TDDC76 – Programmering och datastrukturer

Arv, polymorfi, OOA

Eric Ekström, Klas Arvidsson, 2022
Institutionen för datavetenskap

LINKÖPINGS UNIVERSITET

Agenda

1. Specialisering (Arv)
2. Statisk bindning
3. Polymorfi
4. Synlighet
5. Objektorienterad analys och design (OOA, OOD)
6. UML
7. Arv - ett stort exempel

LINKÖPINGS UNIVERSITET

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 i BAS
- Gör BAS till datamedlem i de nya klasserna

Specialisering (Arv)

Lösning:

- Skapa de nya klasserna genom att basera dem på BAS (Arv i C++).
  - BAS är nu en basklass
  - Varje ny klass är en härledd klass (subklass)
- Fanns saker i BAS som inte var gemensamt för alla nya klasser gör vi härledd klass med detta.

Exempel: Basklass

Person.h

```
class Person
{
public:
  Person(std::string const& n,
         std::string const& p);
  std::string get_phone() const;
  void print_info(ostream& os) const;

private:
  std::string name;
  std::string phone;

};
```

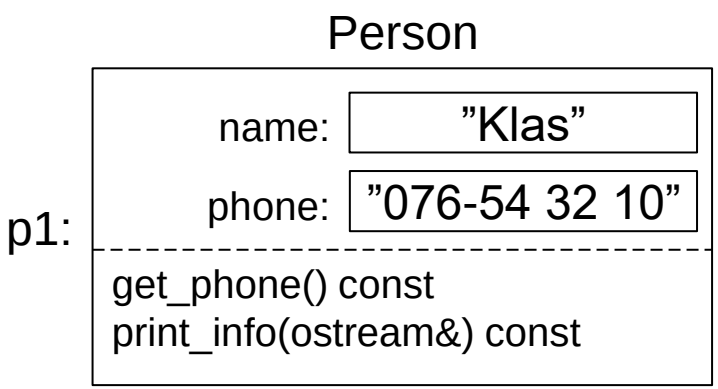
Person.cc

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

std::string Person::get_phone() const
{
   return phone;
}
void Person::print_info(ostream& os)
const
{
  os << ”Name: ” << name << ”\n”;
  os << ”Phone: ” << phone << endl;
}
```

Exempel: Basklass

Person.h

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



Exempel: Härledd klass

Employee.h

```
class Employee : public Person
{
public:
  Employee(std::string const& n,
           std::string const& p,
           std::string const& w,
           int s);

  std::string get_phone() const;
  std::string get_home_phone() const;
  std::string get_work_phone() const;

private:
  std::string work_phone;
  int salary;

};
```

Employee.cc

```
Employee(std::string const& n,
         std::string const& p,
         std::string const& w,
         int s);
  : Person{n, p}, work_phone{w}, salary{s}
{}

std::string Employee::get_phone() const
{
   return Person::get_phone() + ", " +
work_phone;
}
std::string Employee::get_work_phone() const
{
   return work_phone;
}
std::string Employee::get_home_phone() const
{
   return phone; // Error!
}
```

Exempel: Härledd klass

Employee.h

```
class Employee : public Person
{
public:
  Employee(std::string const& n,
           std::string const& p,
           std::string const& w,
           int s);

  std::string get_phone() const;
  std::string get_home_phone() const;
  std::string get_work_phone() const;

private:
  std::string work_phone;
  int salary;

};
```

Vi **baserar** klassen Employee på klassen Person.

Ska välja basklassens **medlemmars skyddsnivå** i den härledda klassen:

- **public**: Ingen ändring från basklassens nivåer. (Det vi oftast vill ha)
- **protected**: Publika medlemmar i basklassen blir skyddade i den härledda klassen.
- **private**: publika och skyddade medlemmar i basklassen blir privata i den härledda klassen.

Exempel: Härledd klass

Ska alltid initiera sin basklassdel genom att anropa basklassens konstruktor i **datamedlemsinitieringslistan**

Kan göra egna versioner av funktioner från basklassen.

Kan göra **explicita** anrop till basklassens version av en funktion

Kan inte **komma åt** några av basklassens provata delar

Employee.cc

```
Employee(std::string const& n,
           std::string const& p,
           std::string const& w,
           int s);
  : Person{n, p}, work_phone{w}, salary{s}
{}

std::string Employee::get_phone() const
{
   return Person::get_phone() + ", " +
work_phone;
}
std::string Employee::get_work_phone() const
{
   return work_phone;
}
std::string Employee::get_home_phone() const
{
   return phone; // Error!
}
```

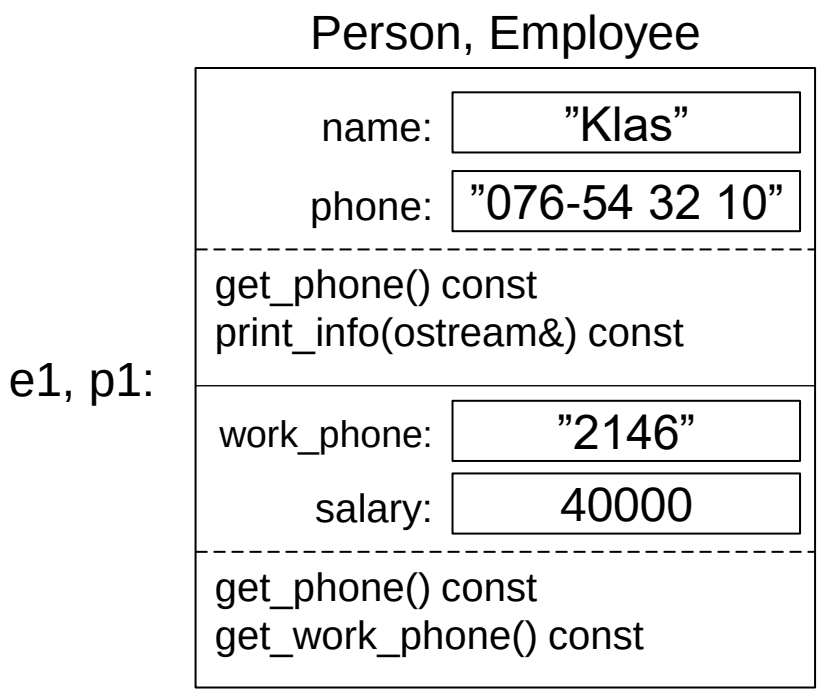
Agenda

1. Specialisering (Arv)
2. Statisk bindning
3. Polymorfi
4. Synlighet
5. Objektorienterad analys och design (OOA, OOD)
6. UML
7. Arv - ett stort exempel

LINKÖPINGS UNIVERSITET

Exempel: Statisk bindning

