UNIVERSITETI I PRISHTINËS "HASAN PRISHTINA"
FAKULTETI I SHKENCAVE MATEMATIKE-NATYRORE
DEPARTAMENTI I MATEMATIKËS
DREJTIMI : SHKENCË KOMPJUTERIKE

PUNIM DIPLOME

"Dependency Injection"

Mentori:
Dr. sc. Faton Berisha

Studenti:
Kushtrim Pacaj

Prishtinë 2016

Abstrakti

Gjatë programimit të aplikacioneve, rëndësi shumë të madhe ka strukturimi i kodit. Një kod mirë i strukturuar është më i kuptueshëm, na lehtëson mirëmbajtjen, na mundëson ndërrimin e moduleve të aplikacionit pa modifikuar pjesë tjera të app-it , si dhe e bën më të lehtë testimin.

Këtu na hynë në punë modeli i *dependency injection*, apo siç njihet ndryshe edhe me termat *inversion of control, Hollywood principle* etj. Koncepti themelor tek *dependency injection* është që klasat të mos merren me krijimin e varësive të tyre, por ato t'iu vijnë të gatshme për përdorim. Në këtë punim diplome do shpjegojmë se pse është i mirë ky pattern i programimit, duke e krahasuar po ashtu me modelet tjera si *ServiceLocator* dhe *Factory pattern*. Më pas do i tregojmë llojet e injektimit, përparësitë dhe mangësitë e secilit lloj, problemet e mundshme që dalin – njëkohësisht edhe zgjidhjet e tyre.

Në pjesën e tretë hyjmë në detajet e JSR 330, që është specifikimi i Java për *dependency injection*, kurse krejt në fund i tregojmë disa nga platformat më të përdorura për këtë pattern, shpjegojmë se si përdoren dhe ku dallojnë nga njëra tjetra sa i përket lehtësisë në konfigurim, shpejtësisë etj. Pas platformave shpjegohet edhe se si mund të bëhet vetë një implementim i thjeshtë i *dependency injection* në gjuhën Java.

Përmbajtja

1. Hyrje

Dependency injection (sq. “Injektimi i varësive”, shkurt: D.I.) është një model (ang. pattern) i cili përdoret në zhvillimin e softuerëve kompjuterik.

Arsyeja kryesore për përdorimin e dependency injection është sepse e bën kodin burimor të aplikacionit më decoupled , d.m.th. klasat nuk i inicializojnë serviset që iu vyejnë, por ato iu vijnë nga DI platforma.

Kjo jo vetëm që e bën kodin më lehtë të menaxhueshëm pasi programerat mund të punojnë ndaras në pjesë të ndryshme të softuerit pa u varur prej njëri-tjetrit, por po ashtu hyn në punë gjatë testimit të tij. P.sh. gjatë unit-testing, nëse duam ta testojmë një klasë specifike, atëherë mund të injektojmë servise/varësi dummy (objekte që gjithmonë kthejnë rezultate të njëjta të para-definuara nga ne) pa ndryshuar asnjë linjë kodi në klasën në fjalë, vetëm duke ndryshuar tek platforma DI konfigurimin se cilat objekte duhet krijuar.

1.1 Use Case dhe konceptet themelore

Fillimisht specifikojmë një shembull të thjeshtë që do ta përdorim më vonë gjatë shqyrtimit të dependency injection.

Supozojmë se duam ta dërgojmë një email te një person tjetër. Atëherë këtë proces mund ta modelojmë si në foton më poshtë:

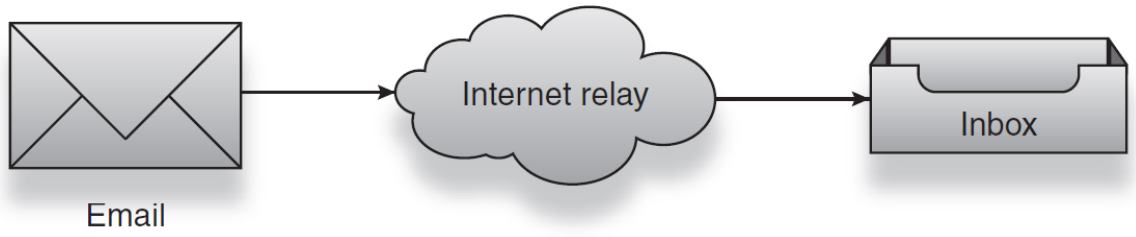

Figura 1 Modeli i dërgimit të një email-i

Një koncept që është me rëndësi ta vërejmë në këtë rast është ideja sjelljes së objekteve si servise. Për shembull, në modelin e mësipërm *Email* mund të mendohet si një servis për kompozim të emailave. Rrjedhimisht *Email* e përdor *InternetRelay* si servis për dërgim të emailave, kurse *InternetRelay* e përdor *Inbox* si servis për pranim të emailës së dërguar.

Edhe puna e krijimit të emailës mund të ndahet në pjesë më të vogla:

1. Shkrimi i tekstit
2. Kontrollimi për gabime drejtshkrimore
3. Shkrimi i adresës së pranuesit (ose kërkimi i adresës – *address lookup*)

Këtë mund ta modelojmë me klasat në vijim: *Emailer* (si wrapper për servis), *TextEditor* (për shkrim të tekstit) , *SpellChecker* (për kontrollim të gabimeve) , *AddressBook* (për kërkim të adresës së pranuesit).

Kur një klasë varet nga klasat tjera për funksionim, themi se ajo është *dependent* nga to. Ngjashëm me *dependency* e nënkuptojmë një klasë që i nevojitet të tjerave që të funksionojnë.

Në këtë rast *Emailer* është i varur nga klasat tjera *(dependent)*, pasi i vyejnë që të funksionojë si duhet, kurse klasat tjera janë *dependencies* të saj.

Varësitë mund të jenë transitive, pasi p.sh *Emailer* varet nga *TextEditor*, por edhe *TextEditor* ka mundësi të ketë *dependencies* të tij, e kështu më radhë.

Ky sistem i objekteve të ndër-varura prej njëra-tjetrës quhet **graf i objekteve**.

1.2 Konstruktimi manual i grafit të objekteve

Duke u bazuar në modelin e mësipërm, nëse duam të ndërtojmë një program për dërgim të emailave që po ashtu bën kontrollim të tekstit, atëherë një prej mënyrave më të thjeshta do ishte:

```
public class Emailer {

    SpellChecker spellChecker = new SpellChecker();
    //variablat tjera

    public Emailer() {
        //inicializojme variablat tjera këtu
    }

    //metodat tjera
    public void sendEmail(String text) {
        if (spellChecker.spellcheck(text)) {
            //kodi për dërgim të emailit
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Emailer emailer = new Emailer();
        emailer.sendEmail("Wubalubadubdub");
    }
}
```

Ku është gabimi këtu ? Askund. Ky kod kompajllohet dhe funksionon.

Por çka nëse vendosim që ta bëjmë aplikacionin tonë t'i përkrahë 2 gjuhë, shqip dhe anglisht ?! Atë herë na duhet t'i shkruajmë 2 kontrollues të drejtshkrimit: *AlbanianSpellChecker* dhe *EnglishSpellChecker*, dhe një interface *SpellChecker* të cilën e implementojnë të dy klasat konkrete.

```
public interface SpellChecker {

     boolean spellcheck(String text);
}
-----------------------------------------------------------------------------------
public class EnglishSpellChecker implements SpellChecker {
    @Override
    public boolean spellcheck(String text) {
        boolean valid = false;
        //the code that does the spellcheck goes here
        return valid;
    }
}
-----------------------------------------------------------------------------------------------------

public class AlbanianSpellChecker implements SpellChecker {
    @Override
    public boolean spellcheck(String text) {
        boolean valid = false;
        //the code that does the spellcheck goes here
        return valid;
    }
}
```

Vërejmë se e kemi një problem me shembullin tonë: krijimi i *SpellChecker* është pjesë internale e *Emailer*, është hard-coded në të. *Emailer* e enkapsulon krijimin e varësive të saj. Zgjidhja për këtë do ishte që objekti që e krijon instancën e *Emailer* t'ia dërgojë edhe *SpellChecker-in* e duhur.

```
public class Emailer {

    SpellChecker spellChecker;
    //variablat tjera

    public Emailer(SpellChecker spellChecker) {
        this.spellChecker = spellChecker;
        //inicializojme variablat tjera këtu
    }

    public void sendEmail(String text) {
        if (spellChecker.spellcheck(text)) {
            //kodi për dërgim të emailit
        }
    }

}
-------------------------------------------------------------------------------
public class Main {
    public static void main(String[] args) {
        SpellChecker spellChecker = new EnglishSpellChecker();
        Emailer emailer = new Emailer(spellChecker);
        emailer.sendEmail("Wubalubadubdub");
    }
}
```

Por edhe kjo nuk është zgjidhja më e mirë e mundshme. Konstruktimi i grafit të objekteve po bëhet manualisht prej klientit të servisit (krijuesi i objektit *Emailer*). Në këtë rast klientit duhet t'i dijë specifikat se si ndërtohet komplet grafi i objektit.

