

Klasser och arv (1/2)

Tobias Wrigstad



Home

News

Information

Lectures

Achievements

Schedule

Dashboard

[Previous entry](#) | [Next entry](#)

Senaste nytt

2014-11-03

Genomgång av vanliga [kompilerfel i Java](#) i kursrepot.

2014-10-26

Genomgång av C-delen av kodprov 1 (2014-10-16) finns [här](#).

2014-10-21

1. [Länkbiblioteket](#) från förra årets kurshemsida har flyttats över hit.
2. [Kursens repo](#) innehåller nu övningsuppgifter för C och Java från kursinstansen 2012.

2014-10-15

Schemat för kodprovet 16/10 finns nu tillgängligt [här](#)

2014-10-14

Sidan för första [kodprovsövningen](#) har fått en länk till koden som skrevs då.



Länkbibliotek

tags: [länkar](#), [ofärdig](#)

Kurslitteratur och läsanvisningar

Kurslitteraturen är endast tips på känt bra böcker. Kursen följer inte dessa specifikt.

- [K&R]
 - B. Kernighan and D. Richie, *The C Programming Language*, 2:a upplagan, Prentice Hall (1988)
- [PKA]
 - D. Poo, D. Kiong, S. Ashok, [Object-Oriented Programming and Java](#), Springer Verlag, 2nd ed., 2008, XIX.
- [Git]
 - Scott Chacon, *Pro Git*, <http://git-scm.com/book/>
- [K&P]
 - B. Kernighan and R. Pike, *The Practice of Programming*, Addison-Wesley, 4 Feb 1999, isbn-10: 020161586X, isbn-13: 978-0201615869.

Länkbibliotek

- [GNU Emacs Manual](#)
 - Explains all the details of emacs.
- [How to move the cursor efficiently in Emacs](#)
 - Moving the cursor is more than meets the eye.
- [Copying and pasting \(aka kill and yank\) in Emacs](#)
 - How to delete stuff and paste it again.
- [The undo command in Emacs](#)
 - The undo command can be confusing. Either never make mistakes or read this.



Små övningsuppgifter

Övningar till F1 och F2

1. Skriv ett program som läser tecken från standard input och räknar antalet meningar. Meningar avslutas med punkt, utropstecken eller frågetecken.
2. Skriv ett program som kopierar standard input till standard output. Vid kopieringen skall all text omgiven av < och > inklusive dessa tecken utelämnas.
3. Skriv ett program som läser tecken från standard input och räknar antalet ord. Med *ord* menas en obruten följd av bokstäver.
4. Modifiera programmet ovan så att det räknar antalet bokstäver det är i det lästa.
5. Skriv en C-funktion för att räkna antalet bokstäver i en sträng.

1 +

Vilka parametrar och

6. Skriv ett C-program som räknar antalet personer i en lista med $n = 1, 2, \dots, 10$

Övningar

1. Skriv en klass Person med en konstruktör som tar namn och personnummer som indata, och som håller reda på hur många personer som har instantierats sedan programmet startades. Det sistnämnda görs lämpligen med en *privat klassvariabel*. När skall den räknas upp? Hur kan man garantera att den alltid räknas upp? Skriv även en instansmetod (vanlig metod) `int getCount()` som returnerar klassvariabelns värde.
2. Skriv personklassen så att utomstående inte har direkt åtkomst till ett personobjekts namn och personnummer. Namn skall gå att byta, men inte personnummer.
3. Utöka personklassen ovan så att *klassen* person har en lista över samtliga personer som skapats i systemet. Modifiera `getCount()` till att returnera denna listas längd istället för att ha en räknare. Finns det några problem med denna typ av design? Vad får det för effekt på minneshantering?



Några förändringar i kursen

- Mål Z70 ändras:
Endast **ett** fullständigt Java-program krävs
- Mål L33–35 utgår (vi återkommer till de 2 personer som gjort L33)
- **Fler** lärarledda genomgångar
Denna fredag 7/11: 13–15 och 15–17; Javaprogrammering (mer info på mail)



Liten ordlista

Svenska	Engelska	Svenska	Engelska
Objekt	Object	Metodspecialisering	Overriding
Klass	Class	Överlagring	Overloading
Arv	Inheritance	Överskuggning	Shadowing
Instansvariabel / fält	Instance variable / field	Klasshierarki	Class hierarchy
Metod	Method	Aggregering	Aggregation
Superklass / basklass	Super class / base class	Typomvandling	Type cast
Subklass / härledd klass	Sub class / derived class	Polymorfism	Polymorphism
Abstrakt klass	Abstract class	Barnklass	Child / sub / derived class
Superanrop	Super call	Instantieras	Instantiate



Ett socialt nätverk

Det sociala nätverket FooBar fungerar ungefär som Facebook. Varje användare har en profil, och varje profil är knuten till noll eller flera vänner. Det finns två sorters vänner, nära vänner och övriga. En användare kan posta statusuppdateringar som kan nämna andra profiler. En statusuppdatering kan gillas eller ogillas av användare (inklusive postaren själv), samt kommenteras på. Varje kommentar kan också gillas eller ogillas.

Till varje profil finns knuten en händelselogg som innehåller statusuppdateringar i omvänt kronologisk ordning, dvs. senast först. I en händelselogg för användaren A visas alla statusuppdateringar som A har gjort, samt alla relevanta statusuppdateringar för alla vänner. En statusuppdatering hos en nära vän anses alltid relevant, men för övriga gäller att den skall vara "het" eller att A nämns i statusuppdateringen eller någon av dess kommentarer. En statusuppdatering anses vara het om den antalet gillanden + antalet ogillanden + antalet kommentarer överstiger ett visst tröskelvärde T.

Användare kan "knuffa till" varandra. Om A knuffar till B betyder detta att A:s alla statusuppdateringar anses relevanta för B i en vecka och tvärtom. Man kan inte ha fler än 5 "aktiva knuffar" samtidigt.

En användare kan be om att få bli vän med en annan användare, som måste tacka ja först. Vänskap är en reflexiv relation. En användare kan lista en eller flera vänner som nära vänner. Nära vänskap är inte nödvändigtvis reflexiv. En särskild kategori av nära vänskap finns dock som är reflexiv, nämligen slätskap och "i ett förhållande". En användare kan lista en eller flera vänner som släkt/i ett förhållande, och dessa måste tacka ja först. Man kan utan vidare säga upp alla typer av vänskap, inklusive nära vänskap och förhållanden.



