TDDE71
Arv, Polymorfi, OOA

Eric Ekström

Institutionen för datavetenskap

LINKÖPING UNIVERSITY

1 Specialisering (Arv)
2 Statisk binding
3 Dynamisk binding - Polymorfi
4 Mer om arv
5 Synlighet
6 Objektoriend analys
7 UML

Löneberäkning för en liten startup

```
vector<Programmer> v{};

v.push_back(Programmer{"Emma", "070- 44 55 66", 20000_kr});
v.push_back(Programmer{"Eric", "070- 11 22 33", 20000_kr});
v.push_back(Programmer{"Klas", "013- 28 21 47", 20000_kr});

for (Programmer const& p : v)
{
  cout << p.get_salary() << endl;
}
```

Startupens programmerare

```
class Programmer
{
public:
  Programmer(string const& n, string const& p, int s);

  int    get_salary() const;
  string get_phone() const;
  void   print_info(ostream& os) const;

private:
  string name;
  string phone;
  int month_salary;
};
```

Startupen hyr in konsulter

```
class Consultant
{
public:
  Consultant(string const& n, string const& p, int hp, int bh);

  int    get_salary() const;
  string get_phone() const;
  void   print_info(ostream& os) const;

private:
  string name;
  string phone;
  int hour_pay;
  int billable_hours;
};
```

Problem 1: Jämför Programmer och Consultant!

Klasserna är praktiskt taget identiska.

KODUPPREPNING!

Problem 2: Hur skriver vi nu löneberäkningen?

```
vector< ??? > v{};
v.push_back(Programmer{"Emma", "070- 44 55 66", 20000_kr});
v.push_back(Programmer{"Eric", "070- 11 22 33", 20000_kr});
v.push_back(Programmer{"Klas", "013- 28 21 47", 20000_kr});

v.push_back(Consultant{"Nils", "070- 77 88 99", 800_kr_h, 80_h});

sort( begin(v), end(v), ??? );

for ( ??? const& p : v)
{
  cout << p.get_salary() << endl;
}
```

- Kan vi använda en vektor för programmerare och en för konsulter?
- Kan vi baka ihop programmerare och konsulter till en klass?

Vi skapar en klass med det som är gemensamt

Vi tänker först ut en gemensam generalisering:
Både programmerare och konsulter *ÄR* anställda!

```
class Employee
{
public:
  Employee(string const& n,
           string const& p);

  int    get_salary() const;
  string get_phone() const;
  void   print_info(ostream& os) const;

private:
  std::string name;
  std::string phone;
};
```

Nu kan vi specialisera med *arv*!

Vi skriver om vår Programmer och Consultant med Employee som *basklass*. Programmer och Consultant blir *härledda* klasser.

```
class Programmer : public Employee
{
public:
  Programmer(string const& n, string const& p, int s);
private:
  month_salary;
};

class Consultant : public Employee
{
public:
  Consultant(string const& n, string const& p, int hp, int bh);
private:
  int hour_pay;
  int billable_hours;
};
```

Specialiserad get_salary()!

Låt oss även definiera respektive `get_salary()` och konstruktor!
Datamedlemsinitieringslistan ska anropa basklasskonstruktorn.

```
Programmer::Programmer(string const& n, string const& p, int s)
 : Employee{n, p}, month_salary{s}      {}

int Programmer::get_salary() const
{
  return month_salary;
}

Consultant::Consultant(string const& n, string const& p, int hp, int bh);
 : Employee{n, p}, hour_pay{}, billable_hours{}      {}

int Consultant::get_salary() const
{
  return hour_pay*billable_hors;
}
```