Rasti në fjalë është shembull i vogël, e me 2 linja kod po krijohet komplet grafi, po në aplikacione reale, ky numër ka gjasa të jetë shumë më i madh sa aq. P.sh. në njërin prej aplikacioneve për Android të kompanisë Google, janë rreth 3000 linja kodi që merren vetëm me ndërtim të grafit të objekteve dhe dhënien e dependencies aty ku ka nevojë. Kurse për një aplikacion të madh server-side, ky numër hip në ~ 100,000.[1]

Pra konstruktimi manual i grafit nga klienti, edhe pse mund të kryejë punë për aplikacione të vogla, nuk është lehtë i menaxhueshëm për aplikacione të mëdha.

Më poshtë shohim 3 pattern-a që mund të na hyjnë në punë për menaxhim të grafit.

1. Factory pattern (sq. "modeli i fabrikës së objekteve")
2. Service Locator pattern (sq. "modeli i lokatorit të serviseve")
3. Dependency injection (sq. "modeli i injektimit të varësive")

[1] https://youtu.be/oK_XtfXPkqw?t=305

1.3 Factory pattern

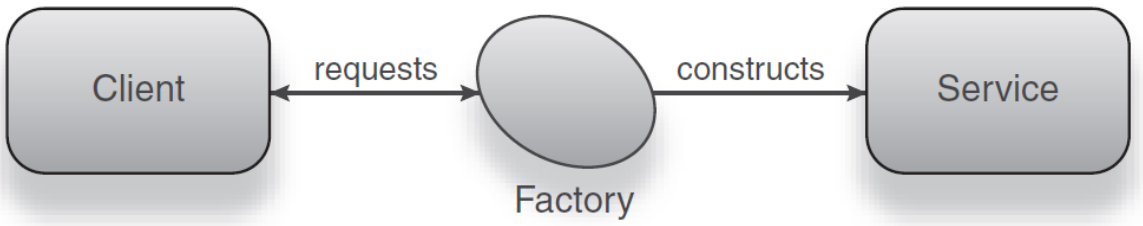

Figura 2 Factory pattern

Ideja bazë te ky model është që përgjegjësinë e krijimit të servisit që na nevojitet (rrjedhimisht edhe krijimit të dependencies të tij) ta ketë një klasë speciale që do ta quajmë fabrikë *(ang. factory)*. [4]

Në rastin tonë specifik do ta kishim një fabrikë të objekteve *Emailer*, pra një *EmailerFactory*.

```
public class EmailerFactory {

    public  Emailer newEnglishEmailer() {
        EnglishSpellChecker englishSpellChecker = new EnglishSpellChecker();
        return new Emailer(englishSpellChecker);
    }

    public  Emailer newAlbanianEmailer() {
        AlbanianSpellChecker albanianSpellChecker = new AlbanianSpellChecker();
        return new Emailer(albanianSpellChecker);
    }

}
```

Prapë, në rastet elementare ky pattern na kryen punë, por problemet me të i vërejmë në use-cases më të komplikuara.

P.sh. nëse kemi vendosim që *Emailer* të jetë më modular, të mos punojë me një *TextEditor* të thjeshtë, por të pranojë instanca të llojllojshme, atëherë e shohim problemin me *factory pattern:* na duhet ta përkrahim çdo kombinim të mundshëm të varësive të saj .

```
public class EmailerFactory {

    public Emailer newSimpleEnglishEmailer() {
        EnglishSpellChecker englishSpellChecker = new EnglishSpellChecker();
        SimpleTextEditor simpleTextEditor = new SimpleTextEditor();
        return new Emailer(englishSpellChecker, simpleTextEditor);
    }
```

```
    public Emailer newAdvancedEnglishEmailer() {
        EnglishSpellChecker englishSpellChecker = new EnglishSpellChecker();
        AdvancedTextEditor advancedTextEditor = new AdvancedTextEditor();
        return new Emailer(englishSpellChecker, advancedTextEditor);
    }

    public Emailer newSimpleAlbanianEmailer() {
        AlbanianSpellChecker albanianSpellChecker = new AlbanianSpellChecker();
        SimpleTextEditor simpleTextEditor = new SimpleTextEditor();
        return new Emailer(albanianSpellChecker,simpleTextEditor);
    }

    public Emailer newAdvancedAlbanianEmailer() {
        AlbanianSpellChecker albanianSpellChecker = new AlbanianSpellChecker();
        AdvancedTextEditor advancedTextEditor = new AdvancedTextEditor();
        return new Emailer(albanianSpellChecker,advancedTextEditor);
    }

}
```

Rrjedhimisht klasa *EmailerFactory* bëhet shumë e madhe, dhe jo shumë e mirëmbajtshme.

1.4 Service locator pattern

Service Locator është një model i ngjashëm me modelin *Factory*, sa që mund të themi edhe se është një lloj i tij, por që ndryshon në disa aspekte nga modeli "standard" fabrikë.

Një prej ndryshimeve te ky model është mënyra e kërkimit të një servisi, që bëhet me anë të një *çelësi*, për ndryshim prej thirrjes së metodave që i pamë më sipër tek klasa *EmailerFactory*. Po ashtu në këtë model e kemi vetëm një klasë që mund të përmbajë çfarëdo servisi, e jo nga një fabrikë për secilin servis ndaras.

Një shembull i përdorimit të një kërkese për servisin *Emailer* do ishte:

```
public class Main {
    public static void main(String[] args) {
        Emailer emailer = (Emailer) new ServiceLocator().get("simpleAlbEmailer");
        emailer.sendEmail("Wubalubadubdub");
    }
}
```

Por edhe ky model ka probleme:

1. Çelësi është thjeshtë String, dhe shumë lehtë mund të gabohet prej programerëve tjerë.
2. Gabimet i vërejmë vetëm në *runtime*, e jo *compile-time.*
3. Prapë sikur tek *Factory pattern* krijimi i grafit po bëhet manualisht prej nesh, në këtë rast vetëm e kemi zhvendosur krijimin manual prej klientit të servisit, tek klasa *ServiceLocator*.
4. Klienti po ka nevoje të kërkojë në mënyrë eksplicite varësitë.

Dosido, nëse prapëseprapë vendosim ta përdorim këtë model, atëherë Java ofron një API që quhet *Java Naming and Directory Interface (JNDI)* që e përdor këtë pattern.[2]

1.5 Dependency injection

Dependency injection është një prej modeleve më të përdorura për menaxhimin e varësive të objekteve. Principi kryesor tek ky model është *inversion-of-control*, apo delegimi i përgjegjësisë për menaxhimin e varësive nga klientët (përdoruesit) e serviseve në fjalë tek *D.I.* platforma. Rrjedhimisht as klasa varësitë e së cilës po i *injektim*, e as klienti (që e përdorë klasën e cekur me herët) nuk dinë asgjë për ato varësi, nuk dinë as se si krijohen – atë pjesë e menaxhon platforma.[3]

Ky fakt ndihmon shumë në projekte softuerike, pasi na mundëson që ta kemi një ndarje të qartë të pjesës së konfigurimit të lidhjes mes klasave me përdorimin e atyre klasave në projekt.

Në fakt nëse duam ta ndryshojmë një modul të aplikacionit, p.sh. në vend të klasës *SimpleTextEditor*, vendosim ta krijojmë dhe përdorim klasën *RichTextEditor*, atëherë këtë gjë mund ta bëjmë vetëm duke e ndryshuar konfigurimin tek platforma, e pa prekur askund në projekt në vendet ku e përdornim deri më tash *SimpleTextEditor*.

Për më tepër nëse vendosim që *RichTextEditor* varet edhe prej pjesëve tjera si *HtmlParser*, *ImageManager* etj. atëherë ato mund ti shtojmë fare lehtë vetëm duke e *RichTextEditor* që ti përdore edhe DI platformën që ti injektojë – çdo klasë që e përdorë *RichTextEditor* mbetet njëjtë dhe nuk di asgjë për ndryshimet. Po të mos përdornim *Dependency Injection*, atëherë do na duhej manualisht të shkonim te çdo klasë të bënim ndryshimet.

Vërejmë se edhe modeli *ServiceLocator* na ndihmonte me disa prej këtyre gjërave, por prapë se prapë klientit i duhej të kërkojë servisin në mënyrë eksplicite, kurse me *Dependency Injection* këtë gjë e menaxhon platforma.

[2] http://tomcat.apache.org/tomcat-7.0-doc/jndi-resources-howto.html

Pasi *Dependency Injection* është model i njohur që një kohë e gjatë, janë krijuar dhe janë në përdorim në ditët e sotshme platforma të shumta të D.I.

Ndër më të njohurat janë:

1. *Spring Framework*, e krijuar fillimisht nga Rod Johnson në vitin 2002.
2. *Guice*, e krijuar nga Google
3. *Dagger 1*, e krijuar nga Square
4. *Dagger 2*, e krijuar nga Google, e bazuar në v1 të Square.

Për momentin *Dagger 2* është më e reja prej tyre, më e avancuar dhe më lehtë e përdorshme, por në kapitujt në vazhdim do i mbulojmë përparësitë dhe mangësitë e secilës.

2. Mënyrat e injektimit

Në modelin *Dependency Injection* ekzistojnë disa mënyra të injektimit të varësive nga ana e platformës në objektet përkatëse:

1. Constructor injection
2. Setter injection
3. Interface injection
4. Method decoration

Në këtë kapitull do shpjegojmë për secilën se si funksionon, dhe se cilat janë përparësitë dhe mangësitë e secilës.

2.1 Constructor injection

Për ta treguar pse dallon ky tip nga *setter injection* fillimisht duhet të tregojmë vetitë speciale që i ka konstruktori krahas metodave normale.

