa vaikuttaa myös muihin luokkiin.

Esimerkkisovelluksen testiprojekti DeliveryApplicationTests sisältää yhden testiluokan DeliverCommandTest, jossa on määritelty kaksi testiä testaamaan tilauksen vahvistusviestin sisältöä. Koodiesimerkissä 21 on esitetty toinen näistä testeistä. Tässä testissä ViewModel-luokan SelectedDestination-ominaisuuteen asetetaan arvoksi ”International”. Tämä vastaa sitä, että käyttäjä olisi valinnut käyttöliittymästä lähetyksen kohteeksi ulkomaat. DeliverCommand-luokan Execute()-metodin suorituksen yhteydessä ViewModel-luokan ConfirmationMessage-propertyyn asetetaan arvoksi ”UPS delivery confirmed”, koska tilauksen toimitukseen on valittu kansainvälinen toimija.

```
[TestMethod]
public void ConfirmationMessageShowsInternationalDeliverer()
{
    ViewModel model = new ViewModel();
    model.SelectedDestination = "International";
    DeliverCommand command = new DeliverCommand(model);
    command.Execute(null);
    Assert.AreEqual("UPS delivery confirmed.",
                                model.ConfirmationMessage);
}
```

Koodiesimerkki 21. Testi, joka testaa tilauksen vahvistusviestin sisältöä.

Esimerkkisovelluksen testejä voidaan pitää enemmän integraatiotesteinä, koska niissä testataan ViewModel-luokan ominaisuuden arvoja DeliverCommand-luokan Execute()-metodin kautta. Testejä ajaessa huomataan, että Sender-luokalla on testattavuuden

kannalta haitallinen riippuvuus Packer-rajapinnan toteuttajaan. Tämä riippuvuus aiheuttaa pakkauksen onnistumisesta kertovan ilmoitusikkunan piirtymisen näytölle. Seuraavassa luvussa esitellään, miten riippuvuusinjektion avulla mahdollistetaan se, että tämä riippuvuus voidaan korvata tyhjällä toteutuksella.

Huomion arvoista on, että todellisessa sovelluksessa kaksi testiä olisi riittämätön määrä varmistamaan sovelluksen toimivuus. Samoin Test Driven Developmentin harrastaminen ohjaa kirjoittamaan koodia, jossa riippuvuusinjektio on vahvasti läsnä. Testattavuuden käsittely laajemmin, samoin kuin ohjelmoinnin niin sanotut parhaat käytännöt ovat kuitenkin tämän työn aihepiirin ulkopuolelle, joten ne jätetään lukijan mielenkiinnon varaan. Aihetta käsitellään muun muassa Lasse Koskelan teoksessa Test Driven: Practical TDD and Acceptance TDD for Java Developers [21].

8.5 Ratkaisu käyttäen riippuvuusinjektiota

Esimerkkisovelluksessa Sender-luokan yksityinen metodi PackGoods() luo uuden instanssin Packer-rajapinnan toteuttajasta riippuen siitä, mahtuuko pakattava tavara pahvilaatikkoon. Julkinen metodi ConfirmDelivery() kutsuu edelleen tätä metodia. Tästä aiheutuu se, että kummassakin testitapauksessa pakkauksen ilmoitusviesti piirtyy näytölle. Jotta tältä voidaan välttyä, pitää Packer-rajapinnan toteuttaja olla korvattavissa. Näin testitapauksissa voidaan korvata konkreettinen CartonPacker tai ContainerPacker matkijalla, jolla on tyhjä toteutus Pack-metodiin. Koodiesimerkissä 22 on esitetty osittainen Sender-luokan toteutus ennen riippuvuuksien muuttamista.

```
public class Sender
{
    private GoodsDeliverer deliverer;
    private bool orderConfirmed;
    private GoodsPacker packer;

    public Sender(GoodsDeliverer deliverer
    {
       this.deliverer = deliverer;
    }

    public void ConfirmDelivery(Goods goods)
    {
        if (deliverer.OrderDelivery(goods))
        {
            orderConfirmed = true;
            PackGoods(goods);
        }
        else
            orderConfirmed = false;
    }

    private void PackGoods(Goods goods)
    {
        if (goods.FitsToABox)
            packer = new CartonPacker();
        else
            packer = new ContainerPacker();

        packer.Pack(goods);
    }
}
```

Koodiesimerkki 22. Sender-luokan toteutus.

