Specifikacija, implementacija, abstrakcija

Specifikacija & implementacija

Specifikacija (angl. specification) `S` je *zaheva*, ki opisuje, kakšen izdelek želimo.

Implementacija (angl. implementation) `I` je izdelek. Implementacija `I` *zadošča* specifikaciji `S`, če ustreza zahtevam iz `S`.

V programiranju je implementacija programska koda. Specifikacije podajamo na različne načine in jih pogosto razvijemo postopoma:

- pogovor s stranko in analiza potreb
- dokumentacija, ki jo razume stranka
- tehnična dokumentacija za programerje

Brez specifikacije ne vemo, kaj je treba naprogramirati. Danes si bomo ogledali, kako v programskih jezikih poskrbimo za zapis specifikacij in kako programski jezik preveri, ali dana koda (implementacija) zadošča dani specifikaciji.

Omenimo še povezavo z algebro. V algebri poznamo *algebraične strukture*, na primer vektorske prostore, grupe, monoide, kolobarje, Boolove algebre, ... Definicija takih struktur poteka v dveh korakih:

- **signatura** pove, kakšne množice, konstante in operacije imamo
- **aksiomi** povedo, kakšnim zakonam morajo zadoščati operacije

Primer: grupa

- signatura:
  - množica G
  - operacija $\cdot : G \times G \to G$
  - operacija $^{-1} : G \to G$
  - konstanta $e : G$
- aksiomi:

$$
\begin{aligned}
x \cdot (y \cdot z) &= (x \cdot y) \cdot z \\
x \cdot e &= x \\
e \cdot x &= x \\
x \cdot x^{-1} &= e \\
x^{-1} \cdot x &= e
\end{aligned}
$$

Primer: usmerjen graf

- signatura:
  - množica V (vozlišča)
  - množica E (povezave)
  - operacija $\mathtt{src} : E \to V$ (začetno vozlišče povezave)
  - operacija $\mathtt{trg} : E \to V$ (končno vozlišče povezave)

- aksiomi: ni aksiomov

Zakaj vse to razlagamo? Ker programski jeziki ponavadi omogočajo zapis *signature* v programskem jeziku, ne pa tudi aksiomov, saj jih prevajalnik ne more preveriti.

Vmesniki

Specifikaciji včasih rečemo tudi **vmesnik (angl. interface)**, ker jo lahko razumemo kot opis, ki pove, kako se uporablja neko programsko kodo. Na primer, avtor programske knjižnice običajno objavi **API (Application Programming Interface)**, ki ni nič drugega kot specifikacija, ki pove, kako deluje knjižnica.

Torej imamo (vsaj) dve uporabi specifikacij:

- zahtevek za programsko kodo (specifikacija)
- protokol za uporabo programske kode (vmesnik)

Specifikacije v Javi

V Javi je specifikacija `S` podana z vmesnikom

```
public interface S {
   ...
}
```

v katerem lahko naštejemo metode. Tipe, ki nastopajo v specifikaciji, podamo kot generične razrede. Na primer, vmesnik za grupo bi zapisali takole:

```
public interface Group<G> {
    public G getUnit();
    public G multiply(G x, G y);
    public G inverse(G x);
}
```

Vmesnik za usmerjeni graf:

```
public interface Graph<V, E> {
    public V src(E e);
    public V trg(E e);
}
```

Seveda v praksi nihče ne piše takih vmesnikov, tu samo razmišljamo o zvezi med matematičnimi signaturami in vmesniki v programskih jezikih. Kasneje bomo videli bolj uporabne vmesnike, ki opisujejo abstraktne podatkovne strukture.