1. Konstruktor quajmë metodën që nuk kthen asgjë (as *void*), dhe e ka emrin e njëjtë me klasën në të cilën gjendet (mund të ketë parametra nëse ka nevojë).
2. Mund t'i inicializojë variablat finale.
3. Thirret vetëm një herë për instancë.
4. Nuk mund të jetë abstrakte (së paku jo në Java) – nëse deklarohet atëherë duhet implementuar.

Me *constructor injection* thjeshtë nënkuptojmë faktin se klasa që ka varësi, i deklaron ato si parametra të konstruktorit.

Më pas kur ndokush kërkon të krijojë objekt prej kësaj klase, platforma automatikisht i gjen varësitë, i krijon objektet nga ato klasë dhe e starton klasën në fjalë përmes konstruktorit të lartcekur, duke ia dërguar varësitë si parametra.

Mënyra se si e din platforma se cilin konstruktor ta thërrasë, e ku ti gjejë varësitë ndryshon nga platforma në platformë, dhe do diskutohet në kapitujt në vazhdim.

2.2 Setter injection

Në këtë rast injektimi i varësive (krijimi dhe vendosja e tyre në objektin të cilit i duhen) bëhet nga platforma me anë të *setter* metodave.

Vërejmë që në këtë rast injektimi bëhet pas krijimit të instancës së objektit, pasi këto metoda nuk mund të thirren pa u kryer së ekzekutuar konstruktori.

Mënyra se si e di platforma se cilat metoda t'i thërrasë prapë varet prej platformës në fjalë:

1. Disa specifikojnë një standard emërimi të metodave si p.sh. *set +<defined_id>* , ku *defined_id* është një id që definohet diku në konfigurim të platformës
2. Disa kërkojnë që metodat që duam të përdoren të "*shënohen"* me *annotations*, e platforma me anë të parametrit e di se cilin objekt ta dërgojë.

Përderisa mund të argumentojmë se ky model e "mbush" klasën me plot *setter* metoda që nuk na vyejnë më vonë, disa platforma (*Guice etj.)* lejojnë që të deklarohet një metodë e vetme ku parametrat (një apo më shumë) janë varësitë që duam ti injektim. Në këtë rast në *Guice* ajo metodë duhet të *"shënohet"* me @Inject.

2.3 Interface injection

Në këtë mënyrë të injektimit, injektimi bëhet përmes metodave të deklaruara në interfejsa të cilët klasa në fjalë i implementon.

P.sh në *use case*-in tonë nëse klasës *Emailer* nëse dëshirojmë që t'i injektim 2 varësi *SpellChecker* dhe *TextEditor* atëherë deklarojmë interfejsat si më poshtë:

```
public interface SpellCheckerInjectable {
    void use(SpellChecker spellChecker);
}


public interface TextEditorInjectable {
    void use(TextEditor textEditor);
}
```

```
public class Emailer implements SpellCheckerInjectable, TextEditorInjectable {

    SpellChecker spellChecker;
    TextEditor textEditor;

    @Override
    public void use(SpellChecker spellChecker) {
        this.spellChecker = spellChecker;
    }

    @Override
    public void use(TextEditor textEditor) {
        this.textEditor = textEditor;
    }

    public void sendEmail(String text) {
        if (spellChecker.spellcheck(text)) {
            //kodi për dërgim të emailit
        }
    }
}
```

Një prej disavantazheve të kësaj metode është që nëse na duhet ti injektojmë shumë varësi, atëherë na duhet t'i definojmë shumë interfejsa. Për më tepër definimi (headeri) i klasës bëhet shumë gjatë dhe nuk duket mirë nëse na duhet t'i implementojmë shumë interfejsa (p.sh. 10+).

Për këtë arsye, interface injection nuk përdoret më në platformat moderne, vetëm platforma e vjetër *Apache Avalon* e përdor. [1]

2.4 Method decoration

Method decoration ndryshon nga mënyrat e mësipërme me faktin se nuk e llogarit objektin si gjëja që duhet injektuar, por metodën.

D.m.th. kur këto metoda thirren, ato nuk e kthejnë vlerën normale të tyre, por një objekt që vendos injektori (*D.I platforma*). Nga kjo sjellje edhe e merr emrin *dekorim i metodave*, pasi po e modifikon sjelljen e metodës.

Në këto raste kur e dimë se metoda e deklaruar nga ne do përdoret në këtë mënyre, atëherë mund të kthejmë *null* si vlerë kthyese pa problem, pasi ky rezultat nuk do përdoret ashtu-kështu.

2.5 Krahasimi i mënyrave të injektimit

Asnjëra nga metodat nuk mund të "fitojë" si metoda më e mirë, pasi secila ka përparësitë dhe mangësitë e veta.

Një prej përparësive të *constructor injection* është se kur e përdorim atë mund t'i inicializojmë fushat e deklaruara si finale. Kjo na siguron që ato fusha gjithmonë do mbesin ashtu si i kemi inicializuar, dhe askush gabimisht apo me qëllim nuk mund ta ndryshojë atë variabël. Këtë gjë nuk mund ta bëjmë përmes *setter injection* sepse setters janë metoda të thjeshta.[1]

Po ashtu injektimi i varësive përmes konstruktorit na siguron që objekti i krijuar është i plotë dhe nuk i mungon asnjë *dependency*, përderisa injektimi përmes setters bëhet pas krijimit të objektit, kështu që ka gjasa që të mos shkojë çdo gjë sipas planit:

1. Mund të bëjë ndonjë gabim në konfigurim, dhe edhe pse e kemi metodën e deklaruar në objekt, platforma të mos e thërrasë.
2. Nëse ndokund e bëjmë inicializim direkt të objektit, mund të harrojmë që për ta inicializuar si duhet, duhet thirrur edhe *setter* metodat.
3. Më e keqja e këtyre është se këto gabime do i vërejmë vetëm në runtime, kur tentojmë ta përdorim objektin/varësinë e pa-inicializuar si dihet.

Por kjo nuk do të thotë që çdo fakt e favorizon metodën e *constructor injection*! Fakti që konstruktori është metodë speciale që përdoret për krijim të instancës së klasës na i sjell disa probleme si do të shohim më poshtë.

2.5.1 Problemi i shumë konstruktorëve (apo problemi piramidë)

Problemi piramidë na shfaqet tek constructor injection kur e kemi një klasë që mund të inicializohet me disa kombinime të ndryshme të varësive, duke u varur prej kontekstit një fjalë.

Marrim një *use-case*:

Modern pentathlon është një garë olimpike që përbëhet nga 5 sporte : fencing (me shpata), notim, kërcim, gjuajtje me armë dhe vrapim. Në rastin tonë le të supozojmë se një atlet mund të marrë pjesë me cilëndo kombinim të sporteve p.sh. të notojë dhe të vrapojë, por jo tjerat etj. (*sqarim: në garë aktuale nuk vlen domosdo kjo rregull*).

Atëherë duke u varur nga garat në të cilat atleti dëshiron të marrë pjesë, atij i vyejnë pajisje të ndryshme – d.m.th. ka klasa ka kombinime të ndryshme të varësive.

```
public class Athlete {

    Sword fencingSword;
    Sneakers runningSneakers;
    Pole jumpingPole;
    SwimmingSuit swimmingSuit;
    Gun shootingGun;

    public Athlete(Sword sword, Sneakers sneakers) {
        this.fencingSword = sword;
        this.sneakers = sneakers;
    }

    public Athlete(Sword sword, Gun gun) {
        this.fencingSword = sword;
        this.shootingGun = gun;
    }

    public Athlete(Sword sword, SwimmingSuit swimmingSuit) {
        this.fencingSword = sword;
        this.swimmingSuit = swimmingSuit;
    }

    public Athlete(Sword sword, SwimmingSuit swimmingSuit, Gun gun) {
        this(sword, swimmingSuit);
        this.shootingGun = gun;
    }

    public Athlete(Sword sword, Sneakers sneakers, Pole pole) {
        this(sword, sneakers);
        this.jumpingPole = pole;
    }

   //other constructors with different parameter combinations

}
```

Siç shihet, në këto raste na duhet nga një konstruktor për çdo kombinim të varësive që mund të injektohen. Ky problem në disa raste quhet edhe problemi i piramidës (*constructor pyramid problem*), pasi kemi shumë konstruktorë, ku disa prej tyre kanë nënbashkësi të parametrave të tjetrit (ai me më shumë parametra e përdor atë me më anë të keyword-it *this*) duke ia dhënë formën e trekëndëshit apo piramidës.

Për më tepër, për shkak të vetive të konstruktorit atë nuk mund ta ri-emërojmë me ndonjë emër tjetër më përshkrues për parametrat, por duhet ta lëmë me emër të klasës.

Problemin e piramidës mund ta shmangim nëse përdorim *setter injection*, pasi në atë rast i kemi thjeshtë disa metoda që platforma mund ti thërrasë në çfarëdo kombinimi që duam.