Goods-luokassa on kaikki tarvittava tieto oikean GoodsPacker-rajapinnan toteuttajan valitsemiseen, sillä toteuttajan valinta tehdään FitsToABox-ominaisuuden perusteella. On myös huomionarvoista, että FitsToABox-ominaisuutta ei käytetä sovelluksessa muuhun tarkoitukseen. Tämä johdattaa miettimään, onko tarpeellista, että ominaisuu-

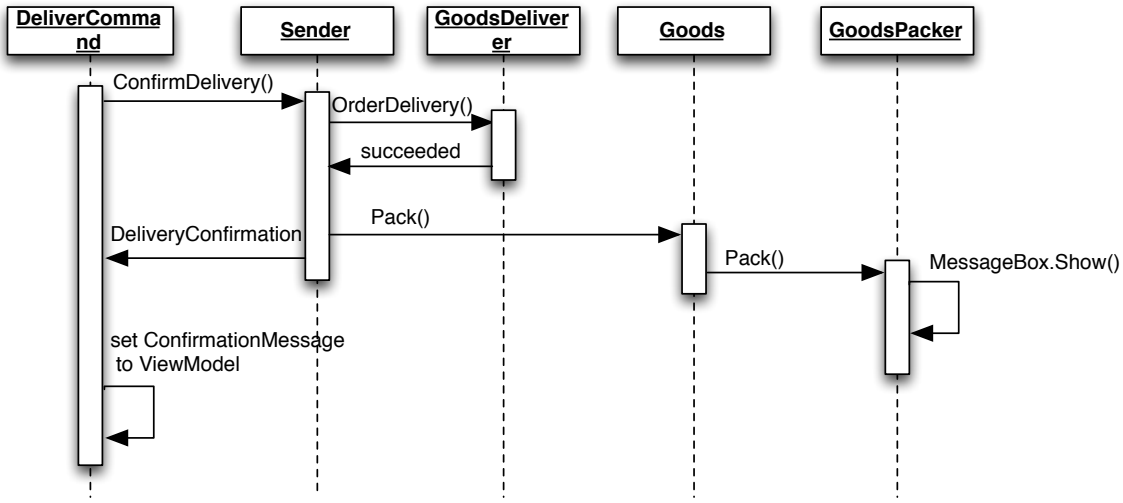
den arvoa kysellään Goods-oliolta sen sijaan, että pakattava asia tietäisi itse pakkaajansa. Goods-luokan alustajaan voidaan korvata FitsToABox-propertyn arvon välitys GoodsPacker-rajapinnan toteuttajan välityksellä. Voidaan luottaa siihen, että Goods-olion luojalla on tieto siitä, millainen pakkaaja kyseiselle tavaralle tarvitaan. Käytännössä sama tieto välitettiin aikaisemmin FitsToABox-ominaisuuden arvoksi. Koodiesimerkissä 23 on esitetty luokkien Goods ja Sender toteutukset tärkeimmiltä osin muutoksen jälkeen.

```
public class Goods
{
    private GoodsPacker packer;
    public string Description { get; set; }

    public Goods(string description, GoodsPacker packer)
    {
        this.packer = packer;
        Description = description;
    }

    public void Pack()
    {
        packer.Pack(this);
    }
}

public class Sender
{
    public void ConfirmDelivery(Goods goods)
    {
        if (deliverer.OrderDelivery(goods))
        {
            orderConfirmed = true;
            PackGoods(goods);
        }
        else
            orderConfirmed = false;
    }
```

```
private void PackGoods(Goods goods)
    {
        goods.Pack();
     }
}
```

Koodiesimerkki 23. Goods-luokka tietää oman pakkaajansa. Sender-luokka kutsuu suoraan Goods-luokan Pack()-metodia. Senderin ei tarvitse välittää, miten tavara pakataan.

Edellisessä kappaleessa tehdyn muutoksen lisäksi myös ViewModel-luokkaan pitää tehdä muutoksia, koska Goods-luokan alustajan signatuuri muuttui. Aikaisemmin ViewModel-luokan alustajassa haettiin GoodsService-luokan GetGoods()-metodin kautta näytöllä näytettävät tavarat. Edellä olevan muutoksen tarkoituksena on kuitenkin mahdollistaa GoodsPacker-rajapinnan toteuttajan korvaaminen tekopalvelulla testitapauksissa. Tämä tarkoittaa sitä, että GoodsService-luokkassa olevia Goods-olioiden luontilauseita pitää muuttaa vastaamaan uutta alustajasignatuuria. Testitapausten kannalta on myös tärkeää, että testeissä käytetään testin kannalta oleellisia Goods-olioita, eli sellaisia, joiden GoodsPacker-toteuttaja on tekopalvelu. Tämän vuoksi ViewModel-luokan alustajaan välitetään halutut Goods-oliot. Koodimuutokset ovat liitteenä. Kuva 9 selventää vielä luokkien yhteistyötä.



Kuva 9. Sekvenssikaavio.

Kuvan 9 sekvenssikaaviosta nähdään luokkien yhteistoiminta. Käyttäjän painaessa Confirm-painiketta kutsutaan DeliverCommand-luokan Execute()-metodia, jonka sisältä toimintaketju käynnistyy. DeliverCommand-luokka kutsuu Sender-luokan ConfirmDeli-

very()-metodia, joka edelleen kutsuu GoodsDeliverer-luokan toteuttajan OrderDelivery()-metodia toimituksen vahvistamiseen. Mikäli toimitus vahvistetaan onnistuneesti, Sender kutsuu Goods-luokan Pack()-metodia, joka edelleen välittää kutsun GoodsPacker-rajapinnan toteuttajalle. GoodsPacker-rajapinnan toteuttaja pakkaa tavaran eli lähettää ilmoitusviestin pakkauksen onnistumisesta MessageBox.Show()-metodin kautta. Lopulta DeliverCommand-luokka päivittää näytölle vahvistuksen tilauksen onnistumisesta kysymällä Sender-luokalta vahvistusviestin.