Varje **användare** har en **profil**, och varje profil är knuten till noll eller flera **vänner**. Det finns två sorters vänner, **nära vänner** och **övriga**. En användare kan posta **statusuppdateringar** som kan nämna andra profiler. En statusuppdatering kan gillas eller ogillas av användare (inklusive postaren själv), samt kommenteras på. Varje **kommentar** kan också gillas eller ogillas.

Till varje profil finns knuten en **händelselogg** som innehåller statusuppdateringar i omvänt kronologisk ordning, dvs. senast först. I en händelselogg för användaren A visas alla statusuppdateringar som A har gjort, samt alla relevanta statusuppdateringar för alla vänner. En statusuppdatering hos en nära vän anses alltid relevant, men för övriga gäller att den skall vara "het" eller att A nämns i statusuppdateringen eller någon av dess kommentarer. En statusuppdatering anses vara het om den antalet gillanden + antalet ogillanden + antalet kommentarer överstiger ett visst tröskelvärde T.



Objekten

- Användare

- Profil

- Vänner

Det finns nära vänner

... och vanlig vänskap

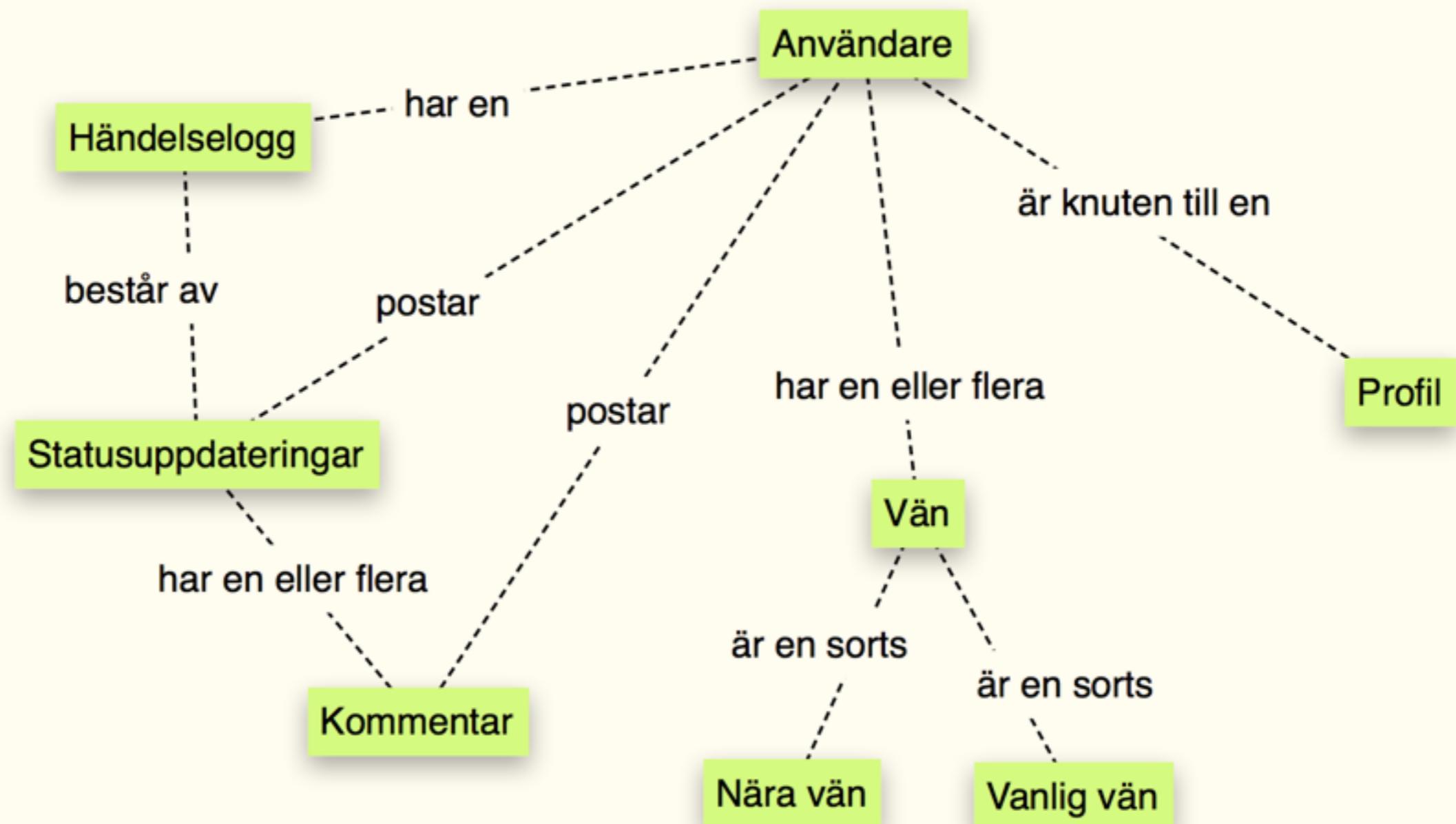
- Statusuppdateringar

- Kommentarer

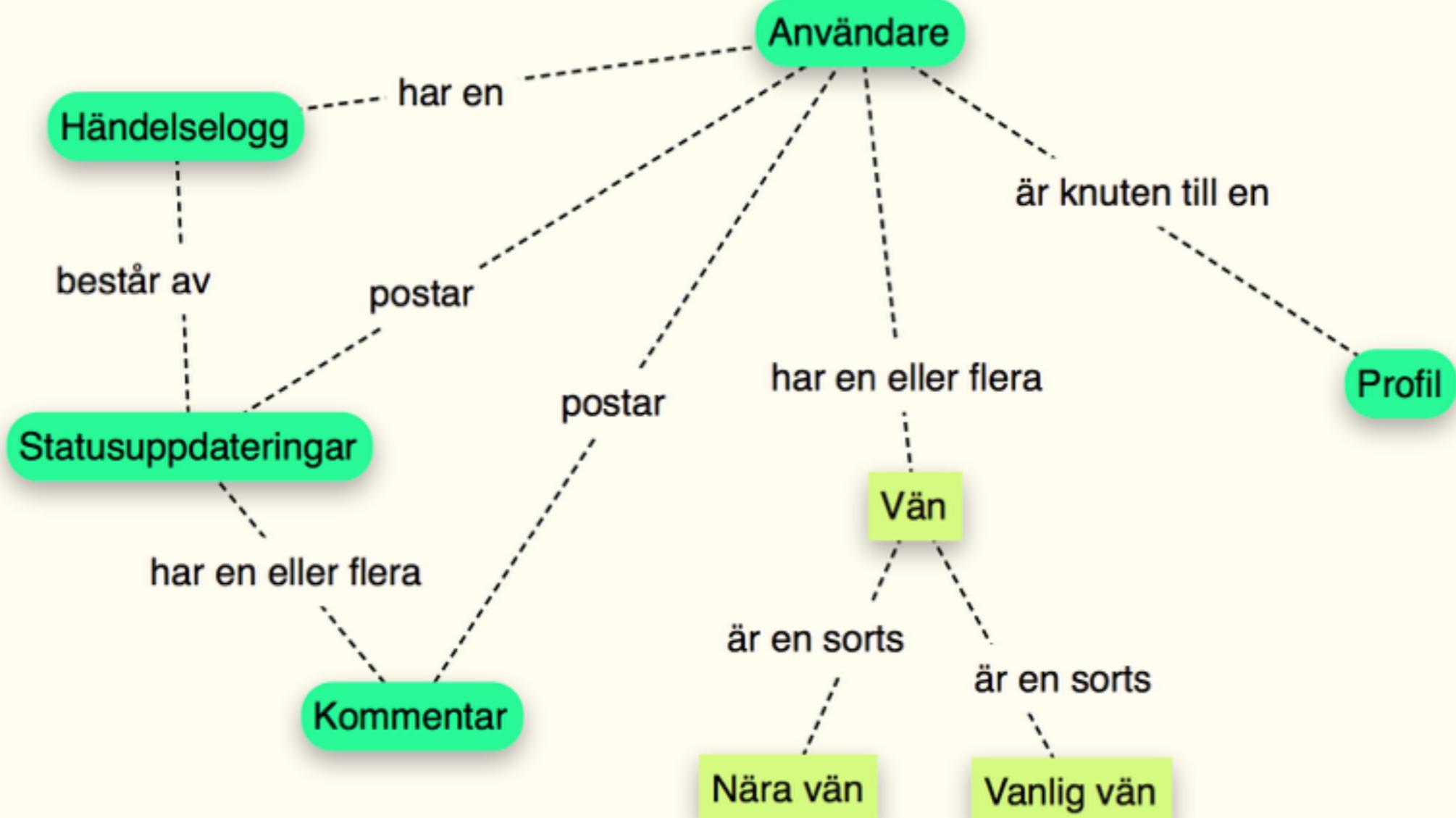
- Händelseslogg



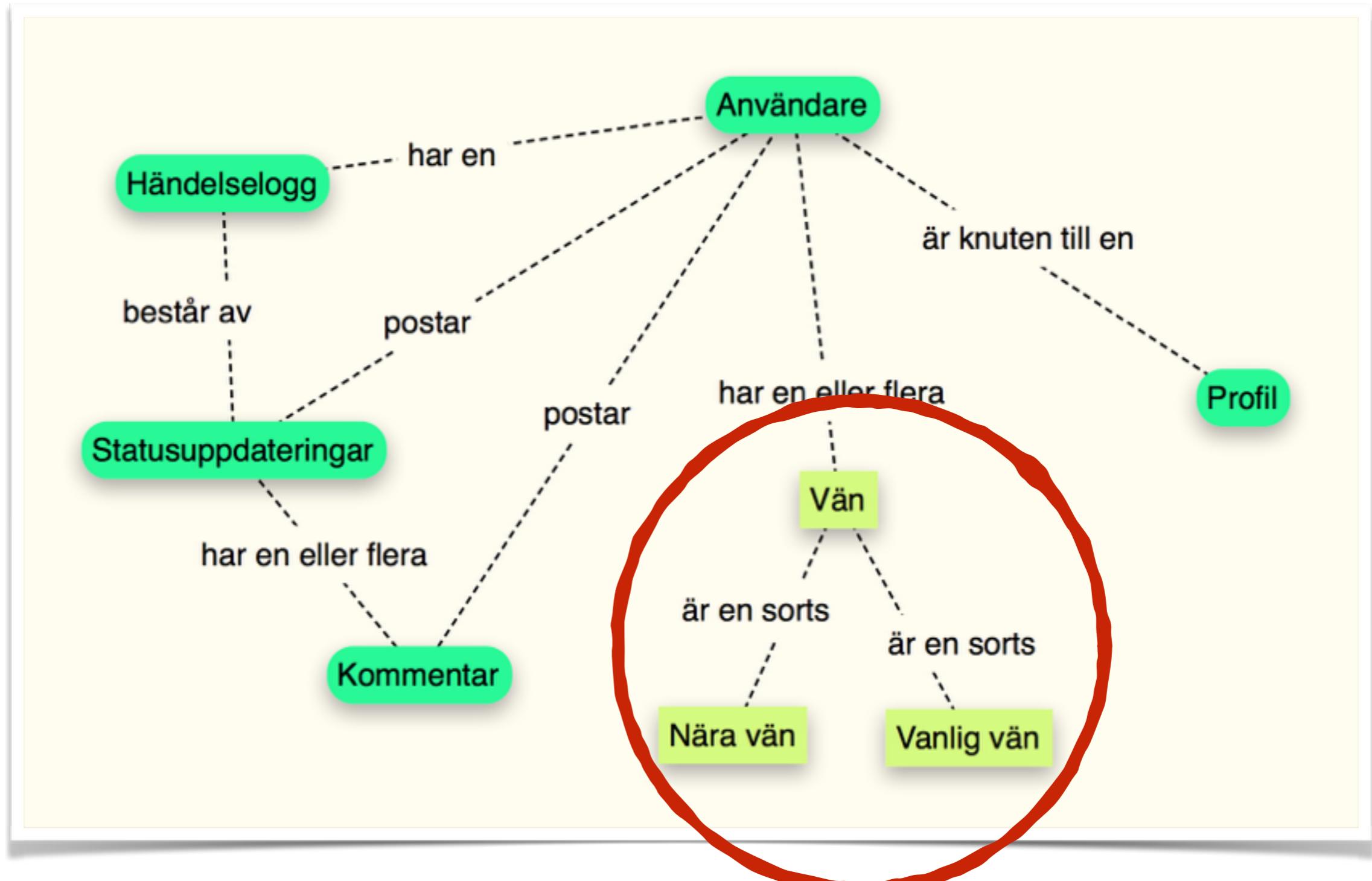
Objektens relationer



Klasser



Inte klasser!



Varför är ”vän” inte en klass?

