ERHVERVSAKADEMI AARHUS

Programmering af forbindelser mellem klasser

Indholdsfortegnelse

Implementering af associering, aggregering og komposition

1. Forbindelser mellem klasser

En væsentlig del af et designklassediagram er klasserne og deres indbyrdes forbindelser. Forbindelserne har multipliciteter i begge ender og med pile angives retning på forbindelsen (en forbindelse kan være enkeltrettet eller dobbeltrettet). En forbindelse er enten en associering, aggregering eller en komposition.

I denne note beskrives, hvordan de tre forbindelsestyper associering, aggregering og komposition kan programmeres.

En forbindelse fra klassen A til klassen B indikerer, at et objekt af klassen A har kendskab til et eller flere objekter af klassen B (afhængig af mutipliciteten). Klassen A har derfor en attribut, der kan referere til en eller flere objekter af klassen B. Disse attributter kaldes linkattributter (eller linkfelter eller blot links).

Forbindelsen mellem objekter er dynamisk og kan variere over tid. Det betyder, at antallet af objekter og præcis hvilke objekter af type B, et objekt af type A har kendskab til, kan variere i løbet af objektets levetid. Klassen A skal derfor have metoder til at tilføje og fjerne de objekter af typen B, som klassen A har kendskab til.

I klasserne tilføjes linkattributter samt metoder til at vedligeholde disse forbindelser. Præcis hvordan de implementeres afhænger af multipliciteten af forbindelsen, typen af forbindelsen, og hvorvidt den er enkelt eller dobbeltrettet.

I det følgende beskrives hvordan de forskellige typer af forbindelser implementeres afhængig af retning og multiplicitet. Først beskrives de enkeltrettede forbindelser (associering, aggregering og komposition) og dernæst de dobbeltrettede forbindelser.

2. Enkeltrettet til-mange associering

Regel: I klassen hvor associeringen starter, tilføjes en link attribut, hvor typen er et objekt, der kan indeholde flere objekter (typisk en ArrayList).

Associeringen er implementeret med en linkattribut i klassen *Group*. Multiplicitetерne fortæller, at en gruppe kan have 0 til mange personer tilknyttet, og en person kan være med i højst én gruppe.

En linkattribut med navnet *persons* af typen ArrayList<Person> tilføjes til klassen *Group*. Der skal selvfølgelig også være metoder, som arbejder med linkattributten. Ofte er der brug for en get metode, som returnerer linkattributten, og en add metode, som tilføjer et objekt til linkattributten. Somme tider er det også nødvendigt med en remove metode, som fjerner et objekt fra linkattributten. I kodeeksemplet herunder er der tilføjet tre metoder: getPersons(), addPerson() og removePerson().

```
public class Group {
    // association --> 0..* Person
    private final ArrayList<Person> persons = new ArrayList<>();

    public ArrayList<Person> getPersons() {
        return new ArrayList<>(persons);
    }

    /**
     * Adds the person to this group,
     * if they aren't connected.
     * Pre: The person isn't connected to another group.
     */
    public void addPerson(Person person) {
        if (!persons.contains(person)) {
            persons.add(person);
        }
    }

    /**
     * Removes the person from this group,
     * if they are connected.
     */
    public void removePerson(Person person) {
        if (persons.contains(person)) {
            persons.remove(person);
        }
    }
}
```

Bemærk, at metoden *getPersons()* returnerer en kopi af link attributten *persons*. Dermed sikres, at kun metoder i klassen Group kan opdatere linkattributten.

Metoden addPerson() sikrer ved hjælp af if-sætningen, at forbindelsen kun sættes, hvis den ikke allerede er sat. Hvis forbindelsen allerede eksisterer, så gør metoden intet. Pre-betingelsen på addPerson() sikrer, at en person er med i højst én gruppe. Pre-betingelsen tjækkes ikke, da det enkelte group objekt ikke har kendskab til alle grupperne i systemet.
Metoden removePerson() sikrer ved hjælp af if-sætningen, at forbindelsen kun brydes, hvis den allerede eksisterer. Hvis forbindelsen ikke eksisterer allerede, så gør metoden intet.