Yllä olevien muutosten jälkeen testitapaukset voidaan korjata niin, ettei pakkaamisen ilmoitusviesti enää piirry näytölle. Testitapauksessa ViewModel-luokalle välitetään GetListOfGoogs()-metodissa luodut Goods-oliot. Goods-olioihin välitettävä GoodsPacker-rajapinnan toteuttaja on FakeItEasy-kehyksen avulla luotu matkija [20]. Näin ollen Pack()-metodin kutsuminen ohjautuu tyhjään toteutukseen. Koodiesimerkissä 24 on esitetty ensimmäinen testitapaus.

```
[TestMethod]
public void ConfirmationMessageShowDomesticDeliverer()
{
    ViewModel model = new ViewModel(GetListOfGoods());
    DeliverCommand command = new DeliverCommand(model);
    command.Execute(null);
    Assert.AreEqual("Itella delivery confirmed.",
                                model.ConfirmationMessage);
}

private static IList<Goods> GetListOfGoods()
{
    IList<Goods> listOfGoods = new List<Goods>();
    var packer = A.Fake<GoodsPacker>();
    Goods goods = new Goods("TestGoods", packer);
    listOfGoods.Add(goods);
    return listOfGoods;
}
```

Koodiesimerkki 24. Muutettu testitapaus.

Riippuvuusinjektion avulla esimerkkisovelluksen arkkitehtuuri parani. Goods-luokan ei ole tarpeellista paljastaa FitsToABox-ominaisuutta, koska luokka tietää oman pakkaa-

jansa. Näin GoodsPacker-riippuvuus voitiin siirtää Sender-luokasta Goods-luokkaan. Goods-olion luojan kannalta tarvittava tieto luotavasta oliosta ei kasvanut, sillä FitsToABox-ominaisuuden arvo on käytännössä sama tieto kuin tieto tarvittavasta GoodsPacker-toteuttajasta. GoodServicen GetGoods()-metodikutsun siirtäminen ViewModel-luokan alustajasta ViewModel-luokan luonnin yhteyteen toteuttaa myös riippuvuusinjektion periaatteen. Näin testitapauksissa voidaan testata vain ennalta määrätyillä Goods-olioilla. Tällä muutoksella on myös positiivinen sivuvaikutus: GoodsService voidaan vaihtaa oikeaan tietokantaan ilman, että ViewModel-luokkaa pitää muuttaa.

Riippuvuusinjektion toteutukseen ei yksinkertaisimmillaan tarvita kolmannen osapuolen säiliöitä. Riippuvuusinjektion toteuttaminen tarkoittaa sille ominaisten suunnitteluperiaatteiden noudattamista eli sitä, että riippuvuudet välitetään niitä tarvitsevalla osapuolelle sen sijaan, että sen tulisi ne itse luoda tai pyytää sitä muualta. Toisaalta myös kolmannen osapuolen säiliöiden käyttäminen vaatii sitä, että sovelluksen arkkitehtuuri noudattaa näitä samoja riippuvuusinjektion periaatteita. Säiliöistä saatava lisähyöty liittyy olioiden luontiin, niiden elinkaareen ja näkyvyysalueeseen. Nämä seikat huomioon ottaen, kolmannen osapuolen riippuvuusinjektiosäiliöt jätettiin ratkaisun ulkopuolelle, sillä niiden käytöstä ei tässä esimerkkisovelluksesta ollut merkittävää hyötyä. [10.]

9 Päätelmät

Riippuvuusinjektion pääperiaate on itsessään yksinkertainen: kuluttajan ja palvelun välinen suora riippuvuus rikotaan välittämällä kuluttajalle sen tarvitsemat palvelut joko luonnin yhteydessä alustajassa tai tarvittaessa myöhemmin settereissä. Riippuvuusinjektio voidaan toteuttaa käsin tai kolmannen osapuolen ohjelmistokehyksen avulla. Käytettäessä jälkimmäistä vaihtoehtoa voidaan muutoksienhallinta keskittää mahdollisimman pieneen osaan sovellusta. Lisäksi suurin osa vastaavista ohjelmistokehyksistä tarjoaa kokonaisvaltaisen riippuvuusinjektioratkaisun, jossa myös palveluiden koko elinkaari on kehyksen hallinnassa.