```
int main()
{
  Employee e1 {”Klas”, ”076-54 32 10”,
               ”2146”, 40000};

  Person& p1 {e1};

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

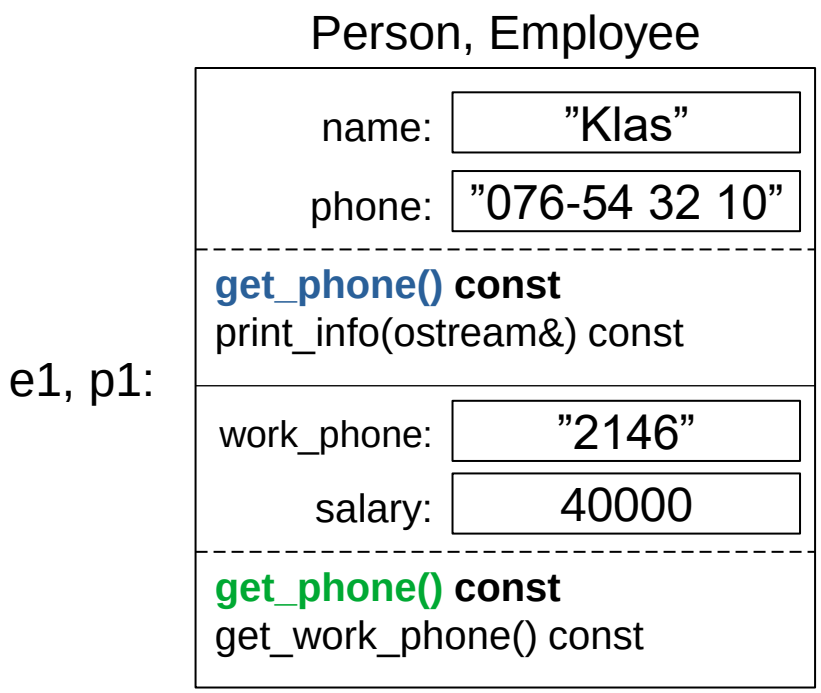


Statisk binding

- Vilken funktion som anroppas avgörs vid kompileringen genom att titta på objektets deklaration i koden.
- Det spelar ingen roll vilken typ av objekt som faktiskt ligger i minne.

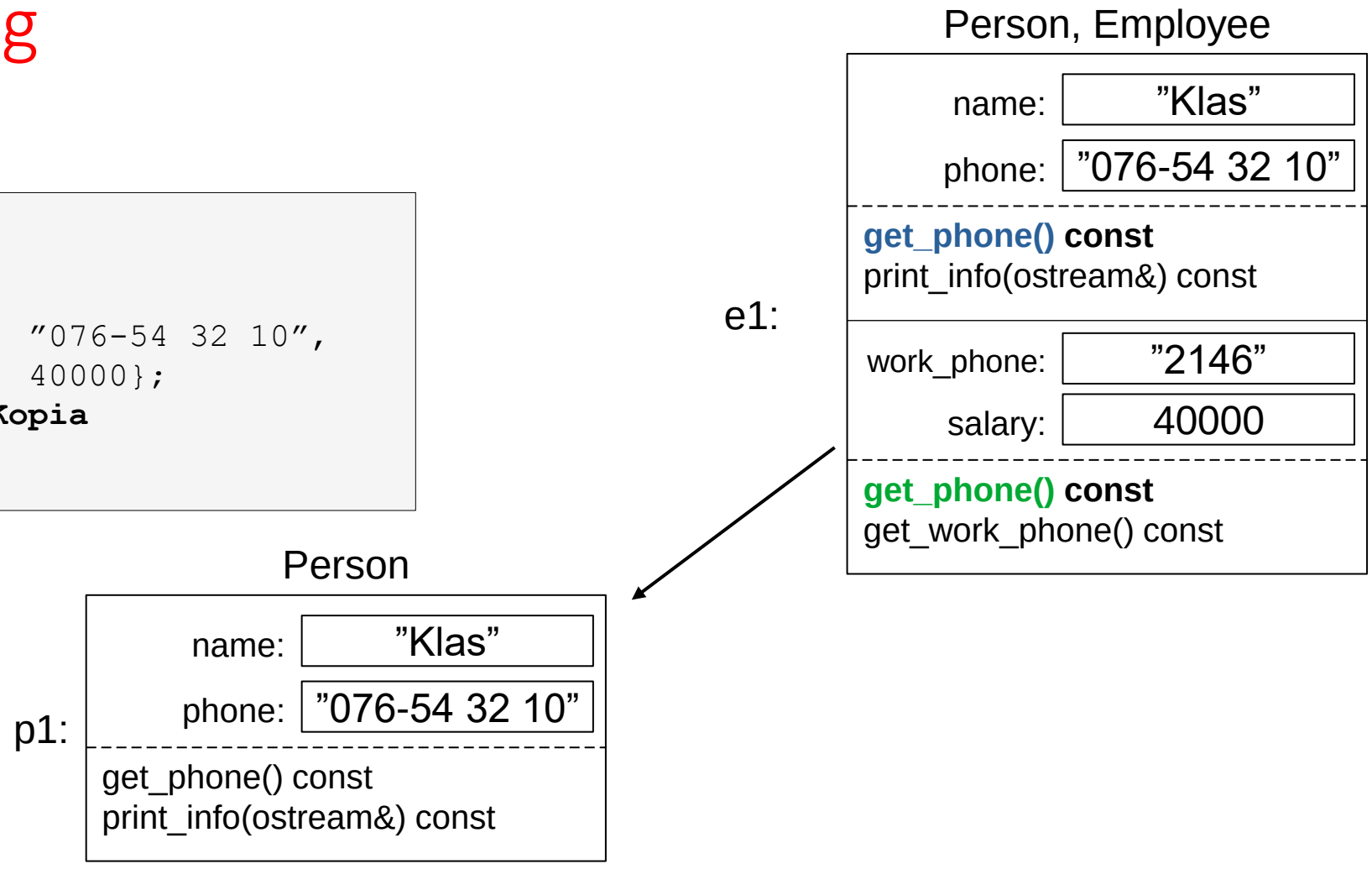
Exempel: Statisk bindning

```
int main()
{
  Employee e1 {”Klas”, ”076-54 32 10”,
               ”2146”, 40000};

  Person& p1 {e1};

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



Exempel: Slicing