2.5.2 Problemi i referimit reciprok

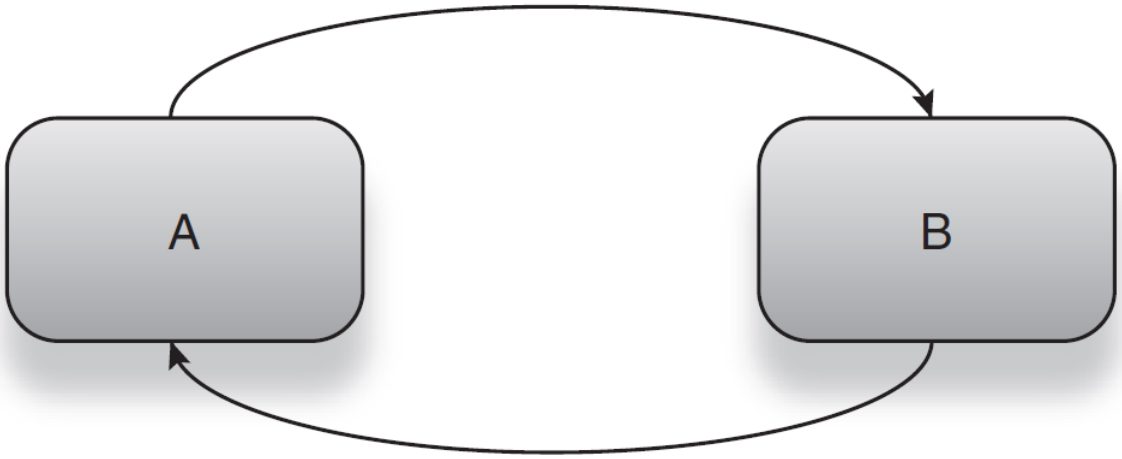

Figura 3 Referimi reciprok në mes klasëve

Problemi i referimit reciprok ndodh kur e kemi një varg të klasave që kanë varësi te njëra tjetra ashtu që formojnë një rreth (d.m.th. klasa 1 ka si varësi të dytën, e dyta të tretën, e kështu më radhë deri te e fundit e cila e ka si varësi klasën e parë). Më poshtë është paraqitur një use-case me 2 klasë.

```
public class A {
    private B b;

    public A(B b) {
        this.b = b;
    }
}
--------------------------------------------------------------------------------
public class B {
    private A a;

    public B(A a) {
        this.a = a;
    }
}
```

Problemi qëndron në faktin se kur *D.I platforma* tenton të krijojë instancë të klasës A, fillimisht e sheh se e ka B si varësi, dhe tenton të krijojë një instancë të klasës B (që ta injektojë në A).

Po prapë kur tenton ta krijojë B, e sheh që kjo klasë e ka si varësi A-në, dhe i duhet fillimisht ta krijojë një instancë të A për ta injektuar në B.

Pra për ta krijuar objektin prej klasës A na nevojitet një objekt B, për ta krijuar B na nevojitet një objekt A, për ta krijuar A na nevojitet B ..... *ad infinitum*.

Pasi ky problem ndodh vetëm kur përdorim *constructor injection*, një zgjidhje e mundshme është që të përdorim *setter injection*. Në atë rast konstruktimi i objekteve do bëhej pa problem, e pas konstruktimit të 2 objekteve, do injektohej referenca e njërit tek tjetri përmes *setter* metodave.

Një mënyrë tjetër e zgjidhjes së këtij problemi është duke përdorur interfejsa dhe një klasë të ndërmjetme:

```
public interface AInterface {
    //public API for A
    void aMethod(Object o);
}
-----------------------------------------------------------------------------------

public interface BInterface {
    //public API for B
    void bMethod(Object o);
}
-----------------------------------------------------------------------------------

public class A implements AInterface {

    private BInterface b;

    public A(BInterface b) {
        this.b = b;
    }

    @Override
    public void aMethod(Object o) {
        //do something
    }
}
-----------------------------------------------------------------------------------

public class B implements BInterface {

    private AInterface a;

    public B(AInterface a) {
        this.a = a;
    }

    @Override
    public void bMethod(Object o) {
        //do something
    }
}
```

```
public class AProxy implements AInterface {
    AInterface a;

    public void setDelegate(AInterface a) {
        this.a = a;
    }

    //pass all call to the 'real' A
    @Override
    public void aMethod(Object o) {
        a.aMethod(o);
    }
}
```

Kurse kur duam t'i krijojmë objektet A dhe B që kanë referencë tek njëra-tjetra:

```
AProxy aProxy = new AProxy();
B b = new B(aProxy);
A a = new A(b);
aProxy.setDelegate(a);
```

Pra B e merr si parametër *proxy-n*, A-ja kur krijohet e merr B-në si parametër në konstruktor, dhe në fund e vendosim objektin e krijuar nga A si delegat te proxy. Në këtë mënyre, *A* ka referencë direkte *B*-në, kurse B e ka *proxy-n* i cili çdo thirrje ia pason A.

Mënyra e konfigurimit varet nga platforma përkatëse. P.sh. *Guice* këtë e zgjidh automatikisht[3] :

Por në disa raste te zgjidhja me proxy e hasim një problem të ri, që në disa literatura quhet *"The In-construction problem"* [4] Ky problem na ndodh kur tek konstruktori i klasës B tentojmë të thërrasim ndonjë metodë të A-së duke përdorur proxy-n e dhënë. Pasi ende as nuk është krijuar objekti A, as nuk i është dhënë proxy-t si delegat, ajo thirrje dështon me *NullPointerException*.

Fatkeqësisht këtë problem nuk mund ta zgjedhim me *constructor injection*, nëse na shfaqet na duhet të përdorim *setter injection* dhe t'i bëjmë thirrjet tek objekti tjetër (në këtë rast A-në) kur të na vjen referenca tek ai përmes *setter* metodës.

[3] Dhanji R. Prasanna – Dependency Injection, faqe 74
[4] Dhanji R. Prasanna – Dependency Injection, faqe 75

3. JSR 330

JSR (Java Specification Requests) janë përshkrime të specifikimeve të propozuara dhe të pranuara të platformës Java.[5]

Një prej këtyre propozimeve është JSR 330, specifikim ky i cili tenton ta standardizojë API për dependency injection.

JSR-330 specifikon 1 interface dhe 5 annotations për përdorim në fushën e DI:

Interfejsi quhet *Provider<T>* , kurse annotations e specifikuara janë: [12]

@Inject	Identifikon fushat, konstruktorët dhe metodat që duhet të injektohen.
@Qualifier	Annotation që përdoret për të specifikuar annotations tona (custom).
@Named	Një annotation përshkrues që ka si parametër një string.
@Scope	Identifikon annotations që përdoren për scoping.
@Singleton	Identifikon tipet që injektori duhet t'i inicializojë vetëm një herë.

Mbajmë në mend se këto janë përpjekje për standardizim, dhe jo çdo DI platformë i përkrahë për momentin.

3.1. Provider<T>

Siç e cekëm më lartë, JSR 330 e deklaron edhe një interface *Provider<T>* :

```
public interface Provider<T> {
    T get();
}
```

Këtë interfejs e përdorim ashtu që në vend se ta inject-im objektin e tipit T, e injektim *Provider<T>*. Kjo gjë na e mundëson :

1. Ti marrim disa instanca të *T* duke e thirrur *get()* disa herë.
2. *Lazy initialization* – objekti të krijohet vetëm kur na nevojitet për herë të parë, d.m.th kur ta thërrasim *get()* herën e parë.
3. Një zgjidhje të mundshme të problemit të referimit reciprok, pasi mund të injektojmë *Provider<T>* në vend të objektit T.

[5] https://jcp.org/en/jsr/overview

3.2. @Inject

Ky annotation përdoret për identifikim të gjërave që duhet injektuar, qofshin ato fusha, konstruktorë apo metoda.

JSR 330 specifikon këto gjëra lidhur me *@Inject* :

1. Fushat dhe metodat që duhet të injektohen mund të jenë statike apo pjesë e një objekti.
2. Pjesët e injektueshme mund të kenë cilindo nivel të qasjes (private, package-private, protected, public).

Për konstruktorë:

1. Sipas specifikimit, konstruktorët duhet të injektohen fillimisht, pastaj fushat e në fund metodat.
2. Konstruktorët e shënuar me @Inject mund të kenë çfarëdo numri të parametrave, me kusht që klasa të mos ketë më shumë se një konstruktor të shënuar me këtë annotation.
3. Shënimi me @Inject është opsional për konstruktorin pa argumente të Java nëse nuk kemi konstruktorë tjerë. D.m.th nuk kemi nevojë ta shkruajmë një konstruktor tonin vetëm që ta shënojmë me @Inject, kjo gjë është e nënkuptuar.

Për fusha:

1. Fushat e shënuara me @Inject nuk mund të jenë finale.

Për metoda:

1. Nuk mund të jenë abstrakte
2. Nuk duhet të deklarojnë ndonjë *"type parameter"* të tyrin.
3. Mund të kthejnë rezultat (edhe pse rezultati zakonisht injorohet prej platformave)
4. Mund të pranojë zero apo më shumë parametra.
5. Nëse një metodë e shënuar me *@Inject* e mbishkruan (ang. override) një metodë tjetër të shënuar me @Inject, injektimi kryhet vetëm një herë, jo dy.
6. Nëse një metodë jo e shënuar me *@Inject* e mbishkruan një të shënuar, atëherë metoda nuk do të injektohet.