Koden for hele eksemplet kan ses i pakken *u_group01_0mperson* i den udleverede kode.

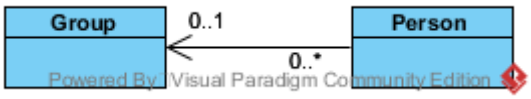
Antag nu, at det det er tilladt for en person at være med i *flere* grupper, som det er angivet i nedenstående figur.

Dette vil give anledning til præcis den samme kode som i ovenstående eksempel. Eneste forskel vil være, at pre-betingelsen på metoden addPerson() ikke skal med.

3. Enkeltrettet til-0..1 associering

Regel: I klassen hvor associeringen starter, tilføjes en simpel link attribut.

Eksempel:



Associeringen er implementeret med en linkattribut i klassen *Person*. Multipliciteterne fortæller, at en person kan være med i højst én gruppe, og at en gruppe kan have 0 til mange personer tilknyttet.

En link attribut med navn *group* tilføjes til klassen *Person*. Der vil ofte være brug for en get metode, som returnerer linkattributten, og en set metode, som opdaterer linkattributten. I kodeeksemplet herunder er der tilføjet 2 metoder*: getGroup()* og *setGroup()*.

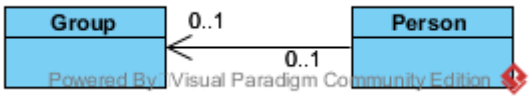
```
public class Person {
    private String name;
    // association --> 0..1 Group
    private Group group;

    public Group getGroup() {
        return group;
    }

    /**
     * Sets the group as this person's group.
     */
    public void setGroup(Group group) {
        if (this.group != group) {
            this.group = group;
        }
    }
}
```

Såfremt gruppen skal fjernes fra en person, kaldes metode *setGroup* med null som paramter: *setGroup(null).*

Metoden setGroup() sikrer ved hjælp af if-sætningen, at forbindelsen kun opdateres, hvis der sker en ænding af linkattributten ikke allerede er sat.

Koden for hele eksemplet kan ses i pakken *u_person0m_01group* i den udleverede kode.

Antag nu, at der i hver gruppe højst må være én person, som det er angivet i nedenstående figur.



Dette vil give anledning til præcis den samme kode som i foregående eksempel. Eneste forskel vil være, at nu skal der være en ekstra pre-betingelse, som sikrer, at gruppen ikke er forbundet med en anden person.

4. Enkeltrettet tvungen associering

Såfremt der i en associering indgår multipliciteten 1 ved en af klasserne, kaldes associeringen for tvungen. I en tvungen associering, skal den ene klasse altid være knyttet til den anden klasse.

Figuren herunder viser et eksempel på en tvungen associering: En person skal altid indgå i en gruppe.

I eksemplet vist i figuren herover vil Person klassen få en linkattribut kaldet group, som ikke kan være null. Det er derfor et oplagt valg at sørge for, at linkattributten sættes i Person klassens konstruktor. I kodeeksemplet herunder sættes linkattributten i Person klassens konstruktor (fremhævet med gul baggrund). Hvis forbindelsen ikke kan slettes eller opdateres, så er det alt der kræves.

```
public class Person {
    private String name;
    // forced association --> 1 Group
    private Group group;

    public Person(String name, Group group) {
        this.name = name;
        this.group = group;
    }

    public Group getGroup() {
        return group;
    }
}
```

Hvis forbindelsen skal kunne opdateres og/eller slettes, så skal Person klassen også have den samme setGroup() metode, som i det ikke-tvungne tilfælde (se afsnit 3).