Riippuvuusinjektiolla saavutetaan edellä mainitut sovelluskehityksen kulmakivet: toiminnallisuuskeskeisyys, modulaarisuus ja testattavuus. Käytettäessä erillistä ohjelmistokehystä riippuvuuksien hallintaan, voidaan sovelluskehityksessä keskittyä toimintalogiikan toteuttamiseen infrastruktuuriaspektien sijaan. Käytettävää palvelua voidaan tarvittaessa muuttaa, koska yksittäiset palvelut ovat omissa moduuleissaan. Modulaari-

suus taas parantaa testattavuutta. Moduuleja voidaan korvata toisilla moduuleilla, jolloin toiminnallisuus on testattavissa näkökulmista riippumatta.

Riippuvuusinjektion käyttäminen on osa esimerkillistä sovelluskehitystä, sillä se mahdollistaa löyhien sidosten syntymisen. Sen toteuttaminen vaatii lisäpanostusta, mutta käytettäessä esimerkiksi TDD:tä, ohjaudutaan sen käyttämiseen luonnostaan, sillä riippuvuusinjektion avulla luokkia voidaan testata eristyksissä. Kolmannen osapuolen ohjelmistokehykset auttavat olioiden elinkaaren ja näkyvyysalueen hallinnassa, mutta niiden käyttö ei ole riippuvuusinjektion ehto. Päinvastoin, tällaiset kehykset olettavat, että sovelluksen arkkitehtuuri noudattaa riippuvuusinjektion periaatteita.

Ei ole olemassa niin pientä sovellusta, johon riippuvuusinjektiota ei voisi käyttää. Sen toteuttaminen tosin lisää sovelluksen kompleksisuutta, mutta samanaikaisesti auttaa sen ylläpitämisessä, kun sovellus laajenee. Tärkeintä on miettiä, mikä osuus sovelluksesta voi muuttua tulevaisuudessa ja keskittyä implementoimaan nykyinen toteutus niin, että muutoksiin voidaan vastata.

10 Yhteenveto

Opinnäytetyössä esiteltiin riippuvuusinjektion periaatetta, sen hyödyntämistä, etuja ja implementoinnin haasteita. Implementoinnin motivoinniksi esiteltiin hyvän olio-ohjelmoinnin suunnitteluperiaatteita, jotka kaikki johdattivat käyttämään riippuvuusinjektiota. Lisäksi pyrittiin havainnollistamaan esimerkkisovelluksen avulla, miten riippuvuuksien aiheuttamaa testattavuuden ongelmaa voidaan ratkaista riippuvuusinjektion avulla.

Riippuvuusinjektion toteutustapoja käytiin läpi esimerkkien avulla. Kolmannen osapuolen riippuvuusinjektiosäiliöiden, kuten Castle Windsor ja Unity, toimintaa esiteltiin ja käytiin läpi niitä tilanteita, joissa niitä kannattaa hyödyntää. Esimerkkisovelluksen yhteydessä todettiin, ettei riippuvuusinjektion toteutus vaadi kolmannen osapuolen ohjelmistokirjastojen käyttöä, eikä niiden käyttö aina tuo riittävästi lisäarvoa. Tästä syystä esimerkkisovelluksen ratkaisuun ei sisällytetty riippuvuusinjektiosäiliötä.

Esimerkkisovelluksen testattavuuden ongelman ratkaisun kautta todettiin, että riippuvuusinjektion avulla sovelluksen arkkitehtuurista tuli testattavampi. Testattavuuden kannalta haitallinen sivuvaikutus eli ilmoitusviesti paketoinnin onnistumisesta saatiin

hävitettyä muuttamalla arkkitehtuuria soveltamaan riippuvuusinjektion periaatteita sekä käyttämällä testissä matkijaa korvaamaan riippuvuudet. Samalla edesautettiin sovelluksen luokkien kapselointia, koska yksittäisen luokan paljastavien julkisten ominaisuuksien määrä väheni.

Lähteet

1 Google I/O 2009 – Big Modular Java with Guice. [video] Viitattu 29.9.2010. Saatavissa: http://code.google.com/p/google-guice/.

2 Fowler, Martin. Inversion of Control. [verkkodokumentti] 26.6.2005 [Viitattu 29.9.2010]. Saatavissa: http://martinfowler. com/bliki/InversionOfControl.html.

3 Fowler, Martin. Inversion of Control Containers and the dependency Injection Pattern. [verkkodokumentti] 23.1.2004 [Viitattu 29.9.2010] Saatavissa: http://martinfowler.com/articles/injection.html#InversionOfControl.

4 Google. Guice. [verkkodokumentti] [Viitattu 20.10.2010] Saatavissa: http://code.google.com/p/google-guice/.

5 PicoContainer Committers. What is PicoContainer. [verkkodokumentti] [Viitatttu 20.10.2010] Saatavissa: http://www.picocontainer.org/index.html.

6 Hellesoy, Aslak; Tirsen, Jon. Introduction. [verkkodokumentti] [Viitattu 20.10.2010] Saatavissa: http://www.picocontainer.org/introduction.html.

7 Java 2 – Ohjelmoinnin peruskirja. Kosonen, Pekka; Peltomäki, Juha; Silander, Simo. WSOY, 2008, 642 sivua.

8 Prasanna, Dhanji R. Dependency Injection – Design patterns using Spring and Guice. Manning Publications Co, 2009.

9 Sun Microsystems, Inc. Core J2EE Patterns - Service Locator. [verkkodokumentti] [Viitattu 22.11.2010] Saatavissa: http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.html.

