TDIU20

Arv, Polymorfi, OOA

Eric Ekström

Institutionen för datavetenskap

LINKÖPING UNIVERSITY

1 Specialisering (Arv)
2 Statisk binding
3 Polymorfi
4 Synlighet
5 Objektoriend analys
6 UML

LINKÖPING UNIVERSITY

Specialisering (Arv)

Förutsättning:

- I vårt program finns en klass (kallad BAS)
- Vi ska skapa nya klasser som har mycket gemensamt med BAS.

Dåliga alternativ:

- Kopiera koden från BAS.
- Lägg till den nya koden direkt i BAS.
- Gör BAS till datamedlem i de nya klasserna.

Specialisering (Arv)

Lösning:

- Skapa de nya klasserna genom att basera dem på BAS.
  - BAS är nu en basklass.
  - Varje ny klass är en härledd klass (subklass).
  - De härledda klasserna “ärver” från BAS.
- Fanns det saker i BAS som inte var gemensamt för alla nya klasser gör vi en härledd klass med detta.

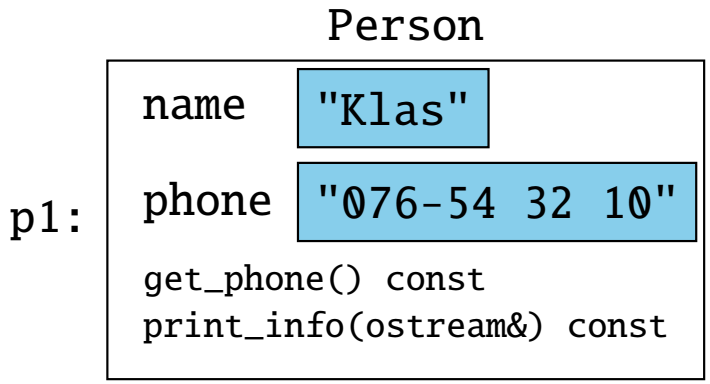
Specialisering (Arv) - Basklass

```
class Person
{
public:

  Person(string const& n,
         string const& p);
  string get_phone() const;
  void print_info(ostream& os) const;

private:
  string name;
  string phone;

};
```

```
Person::Person(string const& n,
               string const& p)
  : name{n}, phone{p}
{}

string Person::get_phone() const
{
  return phone;
}

void Person::print_info(ostream& os) const
{
  os << "Name: " << name << "\n";
  os << "Phone: " << phone << endl;
}
```

Specialisering (Arv) - Basklass

```
int main()
{
  Person p1 {"Klas", "076-54 32 10"};
}
```



Specialisering (Arv) - Härledd klass

```
class Employee : public Person
{
public:

  Employee(string const& n,
           string const& p,
           string const& w,
           int s);
  string get_phone() const;
  string get_home_phone() const;
  string get_work_phone() const;

private:
  std::string work_phone;
  int salary;

};
```

```
Employee::Employee(string const& n,
                   string const& p,
                   string const& w,
                   int s)
  : Person{n, p}, work_phone{w}, salary{s}
{}
string Employee::get_phone() const
{
  return Person::get_phone() + work_phone;
}
string Employee::get_work_phone() const
{
  return work_phone;
}
string Employee::get_home_phone() const
{
  return phone; // Error
}
```

Specialisering (Arv) - Härledd klass

```
class Employee : public Person
{
public:

  Employee(string const& n,
           string const& p,
           string const& w,
           int s);
  string get_phone() const;
  string get_home_phone() const;
  string get_work_phone() const;

private:
  std::string work_phone;
  int salary;

};
```

Vi baserar klassen Employee på klassen Person.

- Employee ärver alla medlemmar från Person.
- Medlemmarna i Person är nu medlemmar i Employee. Synligheten ändras inte.

Employee kan ersätta implementationer av funktioner från Person.

Specialisering (Arv) - Härledd klass

Härledda klassen ska alltid initiera sin basklassdel genom att anropa basklassens konstruktor.

Vi kan göra explicita anrop till basklassens version av en funktion.