Det er valgfrit, om linkattributten skal sættes i klassens konstruktor. Man kan i stedet vælge, at konstruktoren ikke tager en gruppe som parameter, og så sætte linkattributten ved at kalde setGroup()-metoden.
Koden for hele eksemplet kan ses i pakken *u_person0m_1group* i den udeleverede kode.

Antag nu, at retningen af forbindelsen er den anden vej, fra Group til person:

Her kan man vælge at lade Person klassens konstruktor sætte Group klassens linkattribut, så group-objektet peger tilbage på et Person-objekt (ved at kalde group.addPerson(this) i Person-konstruktoren).

```
public class Person {
    private String name;
       /**
        * Pre: group != null
        */
    public Person(String name, Group group) {
       this.name = name;
       group.addPerson(this);
    }
}
```

Koden for hele eksemplet kan ses i pakken *u_group1_0mperson* i den udleverede kode.

5. Enkeltrettet komposition

En komposition er som en associering en forbindelse imellem objekter. Kompositionen anvendes når et overordnet objekt (helheden) består af et antal underordnede objekter (delene), som er så stærkt knyttet til det overordnede objekt, at de ikke kan flyttes til et andet overordnet objekt. I en komposition har helheden ansvaret for at oprette delene.

I en komposition vil delene altid have en tvungen forbindelse til helheden. For en enkeltrettet komposition gælder, at retningen på kompositionen altid er fra helheden mod delene.

Et eksempel på en komposition er givet her:

En komposition programmeres meget lig en associering. Eneste ændring er, at add metoden fra associering erstattes af en create metode, som kan skabe en del.

I eksemplet herover får klassen *Group* en linkattribut kaldet *persons* af typen *ArrayList<Person>*. Da Group skal oprette delene, får klassen Group en *createPerson()* metode (som erstatter addPerson() metoden fra associering). Hvis der er brug for at kunne slette en del, skal helheden have en remove metode, som bryder forbindelsen mellem helheden og delen. I kodeeksemplet herunder er der tilføjet 2 metoder til Group klassen: getPersons() og createPerson().

```
public class Group {
    private String name;
    // composition --> 0..* Person
    private final ArrayList<Person> persons = new ArrayList<>();

    public ArrayList<Person> getPersons() {
        return new ArrayList<>(persons);
    }

    public Person createPerson(String personName) {
        Person person = new Person(personName);
        persons.add(person);
        return person;
    }
}
```

Bemærk, at i klassen *Person* skal konstruktoren have *package* synlighed. Hermed sikres, at konstruktoren for Person klassen kun kan bruges i samme pakke, som klassen Person er erklæret i. Klassen Group er den eneste klasse, som må kalde konstruktoren for Person klassen, og klasserne Group og Person vil altid være erklæret i samme pakke.

Koden for hele eksemplet kan findes i pakken *u_composition_group1_manyperson* i den udleverede kode.

6. Enkeltrettet aggregering

En aggregering er som en komposition en forbindelse mellem helhed og dele. En aggregering er dog ikke så stærk en forbindelse som en komposition. I en aggregering kan delen flyttes fra en helhed til en anden helhed. Endvidere er det mulig for delene at eksistere uden at være del af en helhed, hvis aggregeringen ikke er tvungen.

En aggregering programmeres med linkattributter og metoder, som om den var en associering. Den eneste forskel er, at en aggregering *kan* have en create metode, der opretter delene (samme metode som i en komposition).