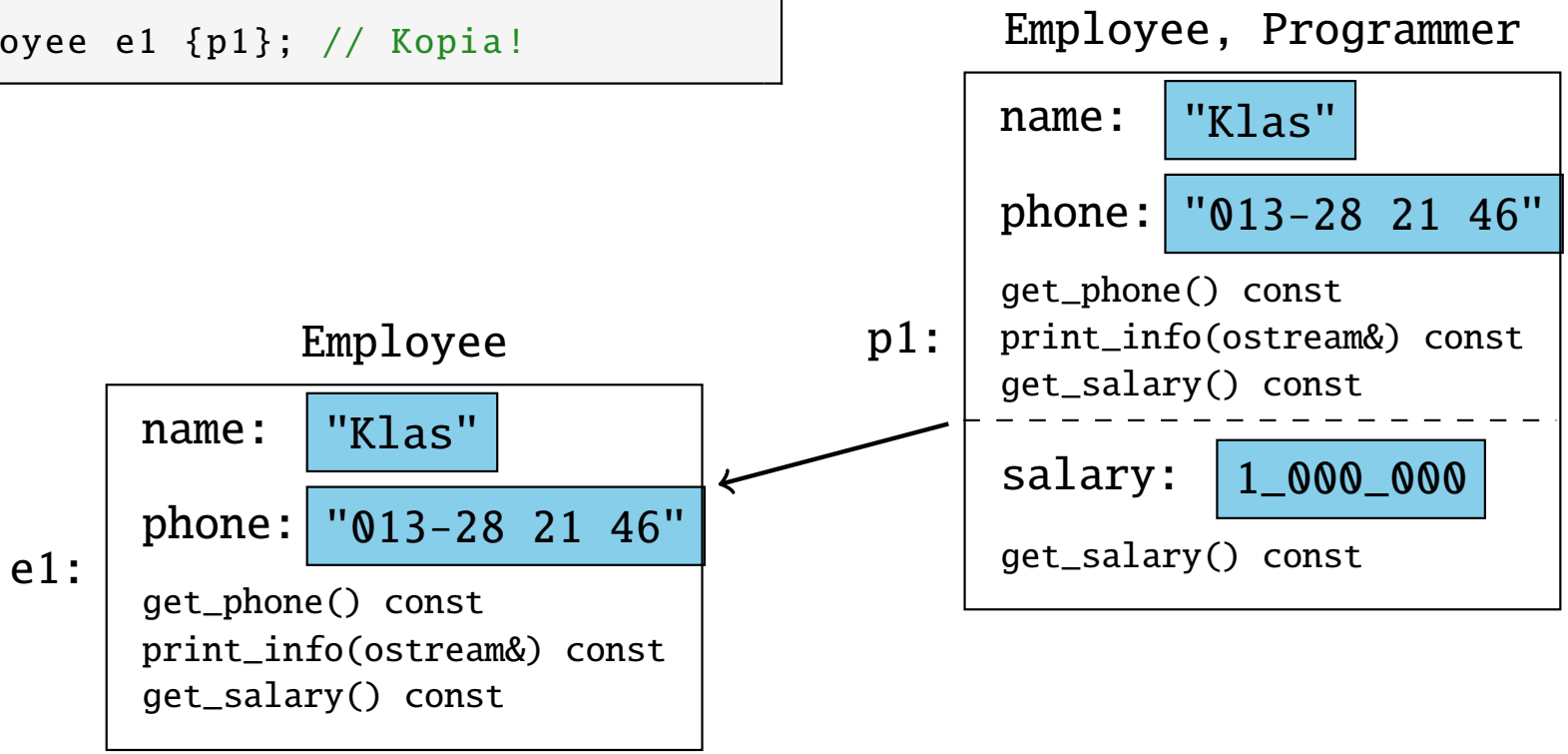
Problem 3: Hur skriver vi nu löneberäkningen?

```
vector< Employee > v{};
v.push_back(Programmer{"Emma", "070- 44 55 66", 20000_kr});
v.push_back(Programmer{"Eric", "070- 11 22 33", 20000_kr});
v.push_back(Programmer{"Klas", "013- 28 21 47", 20000_kr});
v.push_back(Consultant{"Nils", "070- 77 88 99", 800_kr_h, 80_h});

sort( begin(v), end(v), [](Employee const& a, Employee const& b)
      { return a.get_salary() < b.get_salary(); } );

for ( Employee const& e : v)
{
  cout << e.get_salary() << endl;
}
```