Vi kan inte komma åt några av basklassens privata delar.

```
Employee::Employee(string const& n,
                   string const& p,
                   string const& w,
                   int s)
  : Person{n, p}, work_phone{w}, salary{s}
{}
string Employee::get_phone() const
{
  return Person::get_phone() + work_phone;
}
string Employee::get_work_phone() const
{
  return work_phone;
}
string Employee::get_home_phone() const
{
  return phone; // Error
}
```

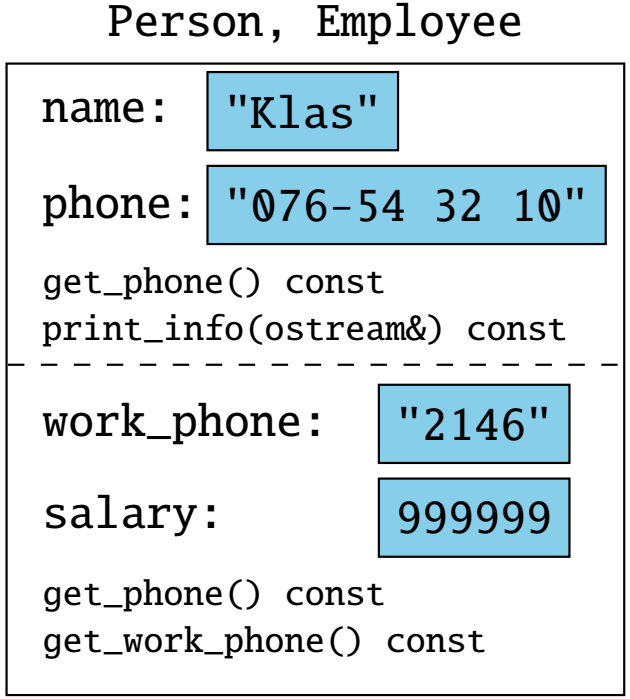
1 Specialisering (Arv)
2 Statisk binding
3 Polymorfi
4 Synlighet
5 Objektoriend analys
6 UML

Statisk bindning

Hur vet kompilatorn vilken funktion som ska anropas?

```
Employee e1 {"Klas", "076-54 32 10",
             "2146", 999999};

Person& p1 {e1};

cout << p1.get_phone()
     << e1.get_phone()
     << e1.get_work_phone()
     << endl;
e1.print_info(cout);
```

e1, p1:

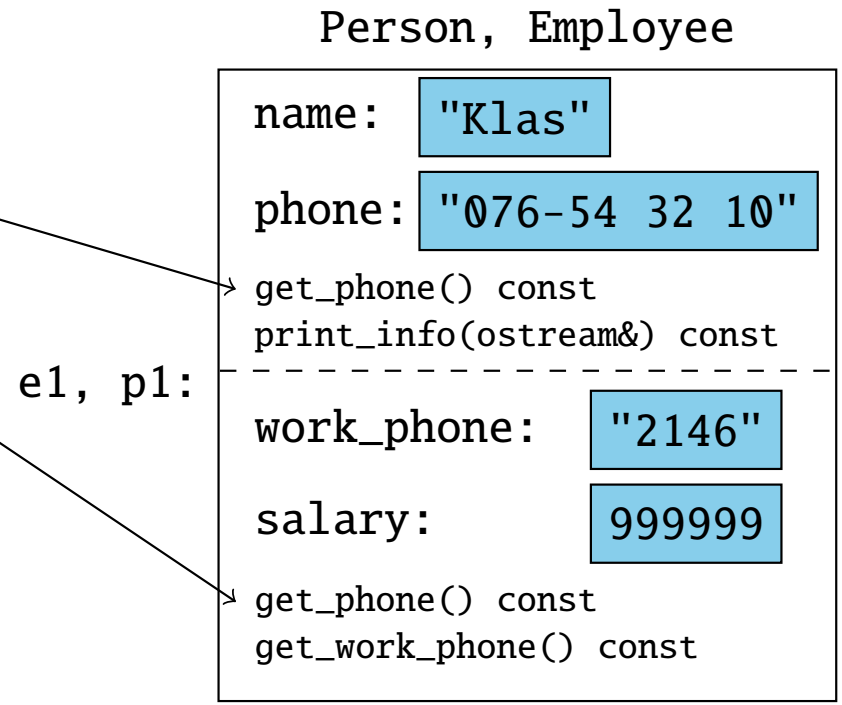


Statisk bindning

- Vilken funktion som anropas avgörs vid kompilering genom att titta på objektets deklaration i koden.
- Det spelar ingen roll vilken typ av objekt som faktiskt ligger i minnet.
- Standard i C++ om inget annat anges.

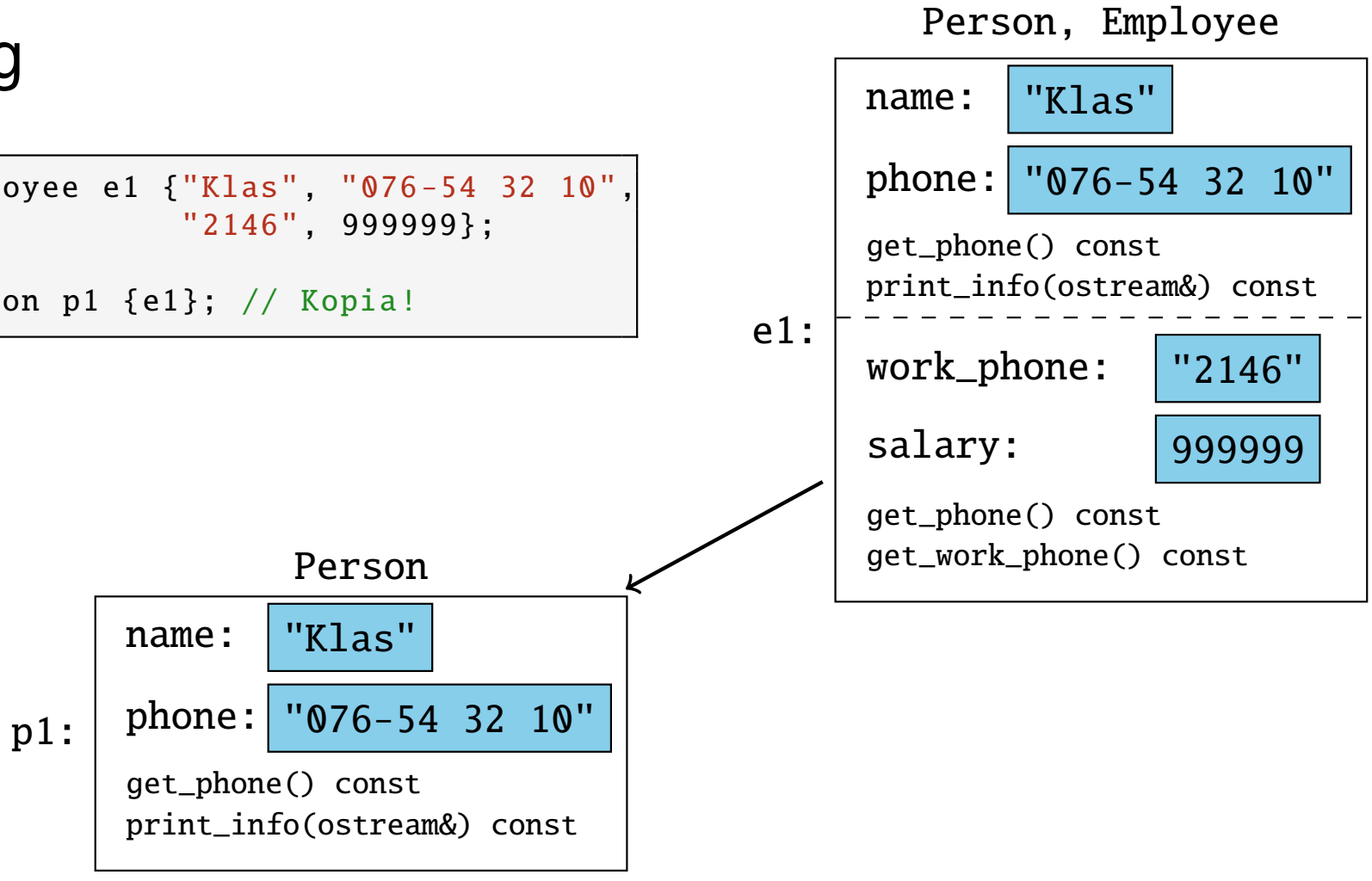
Statisk bindning

```
Employee e1 {"Klas", "076-54 32 10",
             "2146", 999999};

Person& p1 {e1};

cout << p1.get_phone()
     << e1.get_work_phone()
     << e1.get_phone()
     << endl;
e1.print_info(cout);
```