10 Seeman, Mark. Dependency Injection in .NET. Manning Publications Co, 2012.

11 Martin, Robert C. Clean Code, A Handbook of Agile Software Craftsmanship. Pearson Education Inc. Prentice Hall, 2010.

12 Feathers, Michael C. Working Effectively with Legacy Code. Pearson Education Inc. Prentice Hall, 2011.

13 Martin, Robert C; Martin, Micah. Agile Principles, Patters and Practices in C#. Pearcon Education Inc. Prentice Hall, 2006.

14 Miller, Jeremy. Patters in Practice: The Open Closed Principle. MSDN Magazine 5, 2008. [Viitattu 17.03.2013] Saatavissa: http://msdn.microsoft.com/en-us/magazine/cc546578.aspx.

15 SOLID (object oriented design). [Viitattu 03.04.2013] Saatavissa: http://en.wikipedia.org/wiki/SOLID_(object-oriented_design).

16 Spaghetti code. [Viitattu 03.04.2013] Saatavissa: http://en.wikipedia.org/wiki/Spaghetti_code.

17 Global Variables Are Bad. [Viitattu 04.04.2013] Saatavissa: http://c2.com/cgi/wiki?GlobalVariablesAreBad.

18 SpringSource Community. [Viitattu 04.04.2013]. Saatavissa: http://www.springsource.org/.

19 Castle Windsor. [Viitattu 04.04.2013] Saatavissa: http://docs.castleproject.org/Windsor.MainPage.ashx.

20 FakeItEasy. [Viitattu 04.04.2013] Saatavissa: https://github.com/FakeItEasy/FakeItEasy.

21 Koskela, Lasse. Test Driven. Practical TDD and Acceptance TDD for Java Developers. Manning Publications C, 2007.

22 Unity. [Viitattu 7.4.2013] Saatavissa: http://unity.codeplex.com/.

23 Windows Presentation Foundation. [Viitattu 7.4.2013] Saatavissa: http://msdn.microsoft.com/en-us//library/ms754130.aspx.

DeliveryApplication-projektin koodi

```
public interface GoodsDeliverer
{
    bool OrderDelivery(Goods goods);
    string Deliver(Goods goods);
    string NameOfDeliverer { get; }
}

public class LocalDeliverer : GoodsDeliverer
{
    public string Deliver(Goods goods)
    {
        return "Local delivery delivered.";
    }

    public bool OrderDelivery(Goods goods)
    {
        return true;
    }

    public string NameOfDeliverer { get { return "Itella"; } }
}

public class InternationalDeliverer : GoodsDeliverer
{
    public string Deliver(Goods goods)
    {
        return "International delivery delivered.";
    }

    public bool OrderDelivery(Goods goods)
    {
        return true;
    }

    public string NameOfDeliverer { get { return "UPS"; } }
}

public class DelivererResolver
{
    public GoodsDeliverer ResolveDeliverer(
                    Destination deliveryDestination)
    {
        switch (deliveryDestination)
        {
            case Destination.Domestic:
                return new LocalDeliverer();
            case Destination.International:
                return new InternationalDeliverer();
            default:
                return new LocalDeliverer();
```

```
        }
    }
}

public class Goods
{
    private GoodsPacker packer;
    public string Description { get; set; }

    public Goods(string description, GoodsPacker packer)
    {
        this.packer = packer;
        Description = description;
    }

    public void Pack()
    {
        packer.Pack(this);
    }
}

public enum Destination
{
    Domestic,
    International
}

public class GoodsService
{
    public static IList<Goods> GetGoods()
    {
        IList<Goods> goodsList = new List<Goods>();
        var goods = new Goods("Food", new CartonPacker());
        goodsList.Add(goods);
        goods = new Goods("Metal", new ContainerPacker());
        goodsList.Add(goods);
        goods = new Goods("Soil", new ContainerPacker());
        goodsList.Add(goods);
        goods = new Goods("Clothes", new CartonPacker());
        goodsList.Add(goods);
        return goodsList;
    }
}

public interface GoodsPacker
{
    void Pack(Goods goods);
}

public class CartonPacker : GoodsPacker
{
    public void Pack(Goods goods)
    {
        // Models a call to external service.
        MessageBox.Show("Goods packed to a carton box.");
```

```
    }
}

public class ContainerPacker : GoodsPacker
{
    public void Pack(Goods goods)
    {
        // Models a call to external service.
        MessageBox.Show("Goods packed to a container.");
    }
}

/**
 * Models a service dependency. Calling ConfirmDelivery
 * in real life situation would be costly.
 * */
public class Sender
{
    private GoodsDeliverer deliverer;
    private bool orderConfirmed;

    public Sender(GoodsDeliverer deliverer)
    {
        this.deliverer = deliverer;
    }

    public void ConfirmDelivery(Goods goods)
    {
        if (deliverer.OrderDelivery(goods))
        {
            orderConfirmed = true;
            PackGoods(goods);
        }
        else
            orderConfirmed = false;
    }

    private void PackGoods(Goods goods)
    {
        goods.Pack();
    }

    public string DeliveryConfirmation
    {
        get
        {
            return orderConfirmed ?
                deliverer.NameOfDeliverer + "
          delivery confirmed." :
          "Delivery confirmation failed.";
        }
    }
}
```