SLICING! Både Programmer och Consultant *ÄR* Employees. De har *alla* Employees medlemmar och *dessutom* ytterligare några egna. Dessa får inte plats i minnesutrymmet för en Employee. Kompilatorn sparar bara datamedlemmarna för Emploee i vektorn.

Exempel: Slicing

```
Programmer p1 {"Klas", "013-28 21 46",
               1_000_000_kr};

Employee e1 {p1}; // Kopia!
```



Pekare kan rädda oss!

Vi allokerar programmerare och konsulter i sitt eget minne på heapen och låter vektorn peka ut objekten där de ligger! Ingen kopiering till för litet minnesutrymme behöver ske. (Även referenser låter oss referera till härledda klasser utan att råka ut för slicing, men vektorer kan inte hantera referenser.)

```
vector< Employee* > v{};
v.push_back(new Programmer{"Emma", "070- 44 55 66", 20000_kr});
v.push_back(new Programmer{"Eric", "070- 11 22 33", 20000_kr});
v.push_back(new Consultant{"Nils", "070- 77 88 99", 800_kr_h, 80_h});

sort( begin(v), end(v), [](Employee* const& a, Employee* const& b)
      { return a->get_salary() < b->get_salary(); } );

for ( Employee const* e : v)
{
  cout << e->get_salary() << endl;
}
```

1 Specialisering (Arv)
2 Statisk binding
3 Dynamisk binding - Polymorfi
4 Mer om arv
5 Synlighet
6 Objektoriend analys
7 UML

LINKÖPING UNIVERSITY

Problem 4: Vilken get_salary() anropas?

Standard i C++ är *statisk bindning*. Det betyder att om anropet sker på ett objekt av typen X så kommer kompilatorn att använda den medlemsfunktion som är definierad i klassen X.

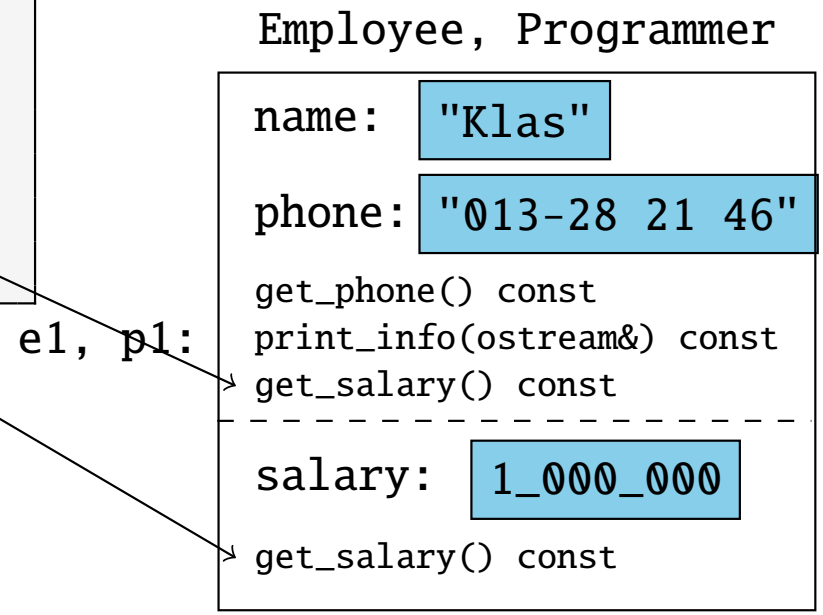
```
for ( Employee const* e : v)
{
  cout << e->get_salary() << endl;
}
```

Vi kommer alltså anropa `Employee::get_salary()`.