Slicing

```
Employee e1 {"Klas", "076-54 32 10",
             "2146", 999999};

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



1 Specialisering (Arv)
2 Statisk binding
3 Polymorfi
4 Synlighet
5 Objektoriend analys
6 UML

LINKÖPING UNIVERSITY

Härledda klasser

```
class Employee : public Person
{
public:
  Employee(int s)
    : salary {s}
  {}

  int get_salary() const
  {
    return salary;
  }

private:
  int salary;
};
```

```
class Programmer : public Employee
{
public:
  Programmer(int s, double exp)
    : Employee{s}, expertise {exp}
  {}

  int get_salary() const
  {
    return salary * expertise;
  }

private:
  int expertise;
};
```

```
class CEO : public Employee
{
public:
  CEO(int s, int b)
    : Employee{s}, bonus{b}
  {}

  int get_salary() const
  {
    return salary + bonus;
  }

private:
  int bonus;
};
```

Härledda klasser

Vilken `get_salary()` kommer anropas?
(Kom ihåg: statisk bindning!)

```
vector<Employee*> v {};

v.push_back(new Employee{...});
v.push_back(new Programmer{...});
v.push_back(new CEO{...});

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

Polymorfi - problemet

Vi vill att varje härledd klass ska ha sin egen specializerade implementation av en funktion.

Vi vill att den härledda klassens implementation anropas automatiskt istället för basklassens.

Polymorfi - problemet

Vi vill att varje härledd klass ska ha sin egen specializerade implementation av en funktion.

Vi vill att den härledda klassens implementation anropas automatiskt istället för basklassens.

Lösning: Slå på dynamisk bindning!

- Deklarera funktionen som `virtual` i basklassen.
- Deklarera funktionen som `override` i härledda klassen.

Dynamisk bindning

Vid anrop kontrolleras vilken typ av objekt som faktiskt finns i minnet. Nyckelordet virtual ger dynamisk bindning för just den funktionen.

```
class Employee : public Person
{
public:
  Employee(int s)
  : salary {s}
  {}
  virtual ~Employee() = default;

  virtual int get_salary() const
  {
    return salary;
  }
};
```

```
class Programmer : public Employee
{
public:
  Programmer(int s, double exp)
    : Employee{s}, expertise {exp}
  {}

  int get_salary() const override
  {
    return salary * expertise;
  }
};
```

Härledda klasser

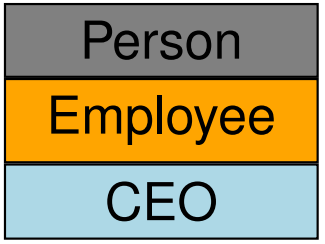
Vilken `get_salary()` kommer anropas?
(Nu med dynamisk bindning!)

```
vector<Employee*> v {};

v.push_back(new Employee{...});
v.push_back(new Programmer{...});
v.push_back(new CEO{...});

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

Virtuell destruktor

```
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.