Betragt nedenstående eksempel:

I kodeeksemplet herunder er tilføjet en *createPerson()* metode. Endvidere et det antaget, at en person skal kunne skifte tilhørsforhold til gruppe. Derfor er der også *addPerson()* og *removePerson()* metoder, der kan anvendes når en person skal skifte gruppe.

```
public class Group {
    // aggregering --> 0..* Person
    private final ArrayList<Person> persons = new ArrayList<>();

    public ArrayList<Person> getPersons() {
        return new ArrayList<>(persons);
    }
```

```
    public Person createPerson(String personName) {
        Person person = new Person(personName);
        persons.add(person);
        return person;
    }

    /**
     * Adds the person to this group,
     * if they aren't connected.
     * Pre: The person isn't connected to another group.
     */
    public void addPerson(Person person) {
        if (!persons.contains(person)) {
            persons.add(person)
        }
    }

    /**
     * Removes the person from this group,
     * if they are connected.
     */
    public void removePerson(Person person) {
        if (persons.contains(person)) {
            persons.remove(person);
        }
    }
}
```

Koden for hele eksemplet kan findes i pakken *u_aggregation_group01_omperson* i den udleverede kode.

7. Dobbeltrettede forbindelser

Dobbeltrettede forbindelser programmeres i store træk ved at se den dobbeltrettede forbindelse som to enkeltrettede forbindelser. Linkattributter og metoder er de samme som i de to enkeltrettede forbindelser, men indholdet af metoderne ændres lidt.

Ved en dobbeltrettet forbindelse skal to linkattributter opdateres, når forbindelsen oprettes eller slettes. Ansvaret for at opdatere begge linkattributter pålægges metoderne i model klasserne. Hermed skal man kun kalde én metode, som opdaterer begge linkattributter.

Derfor skal metoderne i klasserne ændres, så de opdaterer begge linkattributter.

I de følgende afsnit er vist eksempler på programmering af dobbeltrettede forbindelser.

8. Dobbeltrettet mange-til-mange associering

I dette afsnit beskrives, hvad der skal til for at implementere en dobbeltrettet associering, hvor en gruppe kan have mange personer tilknyttet, og en person kan være tilknyttet mange grupper. At associeringen er dobbeltrettet betyder, at personen skal kende de grupper personen er med i, og en gruppe kender alle de personer, der er tilknyttet gruppen. Dette er beskrevet ved nedenstående figur:

Betragtet som to enkeltrettede forbindelser skal klassen *Group* have en linkattribut kaldet *persons* af typen *ArrayList<Person>* (med metoder som getPersons(), addPerson() og evt. removePerson()), og klassen Person skal have en linkattribut kaldet *groups* af typen *ArrayList<Group>* (med metoder som getGroups(), addGroup() og evt. removeGroup()).

Kodeeksemplet herunder viser koden for begge klasser. Ændringerne i forhold til de enkeltrettede associeringer i afsnit 2 og 3 er fremhævet med gul baggrund.

```
public class Group {
    private String name;
    // association: --> 0..* Person
    private final ArrayList<Person> persons = new ArrayList<>();

    public Group(String name) {
        this.name = name;
    }

    public ArrayList<Person> getPersons() {
        return new ArrayList<>(persons);
    }

    /**
     * Adds the person to this group and the group to the person,
     * if they aren't connected.
     */
    public void addPerson(Person person) {
        if (!persons.contains(person)) {
            persons.add(person);
            person.addGroup(this);
        }
    }

    /**
     * Removes the person from this group and the grop from the person,
     * if they are connected.
     */
     public void removePerson(Person person) {
         if (persons.contains(person)) {
             persons.remove(person);
             person.removeGroup(this);
         }
```

```
     }
}

public class Person {
    private String name;
    // association: --> 0..1 Group
    private final ArrayList<Group> groups = new ArrayList<>();

    public Person(String name) {
        this.name = name;
    }

    public ArrayList<Group> getGroups() {
        return new ArrayList<Group>(groups);
    }

    /**
     * Adds the group to this person and the person to the group,
     * if they aren't connected
     */
    public void addGroup(Group group) {
        if (!groups.contains(group)) {
            groups.add(group);
            group.addPerson(this);
        }
    }

    /**
     * Removes the group from this person and the person from the group,
     * if they are connected
     */
     public void removeGroup(Group group) {
        if (groups.contains(group)) {
            groups.remove(group);
            group.removePerson(this);
        }
    }
}
```

Koden for hele eksemplet kan ses i pakken *b_group0m_0mperson* i den udleverede kode.