Är det verkligen den vi vill anropa?

Exempel: Statisk bindning

```
Programmer p1 {"Klas", "013-28 21 46",
               1_000_000_kr};

Employee& e1 {p1};

cout << e1.get_salary()
     << e1.get_phone()
     << p1.get_salary()
     << endl;
```



1 Specialisering (Arv)
2 Statisk binding
3 Dynamisk binding - Polymorfi
4 Mer om arv
5 Synlighet
6 Objektoriend analys
7 UML

LINKÖPING UNIVERSITY

Dynamisk bindning!

I C++ kan vi välja till *dynamisk bindning*.

Det betyder att om anropet sker på en medlemsfunktion deklarerad *virtual* på ett objekt av baslasstypen B så kommer vi under programkörning att titta om det i minnet för objektet i själva verket finns även ett *härlett* klassobjekt H. I så fall används den medlemsfunktion som är definierad i klassen H.

Detta gör att *ett* anrop kan bete sig mångskiftande - polymorft - eftersom olika implementation anropas beroende på vad som finns i minnet.

Dynamisk bindning!

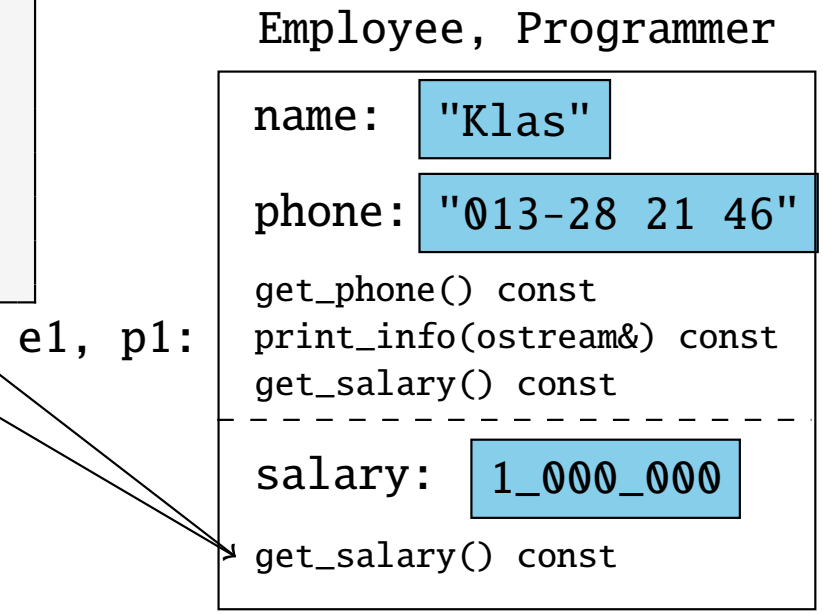
Vi lägger till *virtual* i basklassen och *override* i den härledda.

```
class Employee
{
  ...
  virtual int get_salary() const;
  ...
};
```

```
class Programmer
{
  ...
  int get_salary() const override;
  ...
};
```

Exempel: Dynamisk bindning

```
Programmer p1 {"Klas", "013-28 21 46",
               1_000_000_kr};

Employee& e1 {p1};

cout << e1.get_salary()
     << e1.get_phone()
     << p1.get_salary()
     << endl;
```



Problem 5: Hur skrivs get_salary() i Employee?

Vi kan implementera ett standardbeteende:

```
int Employee::get_salary() const
{
  return 100000_kr; // det saknas datamedlem för lön i Employee
}
```

Eller så kan vi deklarera funktionen *pure virtual* (tom implementation):

```
class Employee
{
  ...
  virtual int get_salary() const = 0;
  ...
};
```

Klassen blir då *abstrakt* och inga instanser tillåts.