```
int main()
{
  Employee e1 {”Klas”, ”076-54 32 10”,
               ”2146”, 40000};
  Person p1 {e1}; // Kopia
}
```



Agenda

1. Specialisering (Arv)
2. Statisk bindning
3. Polymorfi
4. Synlighet
5. Objektorienterad analys och design (OOA, OOD)
6. UML
7. Arv - ett stort exempel

LINKÖPINGS UNIVERSITET

Polymorfi

Vi vill kunna säga åt kompilatorn: ”Ta reda på vad som faktiskt gäller när funktionen körs”

- Statisk bindning
  - Vad typen är deklarerad som vid kompilering
- Dynamisk bindning
  - Vad typen faktiskt pekar eller refererar till när programmet körs

Exempel: Härledda klasser

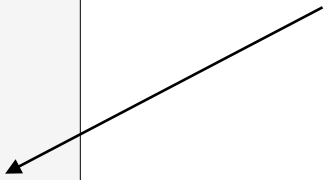
```
class Employee : public Person
{
  Employee(int h)
    : hours {h}
  {
  }

  int get_salary() const
  {
    return salary;
  }

private:
  int salary;

}
```

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

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

private:
  int expertise;

}
```

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

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

private:
  int bonus;

}
```

Exempel: Härledda klasser

```
int main()
{
  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;
  }
}
```



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

Polymorfi

Problem:

- Vi vill att varje härledd klass ska ha sin egen specialiserade implementation av en funktion.
- Vi vill att den härledda klassens implementation anropas automatiskt istället för basklassens implementation.

Lösning: Slå på dynamisk bindning!

- Deklarera funktionen som **virtual** i basklassen.
- Deklarera funktionen som **override** i härledd klass.

Dynamisk bindning

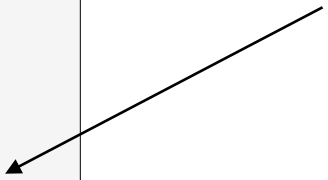
Nyckelordet **virtual** ger dynamisk bindning för just den funktionen. Vid anrop kontrolleras vilken typ av objekt som faktiskt finns i minnet.

```
class Employee : public Person
{
  Employee(int h, int hp)
    : hours {h}, hours_pay{hp}
  {}
  virtual ~Employee() = default;

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

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

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

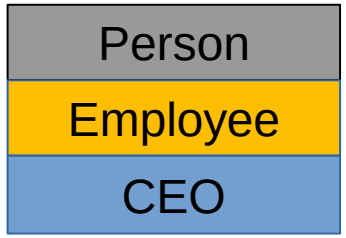
Exempel: Härledda klasser

```
int main()
{
  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;
  }
}
```



Vilken get_salary() kommer anropas?
(med dynamisk bindning)

Virtual destruktor



Delarna i ett CEO-objekt i minnet

```
int main()
{
  Person* p {new CEO{…} };
  Employee* e {new CEO{…} };
  CEO* c { new CEO{…} };

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

Utan virtual destruktor i basklassen Person körs destruktorn för pekartypens objekt och dess basklasser enligt ovan.

Med virtual destruktor i basklassen körs respektive destruktor för alla objektets beståndsdelar oavsett deklaration.

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

Abstrakt klass

Problem:

- Ibland har en basklass ingen rimlig implementation av en funtion som ska överlagras.

```
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;
}
```

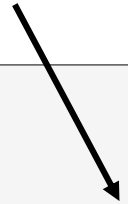
Abstrakt klass

Problem:

- Ibland har en basklass ingen rimlig implementation av en funtion som ska överlagras.

Lösning:

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



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


}

class Triangle
{
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.