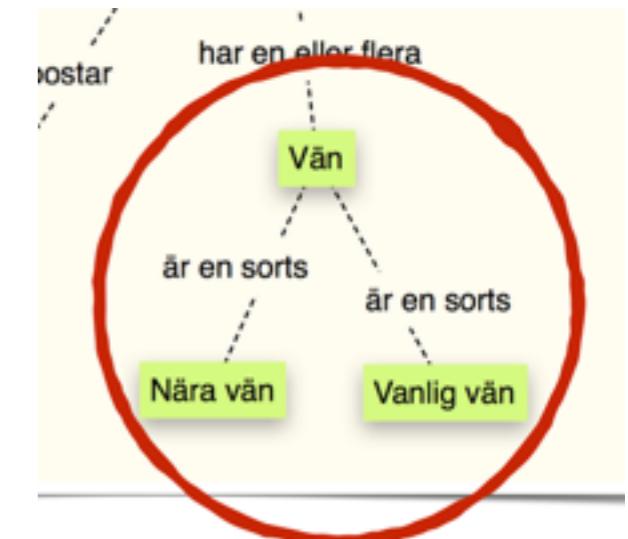
- Vänskap är en **relation** mellan två objekt

Att vara en vän är inte något som definierar användarkonceptet

- Det räcker med två användare för att modellera vänskap

```
class User {  
    User[] friends;  
    User[] closeFriends;  
  
    void addFriend(User u) {  
        friends[...] = u;  
        u.friends[...] = this;  
    }  
}
```

- **Vänskap** skulle kunna vara en klass, men en vän är det inte!

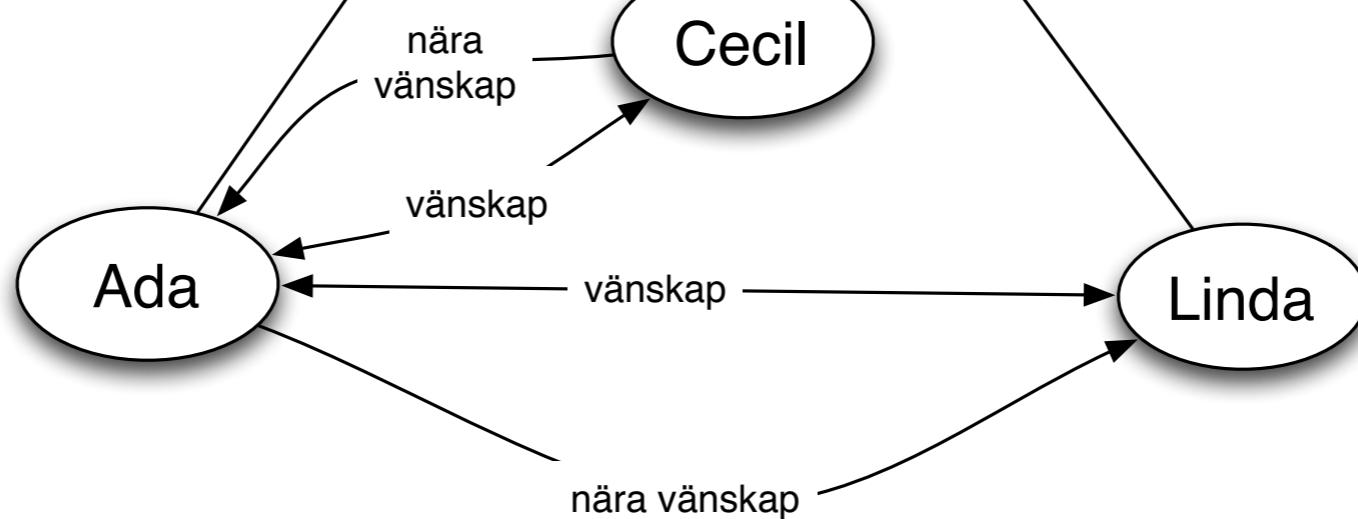


klass



"is a"

instans



relationer mellan objekt



UU

```
class User {  
    User[] friends;  
    User[] closeFriends;  
  
    void addFriend(User u) {  
        friends[...] = u;  
        u.friends[...] = this;  
    }  
}
```

Relationen modellerad som pekare
mellan objekten

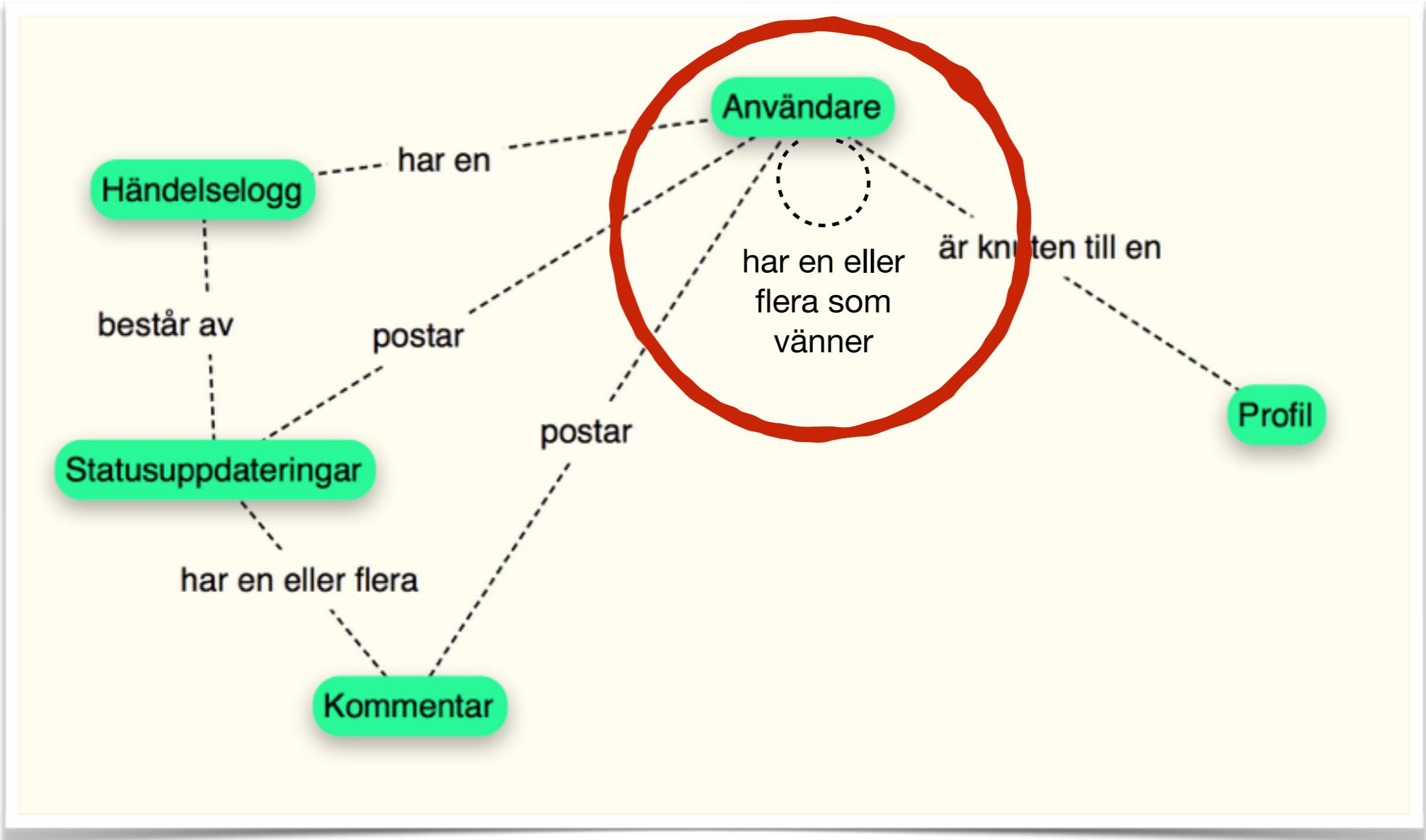


```
class Relation { ... }  
class FriendShip extends Relation { ... }  
  
class User {  
    Relation[] connections;  
  
    void addFriend(User u) {  
        FriendShip f = new FriendShip(this, u);  
        connections[...] = f;  
        u.connections[...] = f;  
    }  
}
```