Utblick 1: Abstrakt klass

Problem:

- Ibland har en basklass ingen rimlig implementation av en funktion.

```
class Shape
{
public:
  virtual double get_area() const
  {
    // ???
  }
};

class Triangle : public Shape
{
public:
  double get_area() const override
  {
    return base * height / 2;
  }
private:
  double base;
  double height;
};
```

Utblick 1: Abstrakt klass

Lösning:

- Sätt implementationen i basklassen till noll (pure virtual).
- En härledd klass måste implementera funktionen.
- Klassen är nu abstrakt. Det går inte att skapa instanser av en abstrakt klass.

```
class Shape
{
public:
  virtual double get_area() const = 0



};

class Triangle : public Shape
{
public:
  double get_area() const override
  {
    return base * height / 2;
  }
private:
  double base;
  double height;
};
```

Utblick 2: Interface

- En abstrakt klass med endast “pure virtual”-funktioner.
- Ett interface är som att ärva en designspecifikation, eftersom härledda klasser måste implementera alla “pure virtual”-funktioner.
- Samma princip som en abstrakt datatyp.

```
class Stack_Interface
{
public:
  virtual int top() const = 0;
  virtual void pop() = 0;
  virtual void push(int i) = 0;
  virtual int size() const = 0;
  virtual bool is_empty() const = 0;
};
```

1 Specialisering (Arv)
2 Statisk binding
3 Dynamisk binding - Polymorfi
4 Mer om arv
5 Synlighet
6 Objektoriend analys
7 UML

LINKÖPING UNIVERSITY

Arv tillåts i flera nivåer

```
class Person
{
  ...
};

class Employee : public Person
{
  ...
};

class CEO : public Employee
{
  ...
};
```

En CEO är nu både en Employee och en Person.

Anrop av basklassens funktion

Vid behov kan vi anropa basklassens version av en funktion. Om samma funktion finns även i den härledda klassen måste vi använda scope resolution, dvs “namn::”.

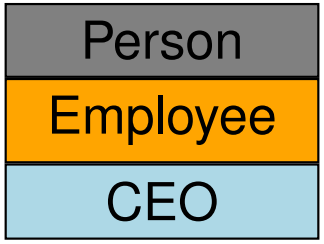
```
class CEO : public Employee
{
public:
  ...
private:
  int bonus;
};

int CEO::get_salary() const()
{
  return Employee::get_salary() + bonus;
}
```

En CEO är nu både en Employee och en Person.

Problem 6: Vad händer när destruktorn körs?

```
Person*  p {new CEO{...}};
Employe* e {new CEO{...}};
CEO*     c {new CEO{...}};

delete p;
delete e;
delete c;
```

```
class Person
{
  ...
  virtual ~Person() = default;
  ...
};
```



Delarna i ett CEO-objekt i minne

Utan virtuell destruktor i basklassen körs destruktorn för pekartypens objekt och dess basklasser.
Med virtuell destruktor i basklassen körs respektive destruktor för alla objektets delar oavsett deklaration.

using och delete

Det går att välja vilka medlemsfunktioner från basklassen som ska finnas i den härledda klassen.

- using - använder en implementation från basklassen.
- delete - tar bort en implementation. Funktionen går inte att anropa från basklassen.

```
class Intern : public Employee
{
  public:
  using Employee::Employee;
  using string Person::get_phone();

  double get_salary() = delete;
};
```

Dynamisk typkontroll

När du behöver komma åt en medlemsfunktion som enbart finns i en viss härledd klass.

Funktionen är olämplig att konvertera till en virtuell funktion i basklassen.

Vi kan kontrollera om en basklasspekare i själva verket pekar på ett objekt av en härledd typ.

```
vector<Employee*> v {
  new Employee{...},
  new Programmer{...},
  new CEO{...},
  new Intern{...}
};

for (Employee* e : v)
{
  CEO* ceo { dynamic_cast<CEO*>(e) };
  if (ceo != nullptr)
  {
    cout << ceo->get_bonus() << endl;
  }
}
```

Exempel

- `std::ostream` är i själva verket basklassen till
  - `std::ostringstream`
  - `std::ofstream`
- `std::istream` är i själva verket basklassen till
  - `std::istringstream`
  - `std::ifstream`
- Möjliggör att återanvända samma kod för olika typer av strömmar.