```
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 härledda klassen.

```
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 virtual-funktion i basklassen.
- Vi kan kontrollera om en basklasspekare i själva verket pekar på ett objekt av härledd typ.

```
int main()
{
  vector<Employee*> v;

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

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

Exempel: Klasshierarki för IO

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

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

Agenda

3 Polymorfi – std::exception

Exempel: Klasshierarki för standard undantag

- std::exception basklassen till
  - std::logic_error
    - Ytterligare 5 undantag härledda från std::logic_error
  - std::runtime_error
    - Ytterligare 9 undantag härledda från std::runtime_error
  - Ytterligare 9 standardundantag härledda från std::exception
- Möjliggör en alla härledda undantag kan fångas och hanteras gemensamt genom att fånga en ***referens*** till deras basklass.
- Möjliggör att skapa egna specialiserade hierarkier av undantag.

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

Exempel: basklassen std::exception

```
class exception
{
public:
  exception() noexcept;
  exception(const exception&) noexcept;
  exception& operator=(const exception&) noexcept;
  virtual ~exception();
  virtual const char* what() const noexcept;
};
```

Knepig att härleda från pga krav på att nya undantag inte får genereras av implementationen och livstiden på returvärdet från what() är begränsad.

Exempel: eget undantag

```
class List_Empty_Error : public std::logic_error
{
public:
  using logic_error::logic_error;
};
```

Återanvänd std::logic_error eller std::runtme_error. De har till större delen löst de knepiga bitarna!

Exempel: try-catch

```
{
  ... // Code before try-block

  try
  {
    f(); // Potential exception-generating code

    ...  // Remainder of try-block
  }
  catch(std::runtime_error& e) { ... } // Handlers
  catch(std::logic_error&   e) { ... } // (in order of
  catch(std::exception&     e) { ... } //  most derived)

  ... // Code after try-catch
}
```

Exempel: flera catch?

```
try
{
  f(); // Assume a std::logic_error appear here
}
catch(std::runtime_error& e) { /* RUNTIME */ }
catch(std::exception&     e) { /* BASE    */ }
catch(std::logic_error&   e) { /* LOGIC   */ }
```

Vilken undantagshanterare aktiveras?

Agenda

1. Specialisering (Arv)
2. Statisk bindning
3. Polymorfi
4. Synlighet
5. Objektorienterad analys och design (OOA, OOD)
6. UML
7. Arv - ett stort exempel

LINKÖPINGS UNIVERSITET

Synlighet

I en klass kan vi sätta synlighet på medlemmarna

- **public** = möjligt att komma åt utanför klassen
- **private** = går inte att komma åt utanför klassen

- Hur blir det med arv?

Synlighet

I en klass kan vi sätta synlighet på medlemmarna

- **public** = möjligt att komma åt utanför klassen
- **private** = går inte att komma åt utanför klassen

- Hur blir det med arv?
  - Ibland vill vi att härledda klasser ska komma åt medlemmar, fast ingen utanför klass-familjen ska komma åt dem

Synlighet

I en klass kan vi sätta synlighet på medlemmarna

- **public** = möjligt att komma åt utanför klassen
- **private** = går inte att komma åt utanför klassen
- **protected** = som private men går att komma åt i härledda klasser

Synlighet

Person.h

```
class Person
{
public:
  Person(std::string const& n,
         std::string const& p);
  std::string get_phone() const;
  void print_info(ostream& os) const;

protected:
  std::string name;
  std::string phone;
};



//
```