DeliveryApplicationUI-projektin koodi

```
public class DeliverCommand : ICommand
{
    private ViewModel model;
    private DelivererResolver resolver;

    public DeliverCommand(ViewModel model)
    {
        this.model = model;
        resolver = new DelivererResolver();
    }

    public bool CanExecute(object parameter)
    {
        return true;
    }

    public event EventHandler CanExecuteChanged;

    public void Execute(object parameter)
    {
        model.ShowProgress = Visibility.Visible;

        var sender = new Sender(ResolveDeliverer());
        ConfirmDelivery(sender);
        ShowDeliveryConfirmation(sender);

        model.ShowProgress = Visibility.Collapsed;
    }

    private void ConfirmDelivery(Sender sender)
    {
        MakeProgress(50); // fake processing time
        sender.ConfirmDelivery(model.SelectedGoods);
        MakeProgress(100); // fake processing time
    }

    private void MakeProgress(int progress)
    {
        model.Progress = progress;
        Wait(progress * 10);
    }

    private void ShowDeliveryConfirmation(Sender sender)
    {
        model.ConfirmationMessage = sender.DeliveryConfirmation;
    }

    private GoodsDeliverer ResolveDeliverer()
    {
        GoodsDeliverer deliverer = null;

        if (model.SelectedDestination ==
```

```
                    Destination.Domestic.ToString())
            deliverer =
                    resolver.
                    ResolveDeliverer(Destination.Domestic);
        else
            deliverer =
                    resolver.
                    ResolveDeliverer(Destination.International);
        return deliverer;
    }

    private void Wait(int interval)
    {
        ExecuteWait(() => Thread.Sleep(interval));
    }

    private void ExecuteWait(Action action)
    {
        var waitFrame = new DispatcherFrame();
        IAsyncResult result = action.BeginInvoke(
                    dummy => waitFrame.Continue = false, null);
        Dispatcher.PushFrame(waitFrame);
        action.EndInvoke(result);
    }
}

public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        DataContext = new ViewModel(GoodsService.GetGoods());
    }
}

public class ViewModel : INotifyPropertyChanged
{
    private IList<Goods> goods;
    private IList<string> destinations;
    private Destination selectedDestination;
    private Goods selectedGoods;
    private string deliverer;
    private Visibility showProgress;
    private int progress;
    private ICommand deliverCommand;
    private string confirmationMessage;

    public ViewModel(IList<Goods> goods)
    {
        Goods = goods;
        Destinations = new List<string>() {
                    Destination.International.ToString(),
                    Destination.Domestic.ToString() };
        SelectedDestination = Destination.Domestic.ToString();
        SelectedGoods = Goods.First();
```

```
    Deliverer = "Itella";
    ShowProgress = Visibility.Collapsed;
    DeliverCommand = new DeliverCommand(this);
}

public IList<Goods> Goods
{
    get { return goods; }
    set
    {
        if (value != goods)
        {
            goods = value;
            NotifyOfPropertyChange("Goods");
        }
    }
}

public IList<string> Destinations
{
    get { return destinations; }
    set
    {
        if (value != destinations)
        {
            destinations = value;
            NotifyOfPropertyChange("Destinations");
        }
    }
}

public string SelectedDestination
{
    get { return selectedDestination.ToString(); }
    set
    {
        if (value != selectedDestination.ToString())
        {
            selectedDestination =
      ResolveSelectedDestination(value.ToString());
            var deliverer =
                new DeliverерResolver().
                ResolveDeliverer(selectedDestination);
            Deliverer = deliverer.NameOfDeliverer;
            NotifyOfPropertyChange("SelectedDestination");
        }
    }
}

public Visibility ShowProgress
{
    get { return showProgress; }
    set
    {
        if (value != showProgress)
```