Läs mer på https://en.cppreference.com/w/cpp/io

1 Specialisering (Arv)
2 Statisk binding
3 Dynamisk binding - Polymorfi
4 Mer om arv
5 Synlighet
6 Objektoriend analys
7 UML

LINKÖPING UNIVERSITY

Synlighet

I en klass kan vi sätta synlighet på medlemmarna.

- public - tillgänglig utanför klassen
- private - endast tillgänglig inom klassen.

Hur blir det med arv?

Ibland vill vi att härledda klasser ska komma åt medlemmar, men ingen utanför klasshierarkin ska komma åt dem.

Synlighet

I en klass kan vi sätta synlighet på medlemmarna.

- public - tillgänglig utanför klassen
- private - endast tillgänglig inom klassen.
- protected - som private men tillgänglig från härledda klasser.

Synlighet

Har du tänkt på varför `public` behövs vid arv?

```
class Employee : public Person
```

Vad händer om vi skriver något annat?

- `public` - medlemmar i basklassen behåller deklarerat skydd i härledd klass.
- `protected` - publika medlemmar blir skyddade i härledd klass.
- `private` - alla medlemmar i basklassen blir privata i härledd klass.

Om inget deklareras väljs privat arv som standard.

1 Specialisering (Arv)
2 Statisk binding
3 Dynamisk binding - Polymorfi
4 Mer om arv
5 Synlighet
6 Objektoriend analys
7 UML

LINKÖPING UNIVERSITY

Objektorienterad analys (OOA)

1. Finn objekten
   (leta substantiv)
2. Klassificera objekten
   CRC (namn, ansvar, samarbeten)
3. Beskriv relationer
   (arv eller association)
4. Identifiera aktörer och användningsfall

Objektorienterad analys (OOA)

Steg 1: Finn objekten

Ett förslag till hur man ska finna objekt är att läsa kraspecifikationen och notera förekommande substantiv.

Objektorienterad analys (OOA)

Steg 2: Klassificera objekten

Varje objekt från steg 1 ska representeras som en klass. Skriv ned klassens:

- namn - Engelska namn, substantiv, singular.
- ansvar - operationer som kan utföras på eller av objeketet.
- samarbetspartners - Vilka andra klasser som klassen behöver samarbeta med för att utföra sina åtaganden.

Objektorienterad analys (OOA)

Steg 3: Beskriver relationer

Bestäm de relationer som finns mellan klasserna:

- arv - x är en specialisering av y
- komposition - x består av y (x finns inte utan y)
- aggregation - x består av y
- association - x känner till y

Objektorienterad analys (OOA)

Steg 4: Aktörer och användningsfall

Ett användningsfall är en interaktion som kan inträffa under exekvering.

- Identifiera en interaktion och beskriv vilka komponenter som är inblandade i att hantera den.
- Syftet är att få en djupare insikt i samarbetet mellan objekt.
- Kan resultera i nya ansvar, klasser eller samarbeten, eller att klasser som inte används tas bort.

1 Specialisering (Arv)
2 Statisk binding
3 Dynamisk binding - Polymorfi
4 Mer om arv
5 Synlighet
6 Objektoriend analys
7 UML