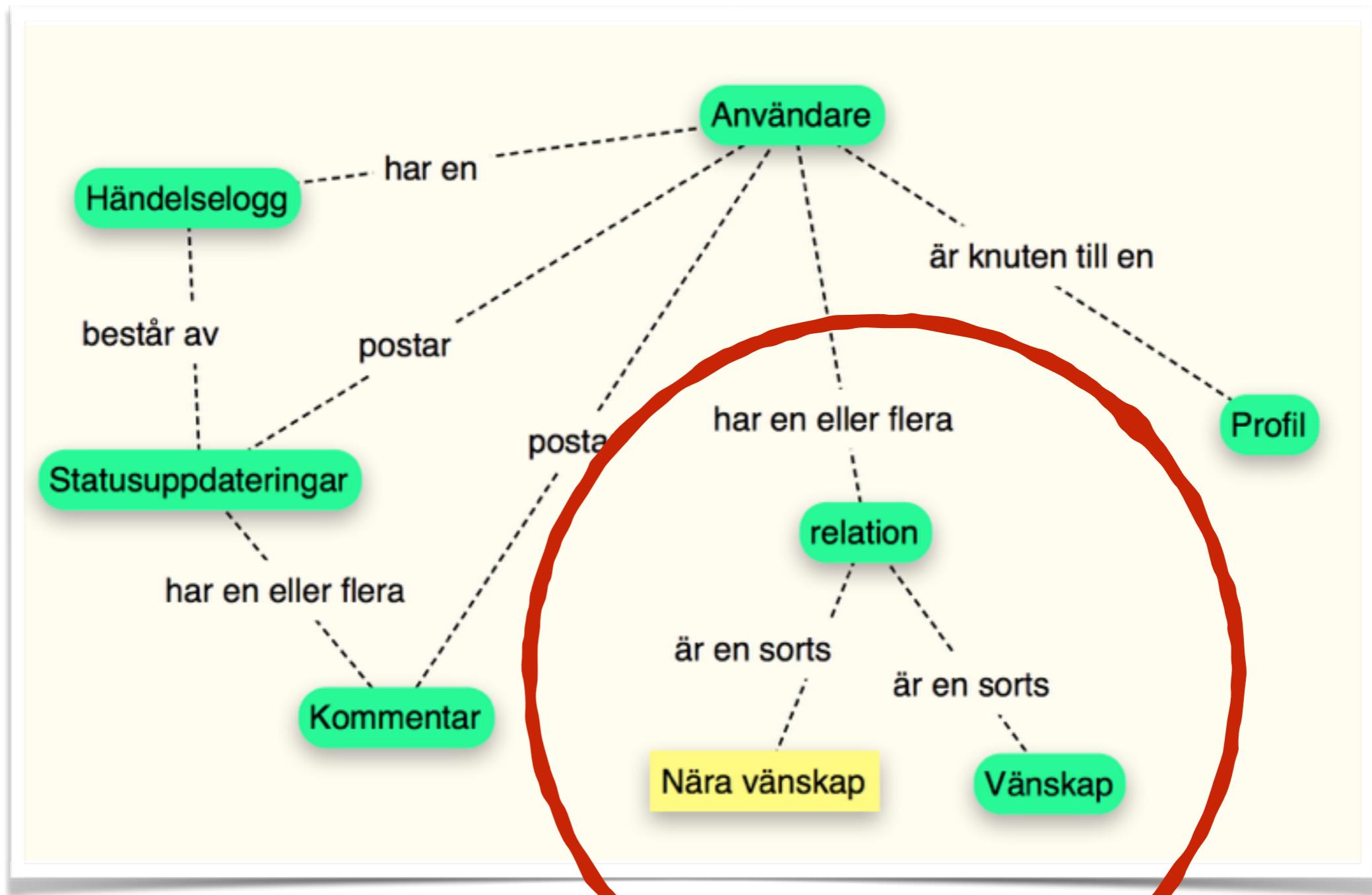
Relationen modellerad
som instanser av
specifika klasser



Mycket bättre (1/2)



Mycket bättre (2/2)



Klasser från objekt

- De flesta objekt vi såg är exempel på koncept i domänen för systemet och bör därför ha motsvarande klasser

Programmet är en modell av verkligheten!