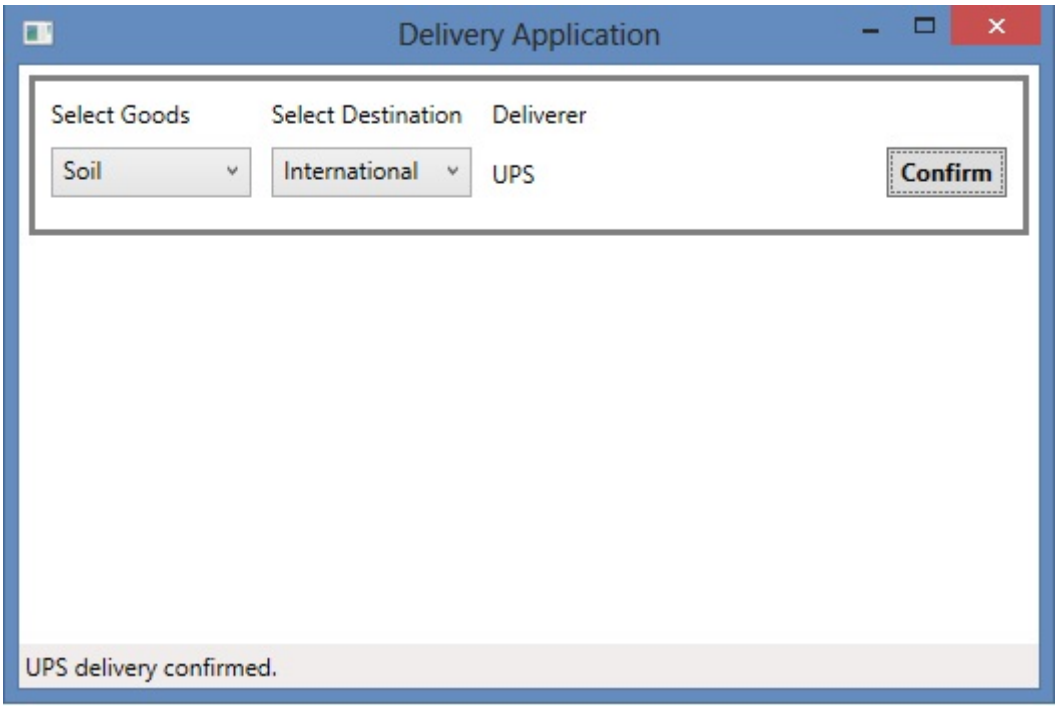
```
        {
            showProgress = value;
            NotifyOfPropertyChange("ShowProgress");
        }
    }
}

public int Progress
{
    get { return progress; }
    set
    {
        if (value != progress)
        {
            progress = value;
            NotifyOfPropertyChange("Progress");
        }
    }
}

public string ConfirmationMessage
{
    get { return confirmationMessage; }
    set
    {
        if (value != confirmationMessage)
        {
            confirmationMessage = value;
            NotifyOfPropertyChange("ConfirmationMessage");
        }
    }
}

public Goods SelectedGoods
{
    get { return selectedGoods; }
    set
    {
        if (value != selectedGoods)
        {
            selectedGoods = value;
            NotifyOfPropertyChange("SelectedGoods");
        }
    }
}

public ICommand DeliverCommand
{
    get { return deliverCommand; }
    set
    {
        if (value != deliverCommand)
        {
            deliverCommand = value;
            NotifyOfPropertyChange("DeliverCommand");
```

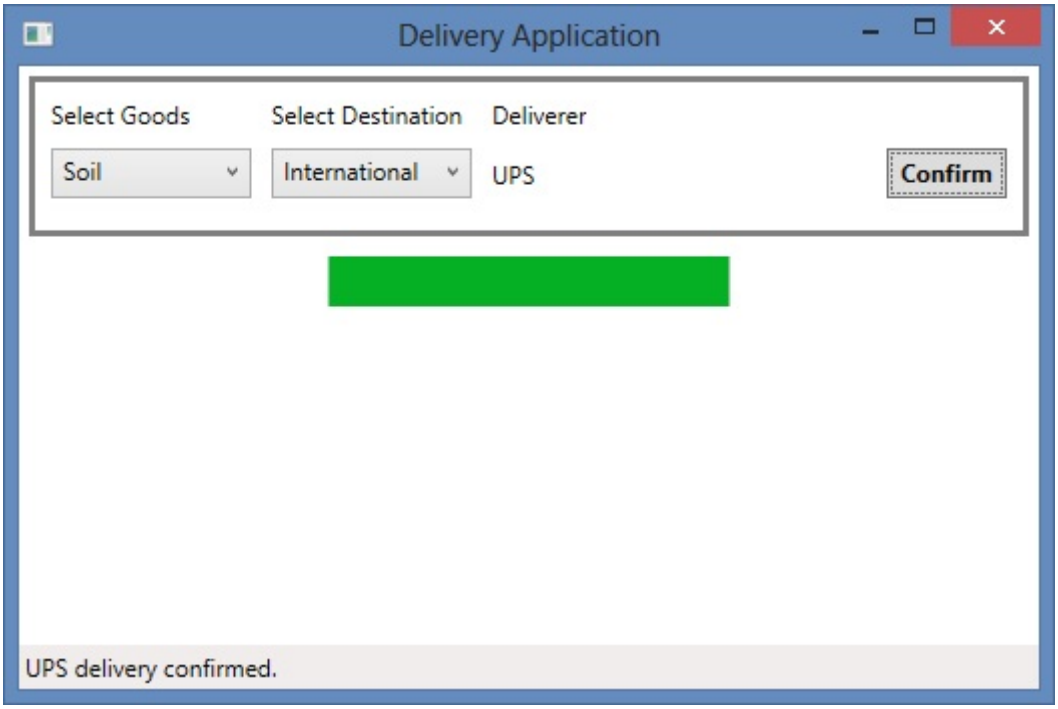
```
            }
        }
    }

    public string Deliverer
    {
        get { return deliverer; }
        set
        {
            if (value != deliverer)
            {
                deliverer = value;
                NotifyOfPropertyChange("Deliverer");
            }
        }
    }

    private Destination ResolveSelectedDestination(
                                        string destination)
    {
        if (destination == Destination.International.ToString())
            return Destination.International;
        else
            return Destination.Domestic;
    }

    public event PropertyChangedEventHandler PropertyChanged;

    public void NotifyOfPropertyChange(string property)
    {
        if (PropertyChanged != null)
            PropertyChanged(this,
                    new PropertyChangedEventArgs(property));
    }
}
```