Employee.h

```
class Employee : public Person
{
public:
  Employee(std::string const& n,
           std::string const& p,
           std::string const& w,
           int s);

  std::string get_phone() const
  {
    return phone + ", " + work_phone;
  }

private:
  std::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 basklasen behåller deklarerat skydd i härledd klass
  - **protected**: public medlemmar i basklassen blir protected i härledd klass
  - **private:** alla medlemmar i basklassen blir private i härledd klass
- Om inget deklareras väljs private arv som standard!

Agenda

1. Specialisering (Arv)
2. Statisk bindning
3. Polymorfi
4. Synlighet
5. Objektorienterad analys och design (OOA, OOD)
6. UML
7. Arv - ett stort exempel

LINKÖPINGS UNIVERSITET

Objektorienterad analys

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

Objektorienterad analys

Steg 1 Finn objekt

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

Objektorienterad analys

Steg 2 Klass, ansvar, samarbete

Varje objekt(substantiv) från teg 1 behöver en klass, skriv ned klassens:

- namn – välj namn med omsorg! Välj engelska namn som substantiv i singularis.
- ansvar – operationer(verb) som kan utföras på eller av objekt av klassen ifråga.
- samarbetspartners – vilka andra klasser som klassen samarbetar med för att utföra sina åtaganden. Fråga dig: ”varifrån får objektet denna information?”, ”till vem ska objektet leverera denna information?”

Objektorienterad analys

Steg 3 Relationer

Bestäm de relationer som finns mellan klasserna i systemet:

- arv (x är en specialisering av y)
- komposition (x består av komponenten y, x finns inte utan y)
- aggregation (x består av komponenten y)
- association (x kan använda en y)

Objektorienterad analys

Steg 4

Ett användningsfall är en interaktion som kan inträffa under systemets exekvering. Syftet med att identifiera aktörer och användningsfall och utföra olika scenarier för användningsfall är att få en djupare