9. Dobbeltrettet 0..1-til-mange associering

I dette afsnit beskrives, hvad der skal til for at implementere en dobbeltrettet associering, hvor en gruppe kan have mange personer tilknyttet, og en person kan højst være tilknyttet én gruppe. At associeringen er dobbeltrettet betyder, at personen skal kende den gruppe personen er med i, og en gruppe kender alle de personer, der er tilknyttet gruppen. Dette er beskrevet ved nedenstående figur:

Betragtet som to enkeltrettede forbindelser skal klassen *Group* have en linkattribut kaldet *persons* af typen *ArrayList<Person>* (med metoder som getPersons(), addPerson() og evt. removePerson()), og klassen Person skal have en linkattribut kaldet *group* af typen *Group* (med metoder som getGroup() og setGroup().

Kodeeksemplet herunder viser koden for begge klasser. Ændringerne i forhold til de enkeltrettede associeringer i afsnit 2 og 3 er fremhævet med gul baggrund.

```
public class Group {
    // association --> 0..* Person
    private final ArrayList<Person> persons = new ArrayList<>();

    public ArrayList<Person> getPersons() {
        return new ArrayList<>(persons);
    }

    /**
     * Adds the person to this group,
     * if they aren't connected.
     */
    public void addPerson(Person person) {
        if (!persons.contains(person)) {
            persons.add(person);
            person.setGroup(this);
        }
    }

    /**
     * Removes the person from this group,
     * if they are connected.
     */
    public void removePerson(Person person) {
        if (persons.contains(person)) {
            persons.remove(person);
            person.setGroup(null);
        }
    }
}

public class Person {
```

```
    private String name;
    // association --> 0..1 Group
    private Group group; // nullable

    /** Note: Nullable return value. */
    public Group getGroup() {
        return group;
    }


    /**

     * Sets the group as this person's group,
     * if they aren't connected.
     */
    public void setGroup(Group group) {
        if (this.group != group) {
            Group oldGroup = this.group;
            if (oldGroup != null) {
                oldGroup.removePerson(this);
            }
            this.group = group;
            if (group != null) {
                group.addPerson(this);
            }
        }
    }
}
```

Da en person kun kan være tilknyttet til en gruppe, har metoden *setGroup()* ansvar for at fjerne en eventuelt ekstsiterende tilknytning til en anden gruppe, inden personen tilknyttes en ny gruppe. Det sker i denne del af koden:

```
            Group oldGroup = this.group;
            if (oldGroup != null) {
                oldGroup.removePerson(this);
            }
```

Dernæst opdateres linkattributten i begge retninger:

```
            this.group = group;
            if (group != null) {
                group.addPerson(this);
            }
```

Koden for hele eksemplet kan ses i pakken *b_group01_0mperson* i den udleverede kode.

10. Dobbeltrettet 0..1-til-mange komposition

I dette afsnit beskrives, hvad der skal til for at implementere en dobbeltrettet komposition, hvor en gruppe kan oprette og tilknytte personer. En person kan kun være tilknyttet den gruppe, der har oprettet personen og derfor er der på *Person* klassen ikke metoder, der kan ændre på, hvilken gruppe personen tilhører. At kompositionen er dobbeltrettet betyder, at personen skal kende den gruppe personen er med i, og en gruppe kender alle de personer, der er tilknyttet gruppen. Dette er beskrevet ved nedenstående figur:

Linkattrbutterne er de samme som for den dobbeltrettede associering beskrevet i afsnit 9. Da det er en komposition har *Group* en metode til at oprette en person og tilknytte den til gruppen. Da personen skal kende den gruppe den tilhører (tvungen sammenhæng), skal constuctor på *Person* klassen tage en gruppe som paramter og constuctoren skal ikke være public.

Da der er en komposition fra *Group* til Person kan personen ikke ændre hvilken gruppe den tilhører og derfor har *Person* klassen **ikke** en *setGroup()* metode.