Vissa egenskaper hos objekt gav inte upphov till nya klasser

Ibland kan det vara vettigt att modellera relationer som objekt / klasser

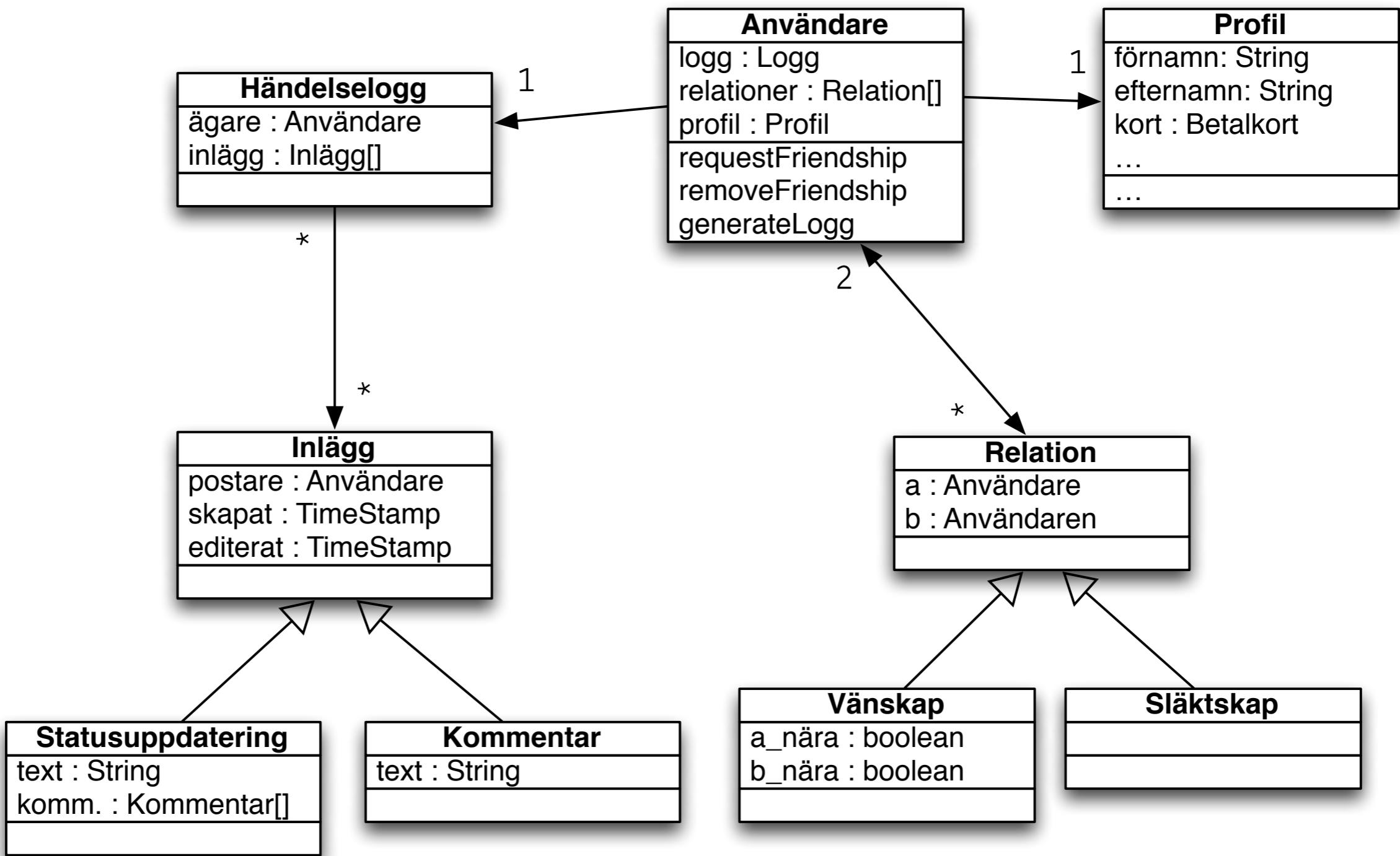
- **Arv:** relationer mellan klasser

Olika typer av relationer (alla linjer med "är en sorts")

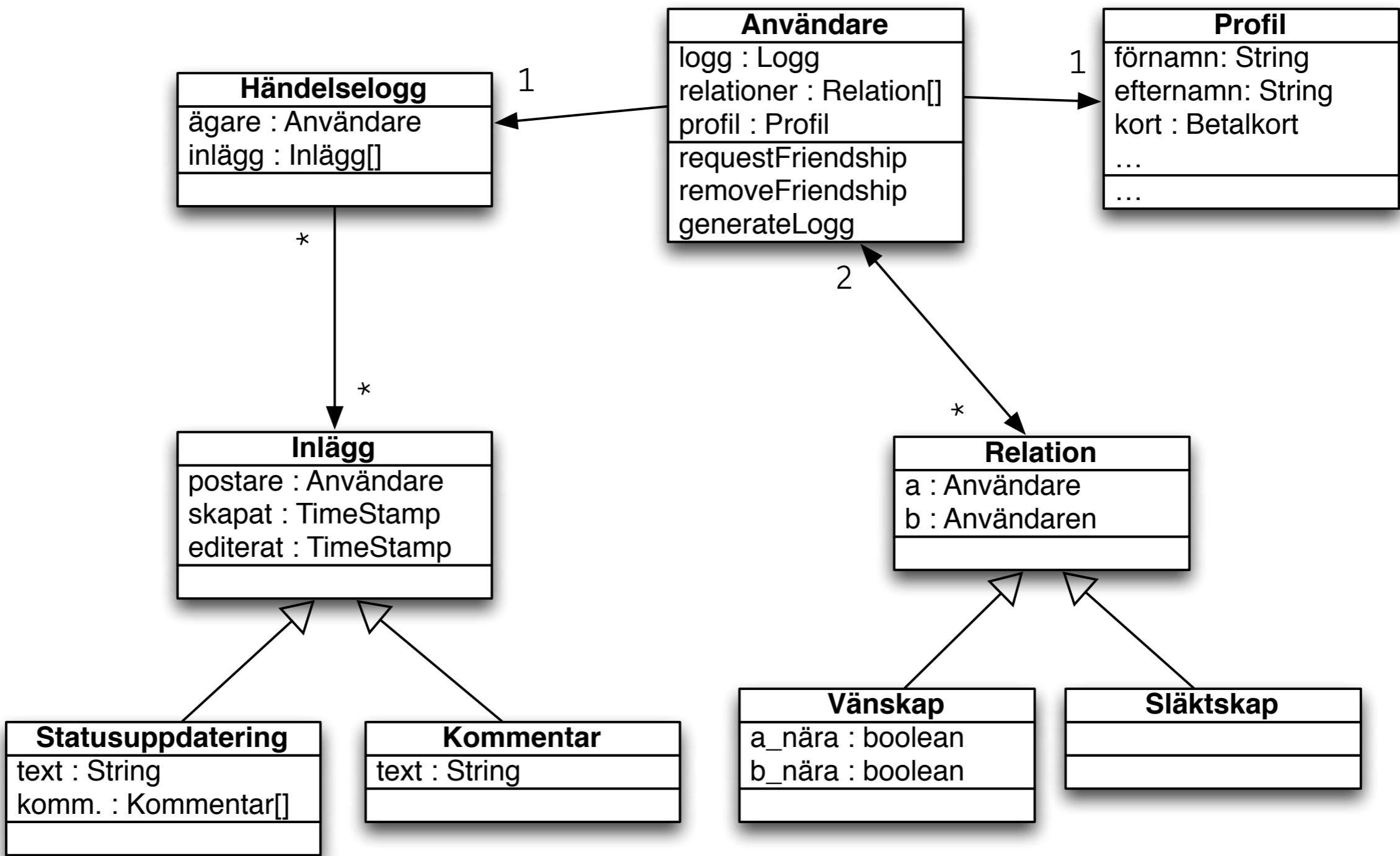
Ibland kan det vara vettigt att generalisera och skapa en gemensam **superklass** för klasser som är väldigt lika