Opsionale është përdorimi i *qualifiers* (custom annotations të deklaruara nga ne). Ato mund të shënojnë një fushë apo parametër për ta përshkruar më mire si p.sh.

```
public class Truck {
    @Inject  @Black  private Provider<Tire> seatProvider;

    @Inject  void load(@Bricks Cargo windshield) {  }
}
```

3.3. @Qualifier

@Qualifier përdoret për të shënuar annotations që do përdoren nga ne për të treguar një gjë më specifike sikur *@Black* ,*@Bricks* në shembullin më lartë. P.sh. kodi për *@Black* do ishte:

```
@Documented
@Retention(RUNTIME)
@Qualifier
public @interface Black {
}
```

3.4. @Named

@Named është thjesht një qualifier që pranon një string si parametër. P.sh. kemi mundur ta përdorim *@Named("black")* në vend se ta krijonim annotation tonë *@Black*.

```
public class Truck {
    @Inject  @Named("black")  private Provider<Tire> seatProvider;

    @Inject  void load(@Named("bricks")  Cargo windshield) {  }
}
```

Por përdorimi i custom qualifiers është më i mirë sepse ka më pak mundësi për gabim në shkrimin e stringut në fjalë.

3.5. @Scope

Nganjëherë kemi nevojë që në një context të caktuar, injektori të na kthejë objektin e njëjtë, e jo një instancë të re. Këtë e arrijmë duke shënuar një annotation me *@Scope* dhe duke e aplikuar atë annotation në objekt.

3.6. @Singleton

@Singleton është një scope që tregon se objekti duhet të ketë jetëgjatësi maksimale (sa aplikacioni) dhe rrjedhimisht krijohet vetëm një herë, e për kërkesa tjera kthehet objekti i njëjtë.

4. Dependency injection platformat

Për momentin ekzistojnë disa librari që janë të zhvilluara për dependency injection. Ndër to më të njohurat janë :

1. Spring
2. Guice
3. Pico Container
4. Dagger 1&2 etj.

Të lartcekurat të gjitha janë DI platforma për gjuhën Java, por ka edhe për gjuhë tjera si p.sh. për .NET :

1. NInject
2. Spring.Net
3. CastleWindsor
4. Structure Map
5. Autofac
6. Unity etj.

Më poshtë do flasim lidhur me disa prej tyre, kryesisht ato për gjuhën Java. Do tregojmë dallimet në mes tyre, se si kanë evoluar platformat ndër vite, cila është më e mirë e më e lehtë për ta përdorur dhe pse.

Gjithashtu do tregojmë se si shkon procesi i konfigurimit dhe inkorporimit të tyre në projektet tona të përditshme.

Po ashtu në fund do tregojmë se si do mund të kodojmë vet ne një mini-librari për dependency injection duke përdorur prapë gjuhën Java.

Për ta punuar librarinë do përdorim *reflection*, ashtu që në runtime kur të na duhet një instancë e një klase, t'i analizojmë annotations në atë klasë,t'i krijojmë dependencies të saj, dhe ta krijojmë një instancë të asaj klase.

Kjo mënyrë është e ngjashme me atë që e bëjnë libraritë e mësipërme, përveç Dagger 2 që të gjitha llogaritjet i bën në compile-time, dhe gjeneron kodin e duhur.

Kjo librari nuk tenton të jetë librari gjithëpërfshirëse që i mbulon të gjitha rastet ekstreme por vetëm ta japë një ide se si mund të implementohet *Dependency Injection* vetë.

4.1. Spring

Spring është një platformë (application framework) për Java, e zhvilluar fillimisht nga Rod Johnson. Versioni i parë ai e publikoi njëkohësisht me publikimin e librës së tij "*Expert One-on-One J2EE Design and Development*" në tetor të vitit 2002, kurse versioni i tanishëm është 4.2.1.

Duhet cekur se *Spring* edhe pse si pjesë kryesore e ka IoC (inversion of control – dependency injection) , nuk është e zhvilluar vetëm për të, por ka edhe module tjera si:

- Aspect-oriented programming framework
- Authentication and authorization
- Transaction management framework
- Model-view-controller framework
- Remote access framework etj.

Moduli për dependency injection i Spring na ofron një mënyrë të lehtë për të specifikuar objektet dhe varësitë e tyre, dhe më pas platforma duke përdorur reflection i krijon dhe lidh objektet sipas specifikimit.

Gjatë punës me Spring po ashtu na hasim shpesh në një fjalë të re *beans*, me të cilën njihen objektet që krijohen-menaxhohen nga Spring. Këto *beans* krijohen sipas konfigurimit që ne mund ta japim ose në formë të XML fajllave, apo të annotations nëpër klasa.

Në rastin se duam ta përdorim konfigurimin me XML, atëherë duhet ti dimë disa fjalë kyçe apo atribute që i ofron Spring për definim të *beans*.[5]

class	Definon klasën e cila duhet të përdoret për ta krijuar *bean*-in. Është *mandatory*, pra duhet të specifikohet gjithsesi.
name	Përdoret për ta identifikuar në mënyrë unike një bean brenda fajllave konfigurues.
scope	Përdoret për ta specifikuar se çfarë *scope* kanë objektet e krijuara nga ky bean definition.
constructor-arg	Përdoret për t'i specifikuar argumentet (parametrat) e konstruktorit që do përdoret për inicializim të bean nëse duam të përdorim *constructor injection*.
property	Përdoret kur përdorim *setter injection.*
autowiring	E përdorim nëse duam që të mos specifikojmë constructor-arg , por që Spring të mundohet t'i lidhë vet argumentet me klasat përkatëse.
lazy-initialization	Specifikon se krijimi i instancës së bean duhet të bëhet kur të kërkohet për herë të parë, jo në startup.
initialization-method	Përdoret për ta specifikuar ndonjë metodë që duam që të thirret pasi të jetë krijuar si duhet instanca nga IOC kontejneri.
destruction-method	Përdoret për ta specifikuar ndonjë metodë që duam që të thirret kur të shkatërrohet kontejneri që përmban këtë bean

Një shembull elementar se si shkruhet XML fajlli konfigurues për *bean* është paraqitur më poshtë:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <!-- Më poshtë është paraqitur një definicion i thjeshtë i bean -->
   <bean id="foo" class="x.y.DroidFight" init-method="startFight">
       <constructor-arg ref= "C3PO" />
       <constructor-arg ref= "R2D2" />
       <constructor-arg ref= "BB8" />
   </bean>

   <!-- Një bean lazy, pra inicializimi bëhet pas thirrjes së parë -->
   <bean id="C3PO" class="x.y.C3PO" lazy-init="true"/>

   <!-- Bean me metodë inicializuese-->
   <bean id="R2D2" class="x.y.R2D2" init-method="startBeeping"/>

   <!-- Bean me metodë finalizuese -->
   <bean id="BB8" class="BB8" destroy-method="stopRolling"/>
</beans>
```

Ky XML i paraqitur është një bean definition për klasat e paraqitura më poshtë:

```
public class R2D2 {

    public void startBeeping() throws Exception {
        //init method
    }

}
```

```
public class C3PO {

    public C3PO() {
        System.out.print("Hello, I am C3PO, human cyborg relations");
    }
}
```

```
public class BB8 {

    public void stopRolling() throws Exception {
        //destory-method
    }
}
```

```
public class DroidFight {
    private R2D2 r2D2;
    private BB8 bb8;
    private C3PO c3PO;

    public DroidFight(R2D2 r2D2, BB8 bb8, C3PO c3PO) {
        this.r2D2 = r2D2;
        this.bb8 = bb8;
        this.c3PO = c3PO;
    }

    public void startFight() {
        //fight logic goes here
        System.out.print("BB8 wins");
    }

    public void setR2D2Method(R2D2 r2D2) {
        this.r2D2 = r2D2;
    }

    public void setBb8Method(BB8 bb8) {
        this.bb8 = bb8;
    }

    public void setC3POMethod(C3PO c3PO) {
        this.c3PO = c3PO;
    }
}
```

XML i mësipërm e konfiguronte platformën që të përdor *constructor injection*. Për *setter injection*, konfigurimi i *DroidFight* do dukej kështu: (pjesa tjetër mbetet njëjtë)