insikt om samarbetet mellan objekt. Under denna aktivitet kan man till exempel upptäcka nya aktörer och användningsfall, nya objekt/klasser, nya samarbetspartners och nya relationer mellan klasser och objekt. Omvänt gäller att varje objekt/klass/ansvar/samarbete ska ingå i något användningsfall för att dess existens ska vara motiverad.

Objektorienterad design

Steg 5 - Förfining av analysen

- Design i betydelse konstruktionsritning
- Analysresultatet förfinas med konkret information om datamedlemmar och medlemsfunktioner, inklusive rätt datatyper och parmeterlistor
- Ska innehålla allt som behövs för en komplett headerfil (vad som ska finnas i klassen)

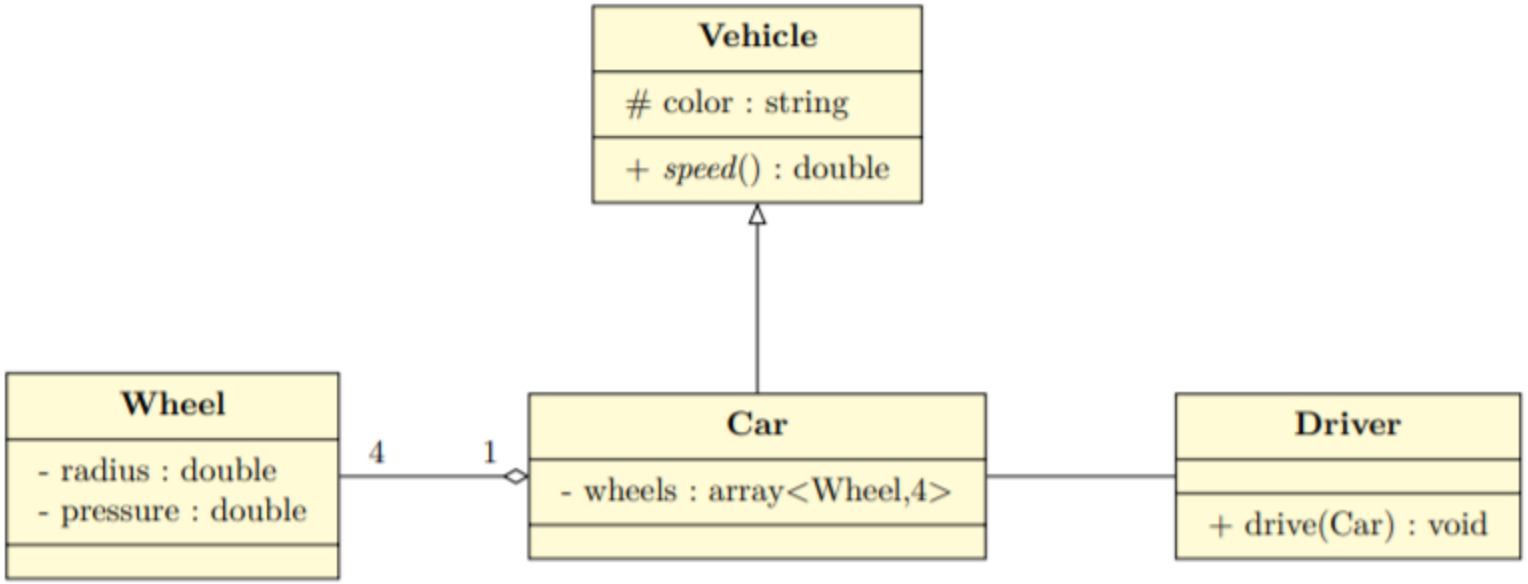
Agenda

1. Specialisering (Arv)
2. Statisk bindning
3. Polymorfi
4. Synlighet
5. Objektorienterad analys och design (OOA, OOD)
6. UML
7. Arv - ett stort exempel

LINKÖPINGS UNIVERSITET

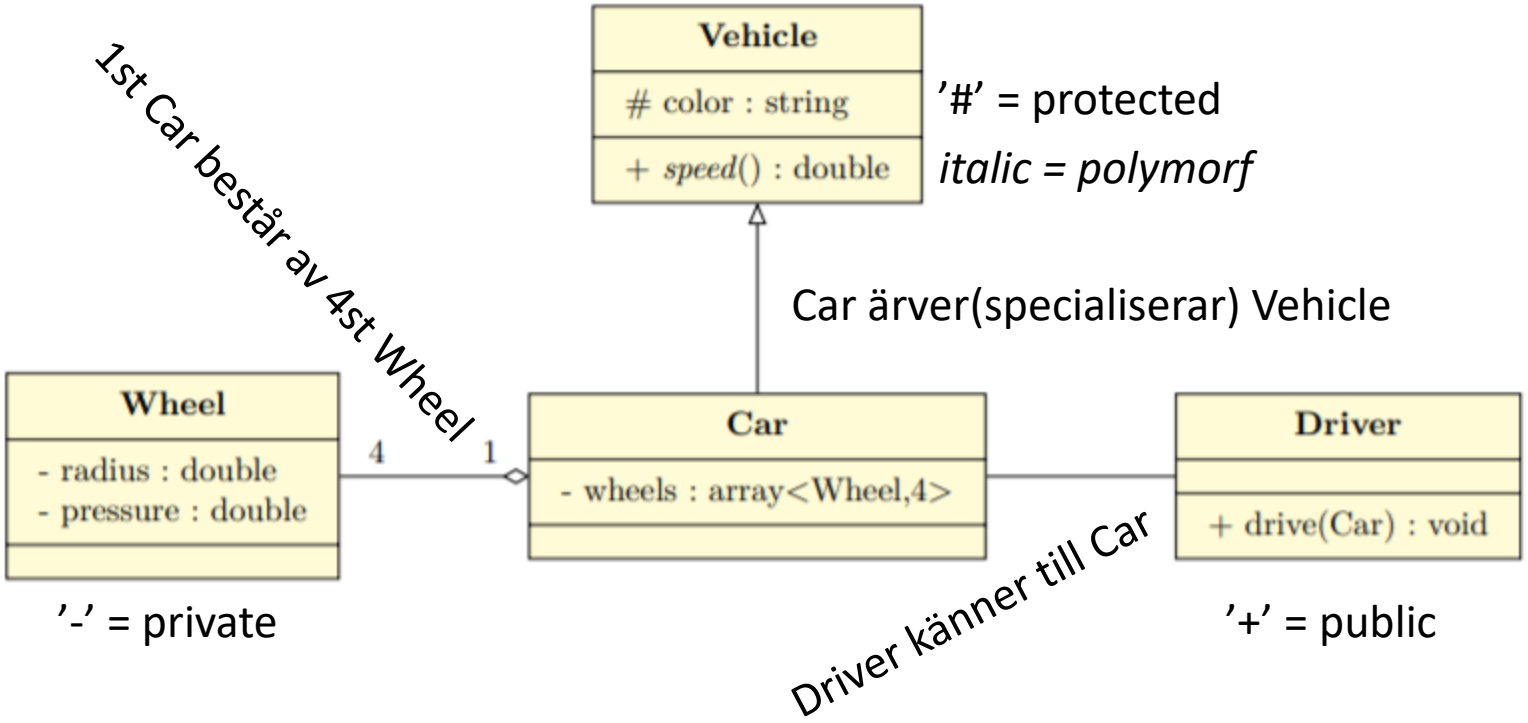
UML Klassdiagram

Resultat av objektorienterad design