Ett initialt klassdiagram [UML]



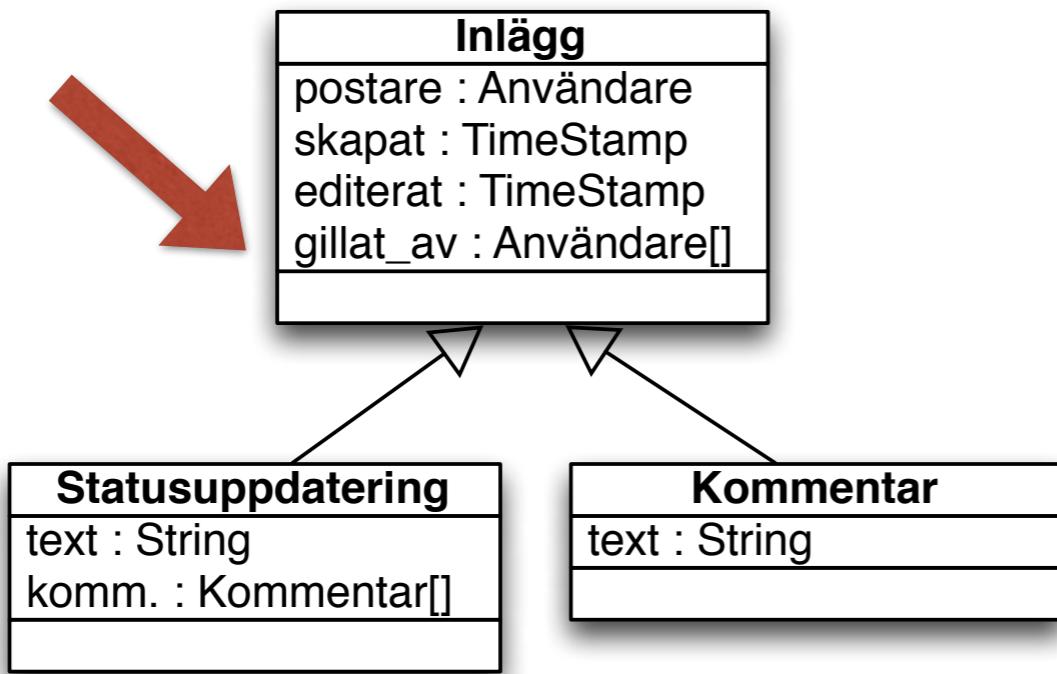
Ett initialt klassdiagram [UML]



Var/hur lägger vi till stöd för att gilla inlägg?



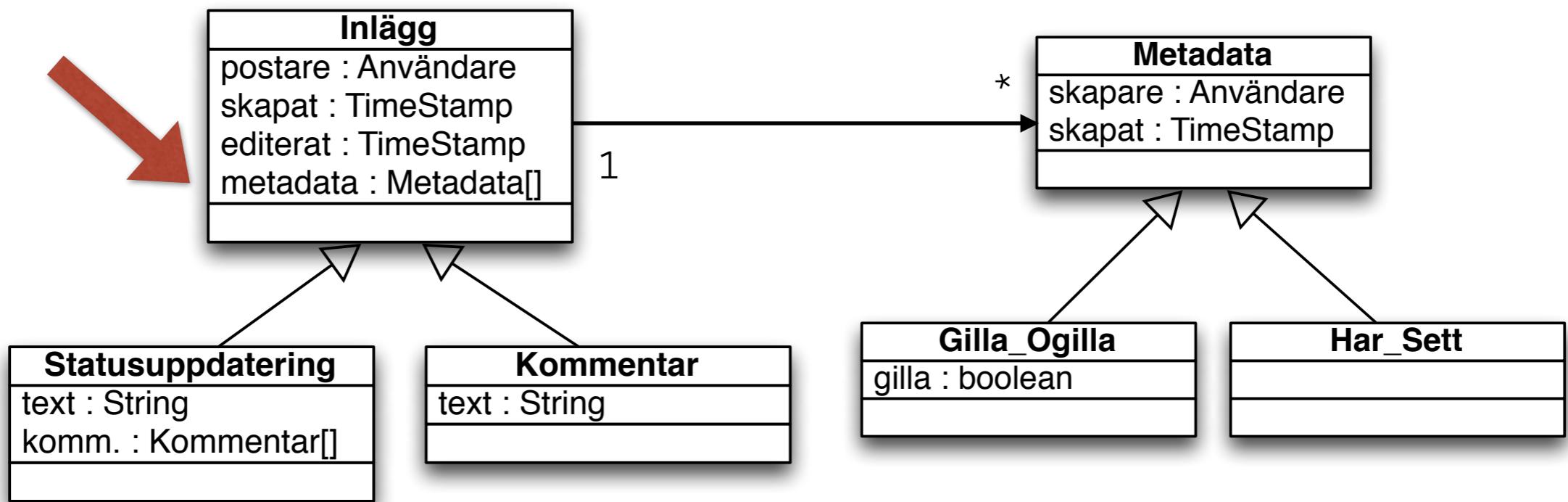
Försök #1: "gillat_av"



Om en användare gillar ett inlägg, lägg till vederbörande i en lista "gillat_av"



Försök #2: "metadata"