```
<bean id="foo" class="x.y.DroidFight" init-method="startFight">

       <property name= "C3POMethod"  ref= "C3PO" />

       <property name= "R2D2Method"  ref= "R2D2" />

       <property name= "BB8Method"   ref= "R2D2" />

   </bean>
```

Siç thamë më herët, Spring përkrah edhe konfigurimin përmes annotations:

1. @Autowired – e ngjashme me @Inject që specifikon JSR 330
2. @Qualifier – e njëjtë me @Qualifier të JSR 330
3. @Required – përdoret tek setter metodat, për të treguar se domosdo duhet të inicializohet, përndryshe hedh *BeanInitializationException.*
4. @PostConstruct – si init-method te konfigurimi me XML
5. @ PreDestroy – si destroy-method te konfigurimi me XML

Spring po ashtu përkrah scoping:

1. Singleton – një instancë e vetme krijohet për kontejner
2. Prototype – në çdo kërkesë krijohet instancë e re

Tri scopes tjera shërbejnë vetëm për ueb-aplikacione:

3. Request – një instancë e njëjtë për HTTP request
4. Session – një intancë e njëjtë brenda HTTP sesionit
5. Global-session – për një instancë që duhet të shpërndahet brenda sesioneve të ndryshme.

Përveç scopes të mësipërme, mund të krijojmë edhe tjera sipas nevojës.

Spring është një platformë shumë e përdorur edhe në ditët e sotme, por prapë i ka disa mangësi në krahasim me Guice apo Dagger.

1. Konfigurimi përmes XML është shumë i gjate, d.m.th. duhet fjalë shumë për të specifikuar një gjë. Në versionet e fundit mund t'i përdorim edhe annotations, po në fillim konfigurimi përmes XML ishte mënyra e vetme.
2. Validimi i konfigurimit bëhet në runtime, d.m.th. nëse bëjmë ndonjë gabim në ndonjë emër në ndonjërën prej bean fajllave, këtë gabim e vërejmë vetëm pasi ta startojmë aplikacionin.
3. Po ashtu edhe validimi i grafit bëhet në runtime, d.m.th. nëse kemi harruar ta specifikojmë ndonjë konstruktor për ndonjë parametër, prapë gabimin e vërejmë pas startimit.

4.2. Guice

Guice (lexohet si fjala angleze “juice") është një dependency injection platformë/librari e krijuar nga kompania Google, dhe deri në krijimin e Dagger 2 përdorej në shumicën e aplikacioneve të punuara nga kjo kompani.

Guice në krahasim me Spring është librari shumë më e vogël, më lightweight, pasi nuk ka shumë module si Spring, por fokusohet vetëm në *dependency injection*. Po ashtu Guice konfigurimin e bën vetëm përmes Java objekteve (moduleve, annotations), jo edhe me XML si Spring. Për më tepër Guice ishte libraria e parë për *dependency injection* për shfrytëzimin e risive në Java 5 si generics dhe annotations për të lehtësuar konfigurimin dhe për të zvogëluar mundësitë e gabimeve.

Në Guice, konfigurimi i dependencies bëhet përmes *moduleve*, saktësisht përmes klasave që e zgjerojnë *AbstractModule*. Metodë me rëndësi në këtë klasë është *configure( )*, në të cilën ne duhet ti specifikojmë lidhjet. Një shembull i thjeshtë që paraqet një konfigurim të mundshëm është paraqitur më poshtë:

```
public class BillingModule extends AbstractModule {
    @Override
    protected void configure() {

        bind(TransactionLog.class)
                .annotatedWith(DatabaseLog.class)
                .to(DatabaseTransactionLog.class);

        bind(CreditCardProcessor.class)
                .to(PaypalCreditCardProcessor.class)
                .in(Singleton.class);
    }

}
```

Kodi i mësipërm i tregon Guice që nëse duhet të injektohet në ndonjë klasë ndonjë objekt i *TransactionLog*, si dhe ajo varësi (parametri në konstruktor, metodë apo variabla e fushës) është e shënuar me annotation *@DatabaseLog*, atëherë Guice duhet ta krijojë një objekt të klasës *DatabaseTransactionLog* dhe ta injektojë atë.

Linja tjetër i tregon se nëse kërkohet *CreditCardProcessor*, atëherë Guice ta krijojë objektin nga klasa *PaypalCreditCardProcessor* dhe ta injektojë. Linja e fundit në atë shprehje (metoda *in*) e specifikon scope, që në këtë shembull është singleton.

Në Guice për të specifikuar se çka duhet të injektohet brenda klasave (cili konstruktor apo cilat metoda) përdorim annotation *@Inject* sipas specifikacionit JSR 330. Që nga versioni 3.0, Guice i bindet specifikacionit JSR 330, po i ka edhe annotations dhe interfejsat e vet në paketën *com.google.inject*. Në disa raste është njëjtë cilindo version ta përdorim, tek disa ndryshon sjellja, siç tregohet në tabelën më poshtë:[6]

JSR-330 **javax.inject**	***Guice*** **com.google.inject**	
@Inject	@Inject	Ndryshojnë në disa aspekte, si p.sh. @Inject i Guice ka edhe një parametër *optional*.
@Named	@Named	Njëjtë cilëndo ta përdorim
@Qualifier	@BindingAnnotation	Njëjtë cilëndo ta përdorim
@Scope	@ScopeAnnotation	Njëjtë cilëndo ta përdorim
@Singleton	@Singleton	Njëjtë cilëndo ta përdorim
Provider	Provider	Ndryshojnë, por mund ta konvertojmë atë të JSR-330 në versionin e Google përmes: *Providers.guicify(Provider<T> provider)*

Që prej Guice 2.0, përveç metodës së cekur më lartë të konfigurimit, është shtuar edhe një mënyrë me @Provides metoda, si më poshtë :

```
public class BillingModule extends AbstractModule {
    @Override
    protected void configure() {

        bind(CreditCardProcessor.class)
                .to(PaypalCreditCardProcessor.class)
                .in(Singleton.class);
    }

}

@Provides @DatabaseLog
public TransactionLog getDatabaseTransactionLog(){
    TransactionLog log =  new DatabaseTransactionLog();
    //do something with log if necessary
    return log;
}
```

[6] https://github.com/google/guice/wiki/JSR330

Kjo në këtë rast është ekuivalente me shembullin e mësipërm, sa i përket *TransactionLog*. Një gjë që ia vlen të ceket është se metoda *getDatabaseTransactionLog( )* lejohet të ketë parametra, e ato parametra do injektohen nga Guice.

Guice kur u krijua i zgjodhi disa probleme që i kishte Spring si konfigurimi më i lehtë, i tërë konfigurimi në Java, po prapë disa probleme mbetën:

1. Validimi i grafit edhe në Guice bëhet në runtime, pra gabimet në konfigurim i gjejmë vetëm pasi ta startojmë aplikacionin.
2. Pasi klasat që e lidhin konfigurimin me source kodin tonë gjenerohen në runtime përmes reflection, atëherë nëse duam që gjatë programimit ta dimë se çfarë lloj varësive na e dërgon platforma, këtë nuk mund ta bëjmë thjesht duke shikuar se cila klasë po na thërret. E vetmja mënyrë është nëse shkojmë dhe i lexojmë fajllat konfigurues (modulet në këtë rast), dhe ta gjejmë *binding* përkatës.

4.3. Dagger 2

Edhe Dagger 2, sikur Guice është krijuar nga Google, por ky është versioni i dytë i librarisë Dagger, pasi versioni i parë është krijuar nga Square.

Plusi më i madh i Dagger 2 në krahasim me libraritë e lartcekura është se nuk përdor reflection fare, por çdo gjë e bën në compile time me gjenerim të kodit, e kjo e bën Dagger 2 shumë më të shpejtë se libraritë tjera. Si rrjedhim i kësaj edhe gabimet që i bëjmë në konfigurim i detektojmë gjatë kompajllimit. Po ashtu pasi klasat lidhëse nuk po krijohen në runtime, por gjenerohen gjatë kompajllimit, atëherë edhe duke përdorur veglat e IDE-ve të ndryshme mund të gjejmë se cila klasë po na i injekton varësitë dhe cilat varësi po na i injekton. Dagger i përmbahet komplet specifikimit JSR 330 kështu që të gjitha ato annotation dhe interfejsi mund të përdoren.[7]

Konfigurimi në Dagger bëhet përmes moduleve dhe komponentëve. Modulet janë Java klasa të shënuara me annotation *@Module*, kurse komponentet janë interfejsa të shënuara me *@Component.*

Në module shkruhet konfigurimi se si lidhen gjërat, dhe këtë e bëjmë me anë të *@Provides* metodave, kurse përmes komponentëve (implementimit të tyre) e bëjmë injektimin.

Marrim një shembull, dhe përmes tij e shpjegojmë se si punon Dagger2. Ky shembull mund të gjendet në repositorin e Dagger 2, por për qartësi kodi është shkruar edhe në faqen vijuese:[7]

[7] https://github.com/google/dagger/tree/master/examples/simple/src/main/java/coffee

```java
interface Heater {
    void on();
    void off();
    boolean isHot();
}
```

```java
interface Pump {

  void pump();

}
```

```java
class CoffeeMaker {
    private final Lazy<Heater> heater; // Create heater only when we use it.
    private final Pump pump;

    @Inject
    CoffeeMaker(Lazy<Heater> heater, Pump pump) {
        this.heater = heater;
        this.pump = pump;
    }

    public void brew() {
        heater.get().on();
        pump.pump();
        System.out.println(" [_]P coffee! [_]P ");
        heater.get().off();
    }
}
```

```java
class ElectricHeater implements Heater {
    boolean heating;

    @Override
    public void on() {
        System.out.println("~ ~ ~ heating ~ ~ ~");
        this.heating = true;
    }

    @Override
    public void off() {
        this.heating = false;
    }

    @Override
    public boolean isHot() {
        return heating;
    }
}
```

```java
class Thermosiphon implements Pump {
  private final Heater heater;

  @Inject
  Thermosiphon(Heater heater) {
    this.heater = heater;
  }

  @Override public void pump() {
    if (heater.isHot()) {
      System.out.println("=> => pumping => =>");
    }
  }
}
```

Siç shihet më lartë me *@Inject* e shënojmë konstruktorin që duam të përdoret nga Dagger për të krijuar objekt nga klasa.

Që Dagger ta dijë se cilat implementime të *Heater* dhe *Pump* ti përdorë kur të krijojë objekt prej klasës *CoffeeMaker*, e specifikojmë në module :

```
@Module
class DripCoffeeModule {

  @Provides
  Heater provideHeater() {
    return new ElectricHeater();
  }

  @Provides Pump providePump(Thermosiphon pump) {
    return pump;
  }
}
```

Kjo i thotë Dagger që kur kërkohet *Heater*, të kthejë *ElectricHeater*, kurse kur kërkohet *Pump* të kthejë instancë të *Thermosiphon*. Por pse askund nuk e cekëm në modul se si krijohet objekti nga *Thermisiphon* ? Në Dagger2 nuk është e nevojshme që të ceket nëse klasa ka konstruktor të shënuar me *@Inject* – ai fakt i mjafton Dagger që të dijë si ta krijojë *Thermosiphon*.

Por për ta krijuar injektorin, nuk mjafton vetëm moduli, duhet krijuar edhe një *Component:*