UML Klassdiagram

Resultat av objektorienterad design



Agenda

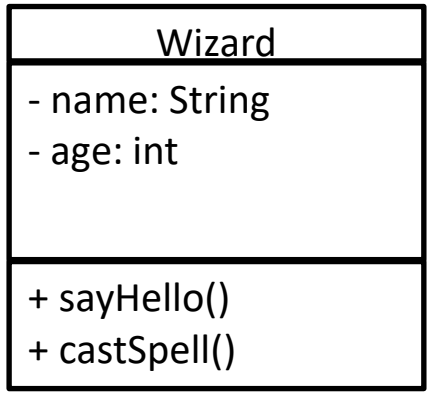
1. Specialisering (Arv)
2. Statisk bindning
3. Polymorfi
4. Synlighet
5. Objektorienterad analys och design (OOA, OOD)
6. UML
7. Arv - ett stort exempel

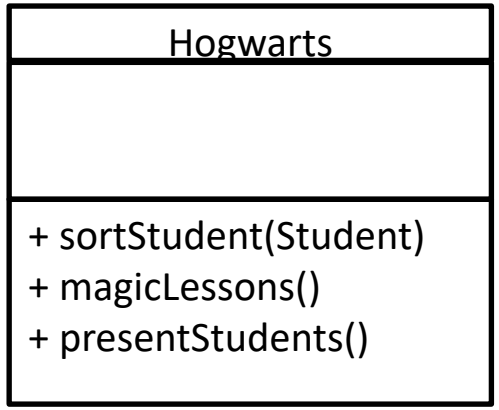
LINKÖPINGS UNIVERSITET

Arv – ett stort exempel

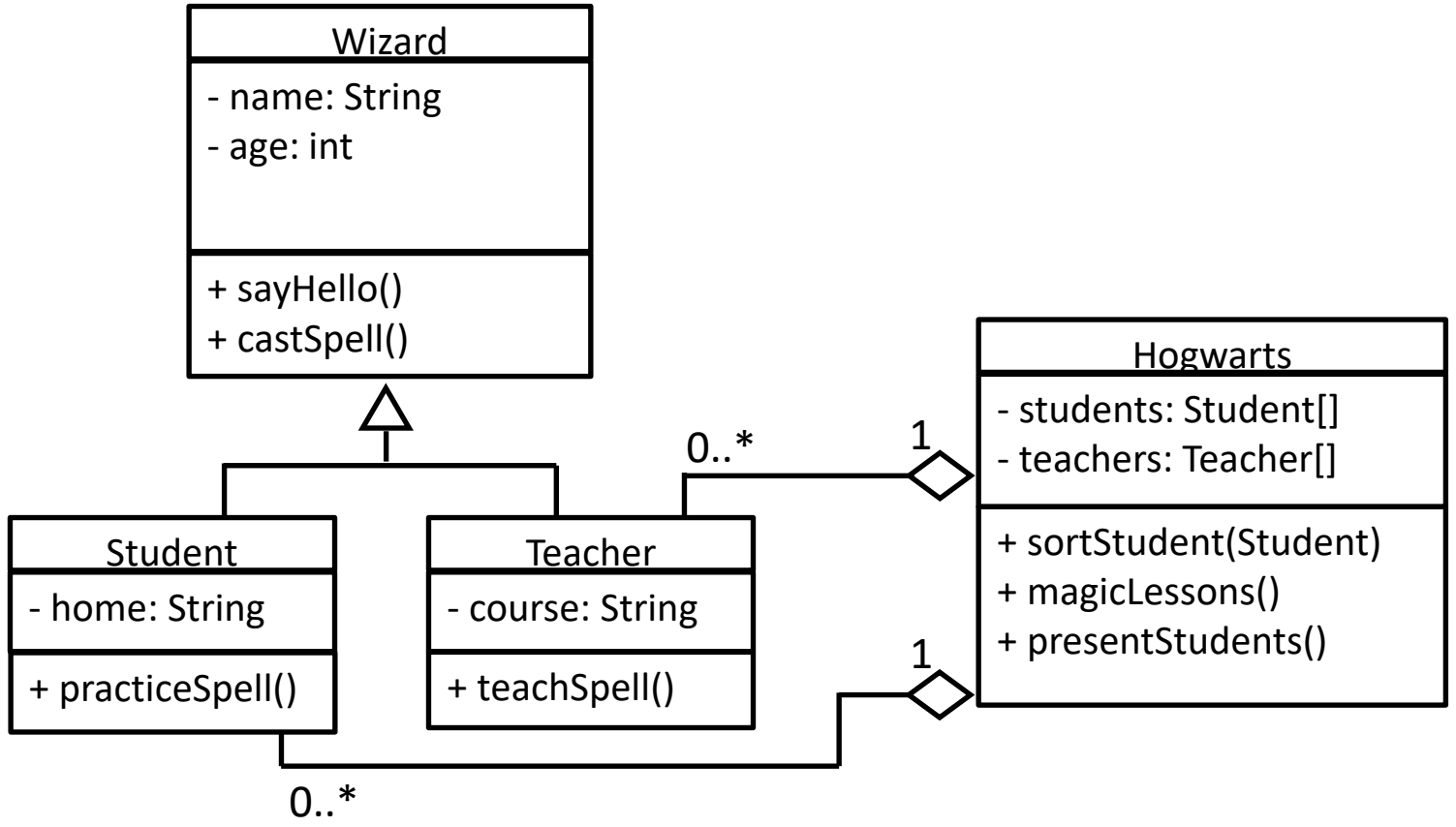
- Hogwarts behöver digitaliseras
- Det finns studenter, lärare, trollformler och Hogwarts självt
- Hur modellerar vi detta?
  - Objektorienterad analys av den kunskap vi har om problemet
  - Top-down: Börja med helheten och arbeta fram detaljer
  - Bottom-up: Börja med detaljer och sätt ihop helheten

Arv – ett stort exempel

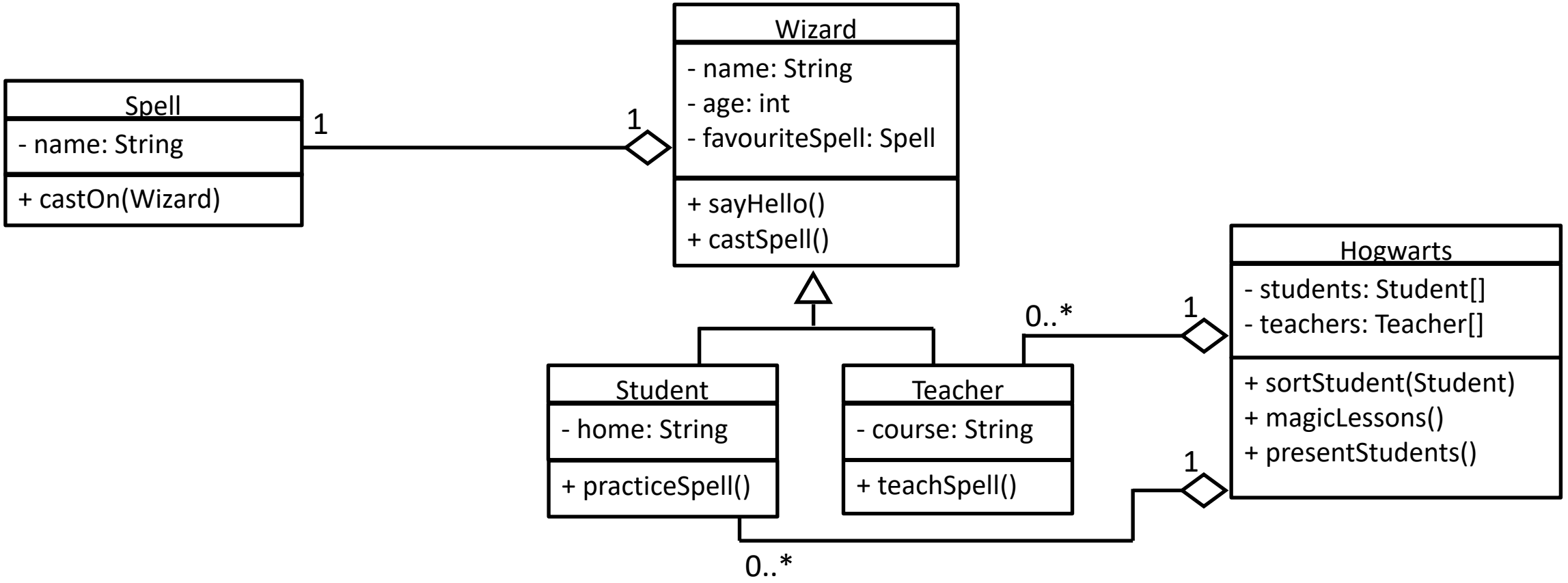




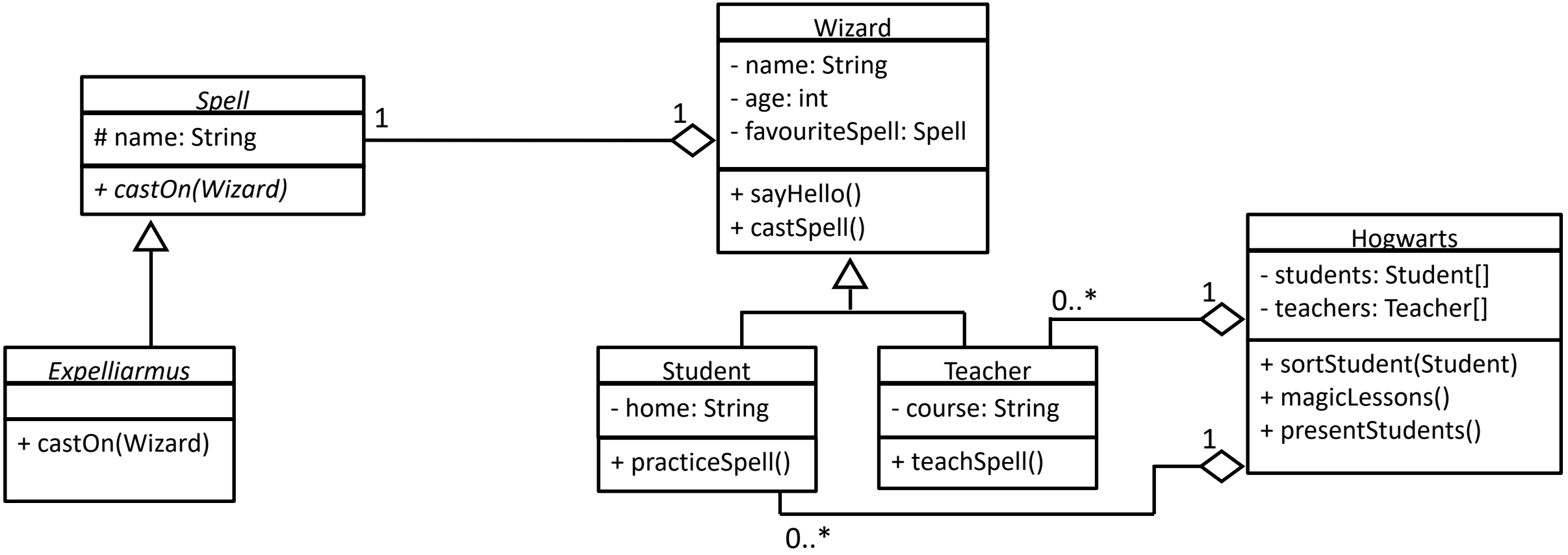
Arv – ett stort exempel



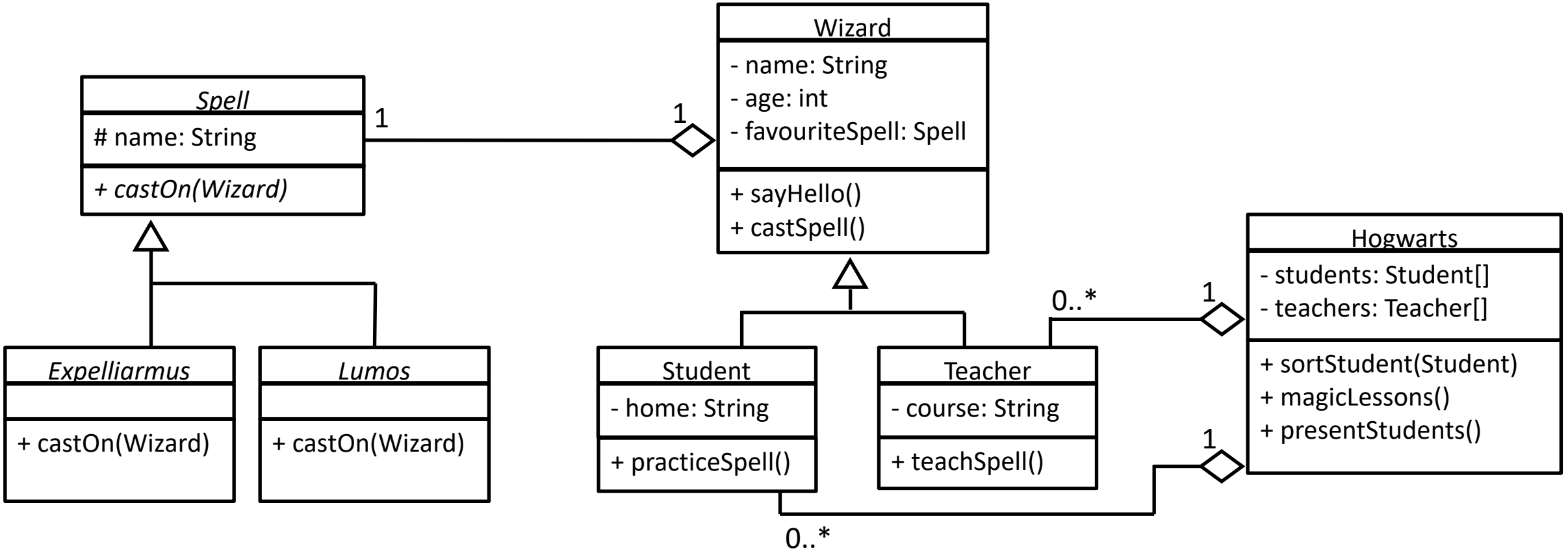
Arv – ett stort exempel



Arv – ett stort exempel



Arv – ett stort exempel



www.liu.se

LINKÖPINGS UNIVERSITET