Kodeeksemplet herunder viser koden for begge klasser.

```
public class Group {
    private String name;
    // composition: --> 0..* Person
    private final ArrayList<Person> persons = new ArrayList<>();

    public Group(String name) {
        this.name = name;
    }

    public ArrayList<Person> getPersons() {
        return new ArrayList<>(persons);
    }

    public Person createPerson(String personName) {
        Person person = new Person(personName, this);
        persons.add(person);
        return person;
    }

    public void removePerson(Person person) {
        if (persons.contains(person)) {
            persons.remove(person);
        }
    }
}
```

```
public class Person {
    private String name;
    // composition: --> 1 Group
    private Group group;

    Person(String name, Group group) { // package visibility
        this.name = name;
        this.group = group;
    }

    public Group getGroup() {
        return group;
    }
}
```

Koden for hele eksemplet kan ses i pakken *b_composition_group1_0mperson* i den udleverde kode.

11.Metoder der ikke skal med

Ved programmering af forbindelser medtages kun de metoder, der er nødvendige i den aktuelle applikation.

Hvis der i en applikation f.eks. ikke er brug for at bryde en forbindelse imellem to objekter, så kan remove-metoden udelades.

Kodeeksemplet herunder viser koden for klasserne Group og Person idet tilfælde, hvor det ikke skal være muligt at opdatere eller slette forbindelser efter de er oprettet. Eksemplet er lavet med udgangspunkt i den dobbeltrettede associering beskrevet i modellen:


```
public class Group {
    private String name;
    // association: --> 0..* Person
    private final ArrayList<Person> persons = new ArrayList<>();

    public Group(String name) {
        this.name = name;
    }

    public ArrayList<Person> getPersons() {
        return new ArrayList<>(persons);
    }
```

```
    /**
     * Adds the person to this group,
     * if they aren't connected.
     * Pre: The person isn't connected to another group.
     */
    public void addPerson(Person person) {
        if (!persons.contains(person)) {
            persons.add(person);
            person.setGroup(this);
        }
    }
}

public class Person {
    private String name;
    // association: --> 0..1 Group
    private Group group;

    public Person(String name) {
        this.name = name;
    }

    public Group getGroup() {
        return group;
    }

    /**
     * Sets the group as this person's group
     */
    public void setGroup(Group group) {
        if (this.group != group) {
            this.group = group;
            if (group != null) {
                group.addPerson(this);
            }
        }
    }
}
```

Bemærk ingen removePerson metode

setGroup()-metoden bliver simplere, da den ikke først skal slette en eksisterende sammenhæng

Koden for hele eksemplet kan ses i pakken *b_group01_0mperson_noremove* i den udleverede kode.

12. Afslutning

I de foregående afsnit er beskrevet, hvorledes forbindelser mellem klasser kan implementeres med linkattributter og metoder. I denne note sikrer metoderne i model klasserne, at modellen ikke kan ende i en ulovlig tilstand.

En anden tilgang til at programmere dobbeltrettede sammenhænge er, at programmere sammenhængene i modelklaserne, som om de altid er enkeltrettede. Det er da den klasse(i vores arkitektur Controlleren), der kalder disse metoder, der skal sørge for at kalde link-metoder på begge modelklasser, for at sikre at linkattributterne passer sammen.

I denne note er vist eksempler på de oftest forekommende forbindelser, men der vil komme situationer, hvor en design klassemodel indeholder kombinationer af forbindelser og multipliciteter, der ikke er med i denne note. I sådanne situationer tages der udgangspunkt i det eksempel fra noten, der passer bedst til den konkrete situation, og derefter laves de tilpasninger, der er nødvendige. Det er vigtigt altid at huske at opdatere begge linkattributter, når der er tale om en dobbeltrettet forbindelse, samt at sikre at multipliciteter ikke kan blive forkerte.