```
@Component(modules = DripCoffeeModule.class)
interface CoffeeShop {

    CoffeeMaker maker();

    void inject(SomeObject someObject);

    OtherComponent getOtherComponent(OtherModule otherModule);
}
```

Vërejmë se *CoffeeShop* është një interface, ne vetëm specifikojmë se çka duam të krijojmë, kurse Dagger e shkruan vetë implementimin. Klasa e gjeneruar nga Dagger do jetë e emërtuar në formatin *Dagger<ComponentName>*, në këtë rast *DaggerCoffeeShop* dhe për ta krijuar instancën e *CoffeeShop*, duhet t'ia dërgojmë një komponentin në fjalë një instancë të modulit të deklaruar sipër interfejsit brenda annotation-it *@Component*, si më poshtë:

```
CoffeeShop coffeeShop = DaggerCoffeeShop.builder()
        .dripCoffeeModule(new DripCoffeeModule())
        .build();
```

Vlen të ceket që pasimi i një instance të modulit tek builder i komponentës në shembullin më lartë është opsional, pasi konstruktori i modulit nuk ka parametra. Nëse konstruktori i modulit do kishte parametra, atëherë do duhej gjithsesi ta pasonim.

Komponentën e krijuar mund ta përdorim në disa mënyra:

```
//1
CoffeeMaker maker = coffeeShop.maker();
maker.brew();

//2
SomeObject someObject = new SomeObject();
coffeeShop.inject(someObject);

//3
OtherComponent otherComponent = coffeeShop.getOtherComponent(new OtherModule());
```

Një përdorim i mundshëm është që ta marrim një instancë të gatshme të *CoffeMaker*, të krijuar komplete me gjithë varësitë. Deklarimi i një metode si *maker( )* në komponentë, që na kthen një objekt të krijuar të grafit quhet ekspozim i grafit. Përdorimi i dytë është shkruarja e një metode si *inject( )*, që merr objekt dhe nuk kthen asgjë (void). Me këtë i injektojmë fushat e shënuara me *@Inject* të objektit që është si parametër (në këtë rast *SomeObject*). Kurse përdorimi i tretë është krijimi i një komponente tjetër që varet nga komponenta ekzistuese, por plus edhe me një modul me të dhëna specifike për komponentën e re (si te rasti 3 tek kodi).

Siç shihet edhe nga shembulli, edhe Dagger përkrah *Provider<T>* dhe *Lazy<T>* që mundësojnë:

- Ta marrim një instancë të re në çdo thirrje të *get( )* – në rastin e *Provider<T>*
- Shtyrja e inicializimit deri në thirrjen e parë të *get( )* – në rastin e *Lazy<T>*

Në Dagger sa i përket scoping, të gatshme e kemi vetëm annotation *@Singleton*, por mund të shkruajmë scoping annotations vetë. Kur i kemi dy komponenta dhe njëra është dependent në tjetrën (apo është subcomponent), atëherë gjithsesi secila duhet të këtë scope të veten të deklaruar (jo të njëjtën scope). Më pas nëse në modulin e përdorur nga komponenta e re duam te deklarojmë diçka si singleton, atëherë në vend të *@Singleton*, e përdorim scope-in e deklaruar tek komponenta. Kjo i thotë Dagger që ta krijojë vetëm një instancë të objektit brenda komponentës – d.m.th. objekti është si singleton lokal për komponentë, dhe mbetet ashtu brenda jetëgjatësisë së komponentës.[8]

Dagger 2 është me kod të hapur[8], e licencuar nën *Apache 2.0 License*.[9]

[8] https://github.com/google/dagger
[9] http://www.apache.org/licenses/LICENSE-2.0

4.4. GallifreyDI

Më poshtë kam tentuar që duke përdorur gjuhën Java ta shkruaj një implementim të thjeshtë të modelit të *dependency injection*. Ky implementim do jetë minimalistik, pa të gjitha features (sq. veçoritë) e librarive të mësipërme, duke u munduar ta shpreh vetëm thelbin e modelit *dependency injection* : objekti të mos merret me krijimin e varësive të tij, ato t'i vijnë të gatshme për përdorim.

Libraria quhet *GallifreyDI,* dhe përmban 3 klasa : *ClassScanner, Repository, Injector*.

Për ta përdorur, fillimisht duhet të inicializojmë librarinë përmes:

```
Injector.init(PACKAGE_NAME);
```

Pas thirrjes së kësaj metode, libraria e skanon atë paketë, i gjen klasat dhe i analizon.

Libraria bën injektimin e variablave të fushës, duke përdorur *setter injection.* Pra që një fushë të injektohet, atë duhet ta shënojmë me *@Inject,* dhe ta shkruajmë një setter për atë fushë.

Nëse fusha është interfejs, apo metodë abstrakte, duhet të specifikojmë se cilin implementim (përkatësisht klasë konkrete) duam ta injektojmë. Këtë e bëjmë duke e shënuar fushën edhe me annotation *@Named("someAlias")* dhe kjo bën që të injektohet klasa e cila është shënuar me atë annotation.

Marrjen e një objekti të krijuar dhe të pajisur me varësi e bëjmë me anë të metodave *getEntity( ),* apo nëse e kemi të krijuar dhe vetëm duam t'ia injektojmë varësitë me anë të metodës *inject( )* :

```
//marrim instacë të coffeeMaker
CoffeeMaker firstCoffeMaker = Injector.getEntity(CoffeeMaker.class);

//marrim një lloj specifik të coffeeMaker
CoffeeMaker nescafeMaker = Injector.getEntity("nescafe", CoffeeMaker.class);

// nëse e kemi të krijuar, vetëm e injektim
CoffeeMaker espressoMaker = new EspressoMaker();
Injector.inject(espressoMaker);
```

Kodi i librarisë është paraqitur në faqet në vijim:

```
public class ClassScanner {

    Map<String, Class> namedEntities;
    Map<Class, Object> singletons;

    Set<String> foundAnnotatedClassNames;


    @SuppressWarnings("unchecked")
    public Repository scan(String packageName) {
        foundAnnotatedClassNames = new HashSet<String>();
        namedEntities = new HashMap<String, Class>();
        singletons = new HashMap<Class, Object>();
        try {
            ClassLoader classLoader =
Thread.currentThread().getContextClassLoader();
            Enumeration<URL> resources =
classLoader.getResources(packageName.replace(".", "/") + "/");
            while (resources.hasMoreElements()) {
                File folder = new File(resources.nextElement().getFile());
                scanFolderForClasses(folder, packageName);
            }
            return Repository.getInstance(namedEntities, singletons);

        } catch (Exception e) {
            throw new RuntimeException(e);
        }

    }



    private void scanFolderForClasses(File folder, String packageName) throws
ClassNotFoundException {

        File[] folderContent = folder.listFiles();
        assert folderContent != null;
        for (File file : folderContent) {
            String fileName = file.getName();
            if (fileName.endsWith(".class")) {
                // shto klasen ne listë
                String filenameWithoutClass = fileName.substring(0,
fileName.lastIndexOf('.'));
                String completeClassName = packageName + "." +
filenameWithoutClass;
                boolean added = foundAnnotatedClassNames.add(completeClassName);
                if (added) {
                    processClass(Class.forName(completeClassName));
                }
            } else if (file.isDirectory() && fileName.indexOf('.') == -1) {
                //thirrje rekurzive qe ta analizojme edhe nënfolderin
                scanFolderForClasses(file, packageName + "." + fileName);
            }
        }
    }
```

```
    private void processClass(Class clazz) {

        if (clazz.isAnnotationPresent(Named.class) ||
            clazz.isAnnotationPresent(Singleton.class)) {
            Named named = (Named) clazz.getAnnotation(Named.class);
            String name = named != null ? named.value() : null;
            if (name == null || name.length() == 0) {
                name = clazz.getSimpleName();
            }
            if (namedEntities.get(name) != null) {
                throw new RuntimeException(
                 String.format(
                 "There are two classes with the same @Named(\"%s\") : %s and %s",
                  name,namedEntities.get(name).getCanonicalName(),
                  clazz.getCanonicalName()));
            }
            namedEntities.put(name, clazz);
            System.out.format("%s class has been registered with alias \"%s\"\n",
                 clazz.getCanonicalName(), name);

            // register singletons:
            if (clazz.isAnnotationPresent(Singleton.class)) {
                singletons.put(clazz, null);
                System.out.format("%s class (alias \"%s\") is singleton\n",
                clazz.getCanonicalName(), name);
            }

        }
    }

}
```

```
public class Repository {

    private static Repository repositoryInstance;

    private Map<String, Class> namedEntities;     //all classes that are found
    private Map<Class, Object> singletons;    //all classes annotated with
@Singleton
    private Set<Class> objectsCurrentlyInitializing = new HashSet<Class>(); // se
we can check for circular dependencies

    private Repository(Map<String, Class> namedEntities, Map<Class, Object>
singletons) {
        this.namedEntities = namedEntities;
        this.singletons = singletons;
    }

    public static synchronized Repository getInstance(Map<String, Class>
namedEntities, Map<Class, Object> singletons) {
        if (repositoryInstance == null) {
            repositoryInstance = new Repository(namedEntities, singletons);
        }
        return repositoryInstance;
    }
```