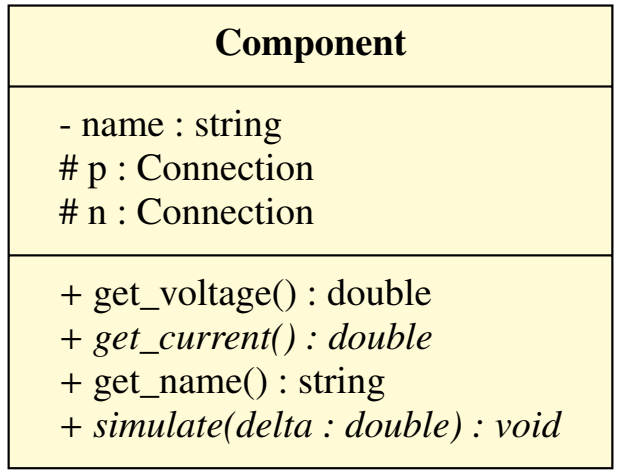
LINKÖPING UNIVERSITY

UML-diagram

- Visuell representation av projektets design
- Oberoende av programmeringsspråk
- Standardiserad representation
- Innehåller:
  - Klasser med alla medlemmar
  - Relationer mellan klasser

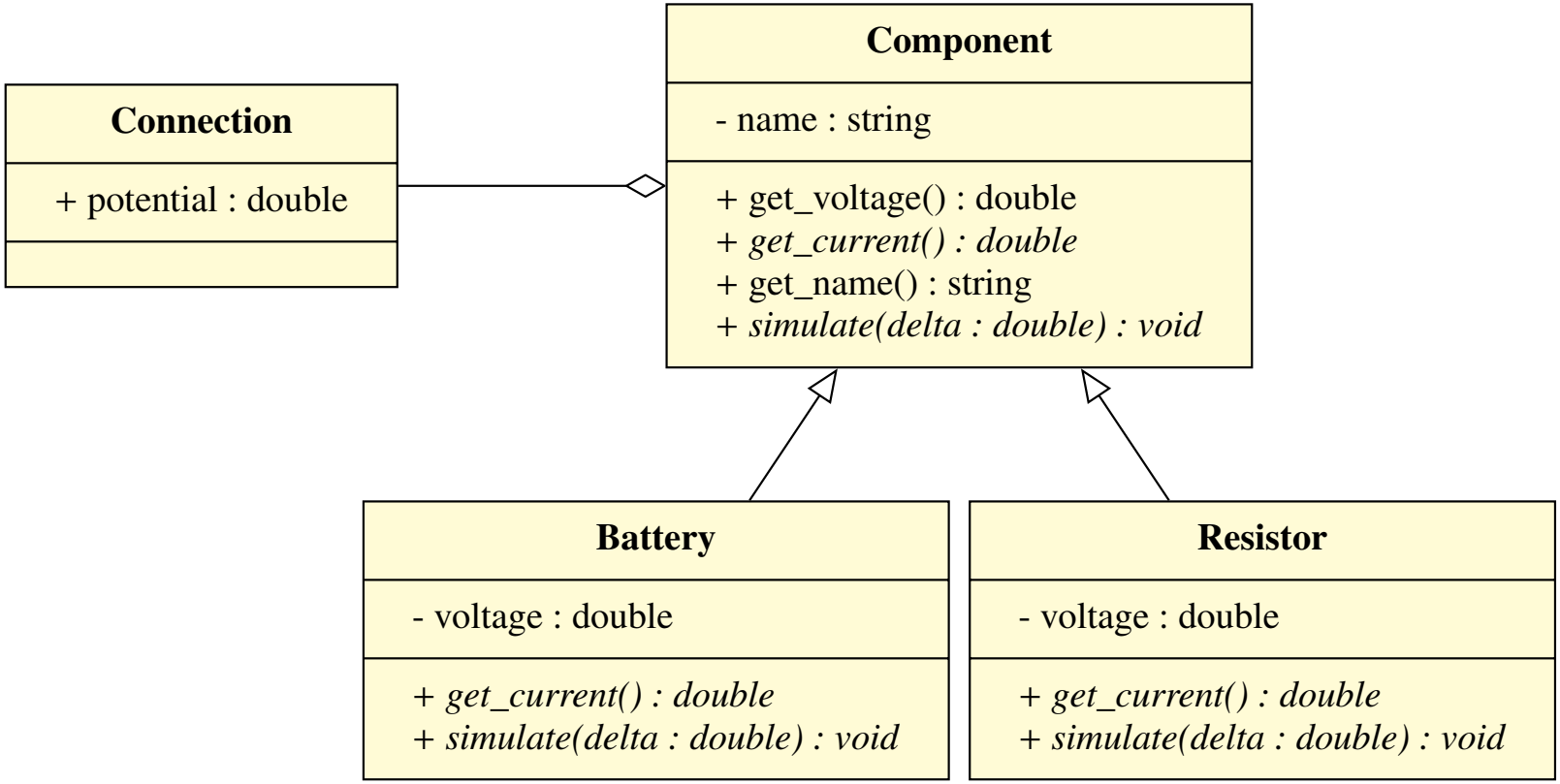
UML-diagram

Hur representeras en klass?

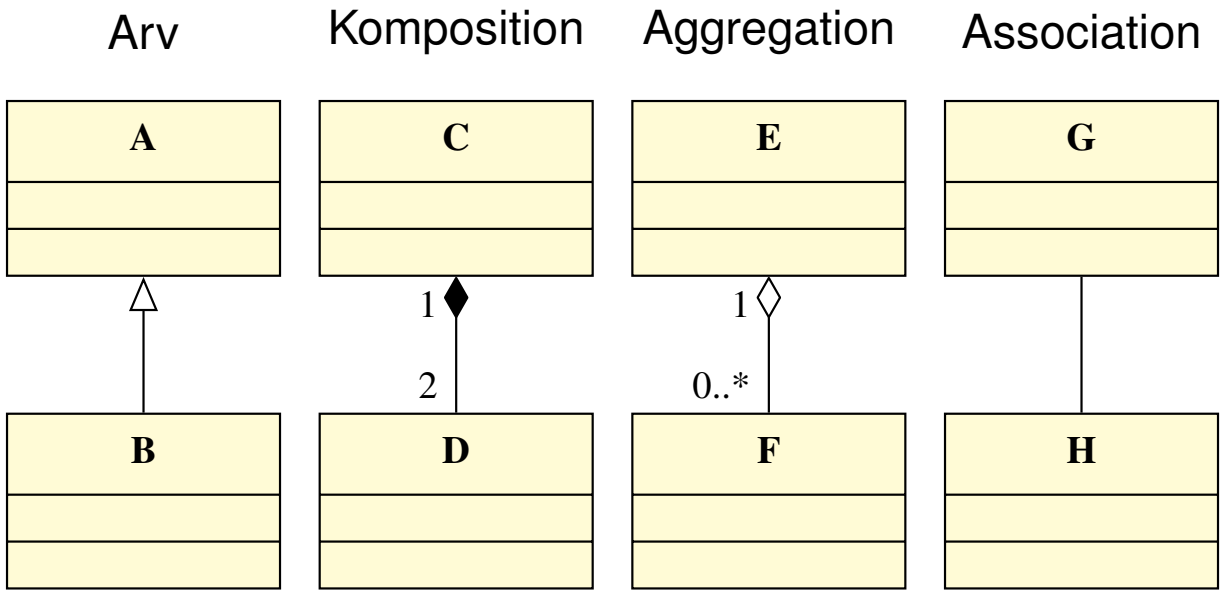


- + public
- – private
- # protected

UML-diagram


Component
- name : string
+ get_voltage() : double
+ get_current() : double
+ get_name() : string
+ simulate(delta : double) : void
Connection
+ potential : double
Battery
- voltage : double
+ get_current() : double
+ simulate(delta : double) : void
Resistor
- voltage : double
+ get_current() : double
+ simulate(delta : double) : void

UML-diagram



“B är en A”
“C består av två D”
“E har av noll eller flera F”
“G känner till H”

www.ida.liu.se/~TDDE71/