DeliveryApplicationTests-projektin koodi

```
[TestClass]
public class DeliverCommandTest
{
    [TestMethod]
    public void ConfirmationMessageShowDomesticDeliverer()
    {
        ViewModel model = new ViewModel(GetListOfGoods());
        DeliverCommand command = new DeliverCommand(model);
        command.Execute(null);
        Assert.AreEqual("Itella delivery confirmed.",
                    model.ConfirmationMessage);
    }

    [TestMethod]
    public void ConfirmationMessageShowInternationalDeliverer()
    {
        ViewModel model = new ViewModel(GetListOfGoods());
        model.SelectedDestination = "International";
        DeliverCommand command = new DeliverCommand(model);
        command.Execute(null);
        Assert.AreEqual("UPS delivery confirmed.",
                    model.ConfirmationMessage);
    }

    private static IList<Goods> GetListOfGoods()
    {
        IList<Goods> listOfGoods = new List<Goods>();
        var packer = A.Fake<GoodsPacker>();
        Goods goods = new Goods("TestGoods", packer);
        listOfGoods.Add(goods);
        return listOfGoods;
    }
}
```

Kuva esimerkkisovelluksen käyttöliittymästä





Alkuperäiset luokat oleellisin osin ennen refaktorointia

```
public class Goods
{
    // ---

    public bool FitsToABox { get; private set; }
    public string Description { get; set; }

    public Goods(string description,
          Destination destination, bool fitsToABox = true)
    {
        this.destination = destination;
        Description = description;

        FitsToABox = fitsToABox;
    }

    // ---
}

public class GoodsService
{
    public static IList<Goods> GetGoods()
    {
        IList<Goods> goodsList = new List<Goods>();
        var goods = new Goods("Food", Destination.Domestic);
        goodsList.Add(goods);
        goods = new Goods("Metal",
                              Destination.International, false);
        goodsList.Add(goods);
        goods = new Goods("Soil", Destination.Domestic, false);
        goodsList.Add(goods);
        goods = new Goods("Clothes", Destination.International);
        goodsList.Add(goods);
        return goodsList;
    }
}

public class Sender
{
    // ---

    public Sender(GoodsDeliverer deliverer)
    {
        this.deliverer = deliverer;
    }


    private void PackGoods(Goods goods)
    {
        if (goods.FitsToABox)
            packer = new CartonPacker();
        else
```

```
            packer = new ContainerPacker();

        packer.Pack(goods);
    }

    // ---
}

public class DeliverCommand : ICommand
{
    // ---

    public void Execute(object parameter)
    {
        model.ShowProgress = Visibility.Visible;

        var sender = new Sender(ResolverDeliverer());
        ConfirmDelivery(sender);
        ShowDeliveryConfirmation(sender);

        model.ShowProgress = Visibility.Collapsed;
    }

    // ---
}

public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        DataContext = new ViewModel();
    }
}

public class ViewModel : INotifyPropertyChanged
{
    // ---

    public ViewModel()
    {
        Goods = GoodsService.GetGoods();
        Destinations = new List<string>() {
                    Destination.International.ToString(),
                    Destination.Domestic.ToString() };
        SelectedDestination = Destination.Domestic.ToString();
        SelectedGoods = Goods.First();
        Deliverer = "Itella";
        ShowProgress = Visibility.Collapsed;
        DeliverCommand = new DeliverCommand(this);
    }

    // ---
}
```

```
[TestClass]
public class DeliverCommandTest
{
    [TestMethod]
    public void ConfirmationMessageShowDomesticDeliverer()
    {
        ViewModel model = new ViewModel();
        DeliverCommand command = new DeliverCommand(model);
        command.Execute(null);
        Assert.AreEqual("Itella delivery confirmed.",
                        model.ConfirmationMessage);
    }

    [TestMethod]
    public void ConfirmationMessageShowInternationalDeliverer()
    {
        ViewModel model = new ViewModel();
        model.SelectedDestination = "International";
        DeliverCommand command = new DeliverCommand(model);
        command.Execute(null);
        Assert.AreEqual("UPS delivery confirmed.",
                        model.ConfirmationMessage);
    }
}
```