```
    public Object getEntity(String alias) throws InstantiationException,
InvocationTargetException, IllegalAccessException {
        Class clazz = getClass(alias);
        return getEntity(clazz);
    }

    public Class getClass(String alias) throws InstantiationException {
        Class clazz = namedEntities.get(alias);
        if (clazz == null) throw new InstantiationException(
         String.format(
         "There exists no class with annotation @Named(\"%s\")", alias));
        return clazz;
    }

    @SuppressWarnings("unchecked")
    public <T> T getEntity(Class clazz) throws IllegalAccessException,
InvocationTargetException, InstantiationException {
        T entity;

        if (singletons.containsKey(clazz)) {
            Object singletonObject = singletons.get(clazz);
            if (singletonObject == null) {
                singletonObject = instantiateObject(clazz);
                singletons.put(clazz, singletonObject);
            }
            entity = (T) singletonObject;
        } else {
            entity = (T) instantiateObject(clazz);
        }

        return entity;
    }

    @SuppressWarnings("unchecked")
    private <T> T instantiateObject(Class<T> clazz) throws IllegalAccessException,
InstantiationException, InvocationTargetException {

        T newInstance = clazz.newInstance();

        if (objectsCurrentlyInitializing.contains(clazz)) {
            throw new InstantiationException(
                    String.format(
       "Circular reference has been detected during the instantiation of class %s",
         clazz.getCanonicalName()));
        }
        try {
            objectsCurrentlyInitializing.add(clazz);
            Injector.inject(newInstance);
            // i injektim edhe varesite e klases se krijuar
        } finally {
            objectsCurrentlyInitializing.remove(clazz);
        }
        return newInstance;
    }

}
```

```
public class Injector {

    private static Repository repository;

    public static void init(String packageName) {
        ClassScanner classScanner = new ClassScanner();
        repository = classScanner.scan(packageName);
    }

    @SuppressWarnings("unchecked")
    public static <T> T getEntity(Class<T> clazz) {
        try {
            Object entity = repository.getEntity(clazz);
            return (T) entity;
        } catch (Exception e) {
            throw new RuntimeException(
         String.format("Could not instantiate class %s", clazz.getName()), e);
        }
    }


    @SuppressWarnings("unchecked")
    public static <T> T getEntity(String alias, Class<T> clazz) {
        try {
            Object entity = repository.getEntity(alias);
            return (T) entity;
        } catch (Exception e) {
            throw new RuntimeException(
         String.format("Could not instantiate class %s", clazz.getName()), e);
        }
    }

    /**
     * Ia injekton ketij objekti varesite e tij
     */

    public static <T> void inject(T object) {
        Class<?> clazz = object.getClass();

        try {
            for (Field field : getInjectedFields(clazz)) {

                String fieldName = field.getName();

                //psh per field coffeeMaker, emri i metodes del setCoffeeMaker
                String setterMethodName = "set" +
                fieldName.substring(0, 1).toUpperCase() + fieldName.substring(1);
                Method setterMethod;
                try {
                    setterMethod =
                        clazz.getMethod(setterMethodName,field.getType());
                } catch (Exception e) {
                    throw new IllegalAccessException(
                 String.format("Cannot find (public) setter %s.%s(%s})",
                 clazz.getCanonicalName(), setterMethodName,
                 field.getType().getCanonicalName()));
                }

                Class<?> fieldType = field.getType();
                String classThatWeAreWorkingOn = object.getClass().getName();
```

```
                Named named = field.getAnnotation(Named.class);
                Object objectToBeInjected;
                if (fieldType.isInterface()) {
//shikojme se a eshte interface kjo, nese po a e implementon klasa me @Named kete
//interface
                    if (named != null && named.value() != null) {
                        checkNamedObjectImplementsInterface(
                                named.value(), field, classThatWeAreWorkingOn);
                        objectToBeInjected = repository.getEntity(named.value());
                    } else {
                        throw new InstantiationException(String.format(
                          "Class %s. Variable \'%s\'s type is of interface, " +
                          "and must be annotated with @Named(\"alias\")",
                                classThatWeAreWorkingOn,
                                field.getName()));
                    }

                } else if (Modifier.isAbstract(fieldType.getModifiers())) {
//kemi te bejme me klase abstrakte,nese s'ka annotation -> error, nese klasa me qat
//annotation nuk e extend -> error
                    if (named != null && named.value() != null) {
                        checkNamedObjectExtendsClass(
                                named.value(), field, classThatWeAreWorkingOn);
                        objectToBeInjected = repository.getEntity(named.value());
                    } else {
                        throw new InstantiationException(String.format(
                        "Class %s. Variable \'%s\'s type is of abstract class, " +
                        "and must be annotated with @Named(\"alias\")",
                                classThatWeAreWorkingOn,
                                field.getName()));
                    }
                } else {
                    objectToBeInjected = repository.getEntity(fieldType);
                }


                try {
                    setterMethod.invoke(object, objectToBeInjected);
                } catch (Exception e) {
                    throw new InstantiationException(String.format(
      "Unable to call setter function [%s] of object {%s} with argument type {%s}",
                            setterMethod.toString(),
                            object.getClass(),
                            objectToBeInjected.getClass()));
                }

                System.out.format(
                  "Injected %s.%s() successfully\n", clazz.getCanonicalName(),
                  setterMethodName);
            }
        } catch (Exception e) {
            throw new RuntimeException(
            String.format("Could not instantiate class %s", clazz.getName()), e);

        }
    }
```

```
    private static void checkNamedObjectExtendsClass(String value, Field field,
String classThatWeAreWorkingOn) throws InstantiationException {
        Class requiredSuperclass = field.getType();


        Class currentSuperClass = repository.getClass(value);
        boolean found = false;
        while (currentSuperClass != Object.class) {
            currentSuperClass = currentSuperClass.getSuperclass();
            if (currentSuperClass == requiredSuperclass) {
                found = true;
                break;
            }
        }
        if (!found) {
            throw new InstantiationException(String.format(
"Could not inject field %s in %s, because the class %s "+
"with annotation @Named(\"%s\") extend doesn't extend the abstract class %s",
                    field.getName(), classThatWeAreWorkingOn, requiredSuperclass,
value, field.getType()
            ));
        }
    }



    private static void checkNamedObjectImplementsInterface(String value, Field
field, String classThatWeAreWorkingOn) throws InstantiationException {

        boolean namedObjectImplementsInterface = false;
        Class classWithAnnotation = repository.getClass(value);
        Class<?>[] interfaces = classWithAnnotation.getInterfaces();
        for (Class<?> anInterface : interfaces) {
            if (anInterface == field.getType()) {
                namedObjectImplementsInterface = true;
                break;
            }
        }
        if (!namedObjectImplementsInterface) {
            throw new InstantiationException(String.format(
"Could not inject field %s in %s, because the class %s" +
"with annotation @Named(\"%s\") annotation doesn't implement the interface",
       field.getName(), classThatWeAreWorkingOn, classWithAnnotation, value
            ));
        }
    }
```

```
 /**
     * Returns all @Inject annotated fields of the class and parent classes.
     */
    private static Collection<Field> getInjectedFields(Class clazz) {
        List<Field> fieldList = new ArrayList<Field>();

        for (Field field : clazz.getDeclaredFields()) {
            if (field.isAnnotationPresent(Inject.class)) {
                fieldList.add(field);
            }
        }

        Class superClass = clazz.getSuperclass();
        if (superClass != null) {
            Collection<Field> fieldsOfParent = getInjectedFields(superClass);
            fieldList.addAll(fieldsOfParent);
        }
        return fieldList;
    }
}
```

Lista e figurave

Bibliografia

[1] Prasanna, Dhanji R. *Dependency Injection.* 1st ed. Manning publications,2009 (https://www.manning.com/books/dependency-injection)

[2] Seemann, Mark. *Dependency Injection in .NET.* 1st ed. Manning publications,2011 (https://www.manning.com/books/dependency-injection-in-dot-net)

[3] Fowler, Martin. *Inversion of Control Containers and the Dependency Injection pattern* (http://www.martinfowler.com/articles/injection.html)

[4] Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra. *Head First Design Patterns.* 1st ed. O'Reilly Media,2004 (http://shop.oreilly.com/product/9780596007126.do)

[5] The IoC container, Spring documentation (http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html)

[6] Guice github wiki (https://github.com/google/guice/wiki)

[7] Dagger 2 documentation (http://google.github.io/dagger/)

[8] Dependency injection with Dagger 2 - Custom scopes (http://frogermcs.github.io/dependency-injection-with-dagger-2-custom-scopes/)

[9] Pico container documentation (http://picocontainer.com/introduction.html)

[10] Dagger 2 - Google I/O presentation slides (https://docs.google.com/presentation/d/1fby5VeGU9CN8zjw4lAb2QPPsKRxx6mSwCe9q7ECNSJQ/pub?start=false&loop=false&delayms=3000&slide=id.p)

[11] Dependency Injection with Dagger 2 (Devoxx 2014) – Jake Wharton talk (https://speakerdeck.com/jakewharton/dependency-injection-with-dagger-2-devoxx-2014)

[12] JSR 330 specification documentation (https://jcp.org/en/jsr/detail?id=330)