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;
};
```

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;
};
```

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;
};
```

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 Polymorfi
4 Synlighet
5 Objektoriend analys
6 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

```
class Person
{
public:

  Person(string const& n,
         string const& p);
  string get_phone() const;
  void print_info(ostream& os) const;

protected:
  string name;
  string phone;

};
```

```
class Employee : public Person
{
public:

  Employee(string const& n,
           string const& p,
           string const& w,
           int s);

  string get_phone() const
  {
    return phone + work_phone;
  }

private:
  string work_phone;
  int salary;

};
```

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 Polymorfi
4 Synlighet
5 Objektoriend analys
6 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 Polymorfi
4 Synlighet
5 Objektoriend analys
6 UML

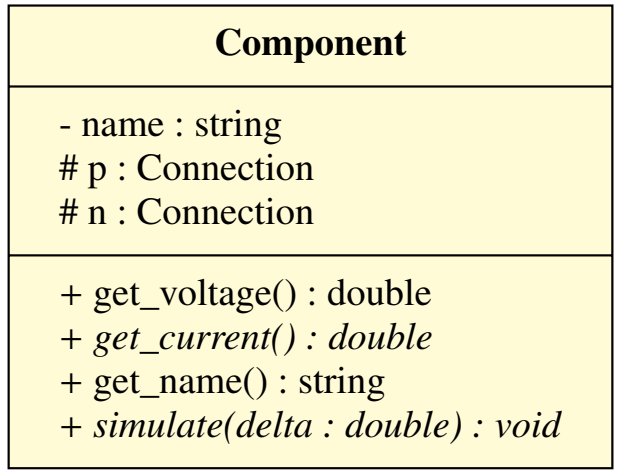
LINKÖPING UNIVERSITY

UML-diagram

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

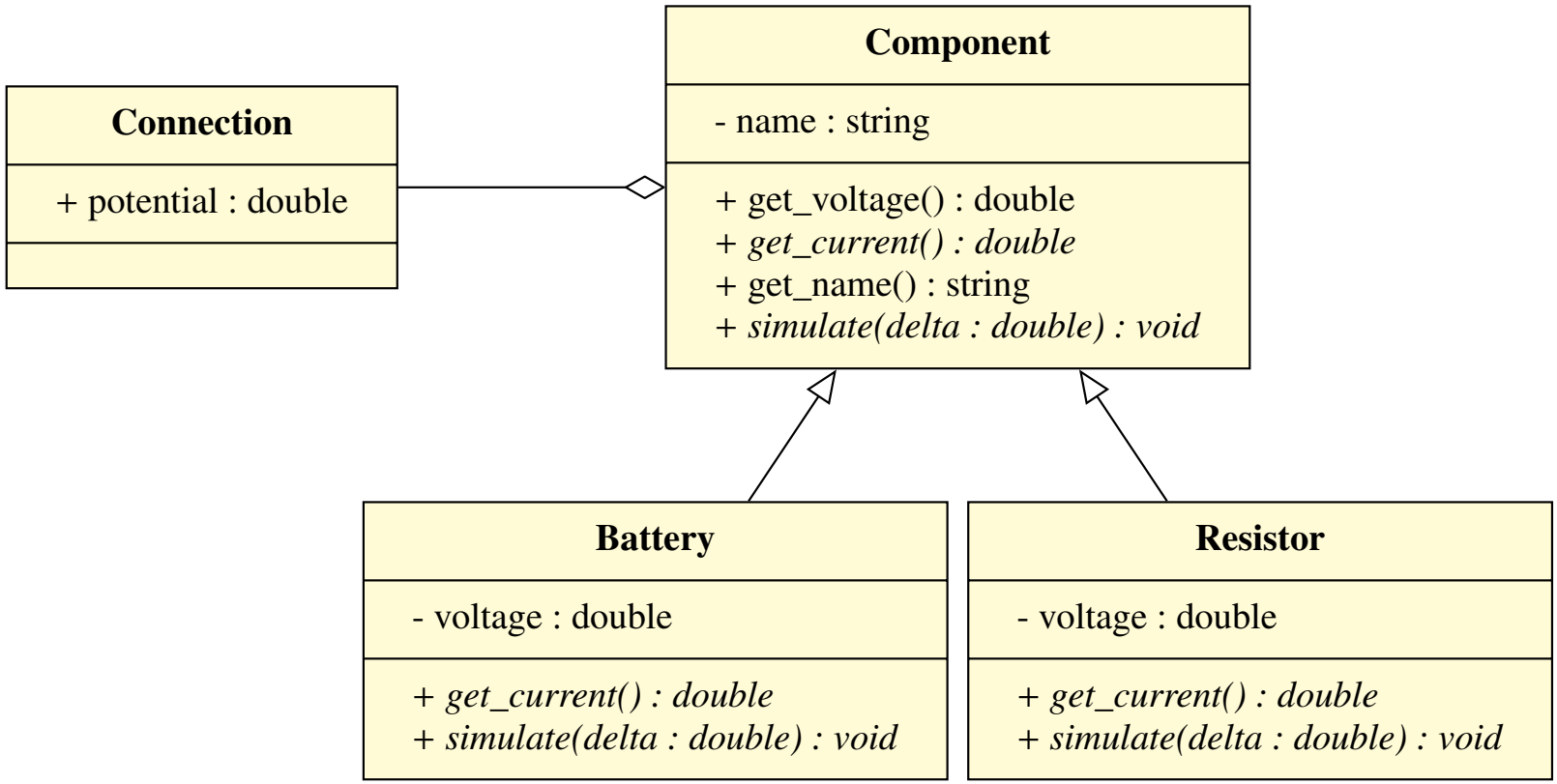
UML-diagram

Hur representeras en klass?

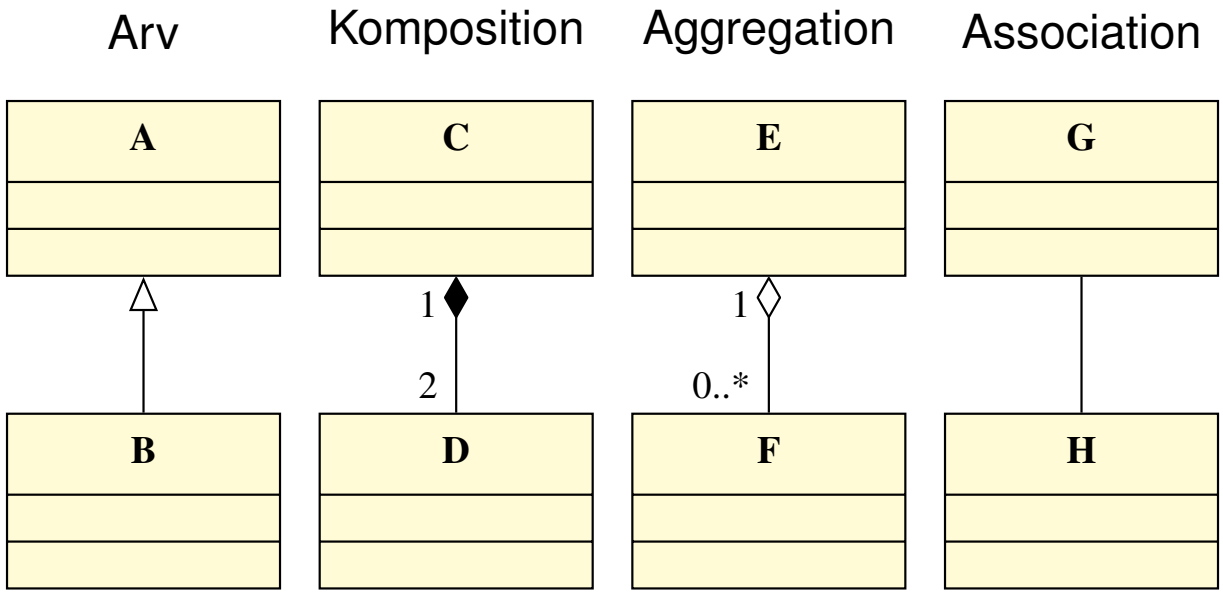


- + public
- – private
- # protected

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.liu.se

LINKÖPING UNIVERSITY