Associera varje inlägg med godtyckligt metadata



Skillnader mellan försök #1 och #2

■ Försök #1

Logiken för metadata ligger i Inlägg-klassen

Måste skapa en ogillat_av, sett_av, etc. för varje ny egenskap man vill lägga till

Enkelt och direkt

■ Försök #2

Logiken för metadata är fakturerat ut ur inlägg och kan ändras "fritt"

Enkelt att lägga till en ny typ av metadata

Mer komplicerat

Måste följa någon form av mönster för att "dra nytta av ny metadata"



Exempel på hur man kan koppla loss metadata och inlägg

```
class Gilla_Ogilla extends Metadata {  
    String decorate(String s) {  
        // if s has like-data from before,  
        // append else add  
    }  
}  
  
class Inlägg {  
    Metadata[] metadata;  
    String text;  
    String render() {  
        String html = text;  
        for (Metadata m : metadata) {  
            html = m.decorate(html);  
        }  
        return html;  
    }  
}
```

Arv (eng. inheritance)

- Låt A och B vara klasser; A har variablerna X och Y, samt metoderna M och N
- Om B ärver av A får B också X och Y och M och N
- Arv fångar generalisering—specialisering (superklass—subklass)

Undvika upprepning av kod

Fångar relation mellan klasser i koden

I statiskt typade språk som Java (C++, C#, m. fl.): viktig för polymorfism

- Metadata superklass, Gilla_Ogilla och Sedd är subklasser till Metadata

I en array av Metadata kan man blanda Gilla_Ogilla och Sedd

- Metadata borde vara en **abstrakt** klass som inte kan instantieras



Partiell klasshierarki för relationer mellan personer

```
abstract class Relation {  
    Person a;  
    Person b;  
}  
  
class Friendship extends Relation {  
    boolean a_close;  
    boolean b_close;  
}  
  
class Family extends Relation {}
```

Arv och konstruktorer

```
abstract class Relation {  
    Person a;  
    Person b;  
  
    Relation(final Person a, final Person b) {  
        assert(a.equals(b) == false);  
        this.a = a;  
        this.b = b;  
    }  
}  
  
class Friendship extends Relation {  
    boolean a_close;  
    boolean b_close;  
    Friendship(final Person a,  
               final Person b,  
               final boolean a_close,  
               final boolean b_close) {  
        super(a, b);  
        this.a_close = a_close;  
        this.b_close = b_close;  
    }  
    static Friendship mutualCloseFriendship(final Person a, final Person b) {  
        return new Friendship(a, b, true, true);  
    }  
}
```

Abstrakta klasser

```
abstract class Relation { ... }
```

```
Person p1 = ...;
```

```
Person p2 = ...;
```

```
Relation r = new Relation(p1, p2); // kompilerar ej
```

OO i C

```
abstract class Relation {
    Person a;
    Person b;

    Relation(final Person a, final Person b) {
        assert(a.equals(b) == false);
        this.a = a;
        this.b = b;
    }
}

struct relation
{
    person_t *a;
    person_t *b;
};

struct relation *init_relation(struct relation *r, person_t a, person_t b)
{
    assert(cmp_person(a, b) != 0);
    r->a = a;
    r->b = b;
    return r;
}

struct relation *new_relation(person_t a, person_t b)
{
    assert(false); // relation is abstract!
}
```

OO i C

```
class Friendship extends Relation {  
    boolean a_close;  
    boolean b_close;  
    Friendship(final Person a,  
               final Person b,  
               final boolean a_close,  
               final boolean b_close) {  
        super(a, b);  
        this.a_close = a_close;  
        this.b_close = b_close;  
    }  
}  
  
struct friendship  
{  
    struct relation;  
    bool a_close;  
    bool b_close;  
};  
  
struct relation *init_friendship(struct friendship *f,  
                                 person_t a, person_t b, bool a_close, bool b_close)  
{  
    init_relation(f);  
    f->a_close = a_close;  
    f->b_close = b_close;  
    return f;  
}  
  
struct relation *new_friendship(person_t a, person_t b, bool a_close, bool b_close)  
{  
    return init_friendship(malloc(sizeof(struct friendship)), a, b, a_close, b_close);  
}
```

Nästa föreläsning

- Skall vi titta mer på beteende och arv
- Overriding
- Overriding ≠ overloading
- Superanrop
- Enkelt arv vs. multipelt arv



Hur tolkar man javakompilatorns felmeddelanden

```
<Filnamn.Java>:<Radnummer>: error: <Beskrivning av felet>  
    <information om var det uppstår>  
    <övrig hjälpinformation om tillämpligt>
```

```
CommonCompilerErrors.java:66: error: cannot find symbol  
    LinkedList myList;  
    ^  
        symbol:   class LinkedList  
        location: class ErrorThree
```



Förstå kompilatorns språk

Kompilatorn säger	Betyder i regel
cannot find symbol	Felstavat namn, eller namnet är inte synligt ännu, t.ex. inte importerat in
method X cannot be applied	Argumentlistans typer fel (för få argument, fel ordning, fel argument?)
incompatible types	Typen på högersidan är inte <i>kompatibel</i> med den till vänster Är de subtyper?
X cannot be converted to String	Glömt att anropa <code>toString()</code> ?



SIMPLE, ett tiostegsprogram för programmering

- 1 **Gör en work breakdown structure på hög nivå**, och läs specifikationen och leta specifikt efter verb (funktioner/beteende) eller substantiv (data/objekt/klasser)
- 2 Skriv kod för att pröva om du tänkt rätt (**tänk på vad som är rätt och hur man kollar det**)
- 3 Ha **alltid ett fungerande program**
- 4 **Kompilera efter varje förändring**
- 5 **Kör programmet hela tiden** för att hitta fel (eller ännu bättre — kör testen!)
- 6 **Dela upp varje problem i delproblem rekursivt, börja lösa först när något verkar enkelt**
- 7 **Dela upp varje delproblem** i mindre steg, **börja med de enklaste**, och spara loopar till sist
 - Om möjligt, **börja med en straight-line-version** (alltså utan if-satser)
 - När den fungerar **lägg till eventuella if-satser en i taget**
 - Om du skriver en loop, gör ovanstående två steg först och **lägg till loopen sist**
- 8 **Fuska** (cheat) varje gång du riskerar att fastna
- 9 **Skarva** (dodge) för att förenkla specifikationer och skapa fler enklare delsteg
- 10 Växla mellan att
 - **tänka**,
 - **koda**, och ibland
 - **refaktorera** (speciellt dina fusk och skarvar).