Specifikacije v OCamlu

V OCamlu lahko podamo poljubno signaturo (tipe in vrednost), ne moremo pa zapisati aksiomov, ki jim zadoščajo. Takole zapišemo signaturo za grupo:

```
module type GROUP =
sig
  type g
  val mul : g * g -> g
  val inv : g -> g
  val e : g
end
```

In takole za usmerjeni graf:

```
module type DIRECTED_GRAPH =
sig
  type v
  type e
  val src : e -> v
  val trg : e -> v
end
```

Implementacija v Javi

V Javi implementiramo vmesnik `I` tako, da definiramo razred `C`, ki mu zadošča:

```
public class C implements I {
   ...
}
```

Razred lahko hkrati zadošča večim vmesnikom. (Opomba: podrazredi so mehanizem, ki se *ne* uporablja za specifikacijo.)

Implementacija v OCamlu

Implementacija v OCamlu se imenuje **modul (angl. module)**. Modul je skupek definicij tipov in vrednosti, lahko pa vsebuje tudi še nadaljnje podmodule.

Nekaj primerov (nepraktičnih) implementacij grup in grafov si ogledamo v datoteki `algebra.ml`, kasneje pa bomo videli bolj uporabne primere.

Abstrakcija

Ko gradimo večje programske sisteme, so ti sestavljeni iz enot, ki jih povezujemo med seboj. Za vsako enoto je lahko zadolžena ločena ekipa programerjev. Programerji opišejo programske enote z *vmesniki*, da vedo, kaj kdo počne in kako uporabljati kodo ostalih ekip.

A to je le del zgodbe. Denimo, da prva ekipa razvija programsko enoto `E`, ki zadošča vmesniku `S` in da druga ekipa uporablja enoto `E` pri izdelavi svoje programske enote. Dobra programska praksa pravi, da se druga ekipa ne sme zanašati na podrobnosti implementacije `E`, ampak samo na to, kar je zapisano v specifikaciji `S`. Na primer, če `E` vsebuje pomožno funkcijo `f`, ki je `S` ne omenja, potem je druga ekipa ne sme uporabljati, saj je `f` namenjena *notranji* uporabi `E`. Prva ekipa lahko `f` spremeni ali zbriše, saj `f` ni del specifikacije `S`.

Če sledimo načelu, da mora programski jezik neposredno podpirati aktivnosti programerjev, potem bi želeli *skriti* podrobnosti implementacije `E` tako, da bi lahko programerji druge ekipe imeli dostop *samo* do tistih delov `E`, ki so našteti v `S`.

Kadar *skrijemo* podrobnosti implementacije, pravimo, da je implementacija **abstraktna**.

Programski jeziki omogočajo abstrakcijo v večji ali manjši meri:

- Java nadzoruje dostopnost do komponent z določili `private`, `public` in `protected`
- Python nima nikakršne abstrakcije
- OCaml omogoča abstrakcijo z določilom `M : S`, kjer je `M` module in `S` signatura. S tem skrijemo vsebino modula `M`, razen tistih komponent, ki so naštete v `S`.

Generično programiranje

Z izrazom *generično programiranje* razumemo kodo, ki jo lahko uporabimo večkrat na različne načine. Na primer, če napišemo knjižnico za 3D grafiko, bi jo želeli uporabljati na več različnih grafičnih karticah. Ali bomo za vsako grafično kartico napisali novo različico knjižnice? Ne! Želimo **generično** implementacijo, ki bo preko ustreznega *vmesnika* dostopala do grafične kartice. Proizvajalci grafičnih kartic bodo implementirali *gonilnike*, ki bodo zadoščali temu vmesniku.

Generično progarmiranje v Javi

Java podpira generično programiranje. Ko definiramo razred, je ta lahko odvisen od kakega drugega razreda:

```
public class Knjiznica3D<Driver extends GraphicsDriver> {
  ...
}
```

Generično programiranje v OCaml

V OCamlu je generično programiranje omogočeno s **funktorji (angl. functor)** (opomba: v matematiki poznamo funktorje v teoriji kategorij, ki nimajo nič skupnega s funktorji v OCamlu).

Funktor je preslikava iz struktur v strukture in je bolj splošen kot generični razredi v Javi (ker lahko struktura vsebuje podstrukture in definicije večih tipov, razred pa ne more vsebovati definicij podrazredov).

Funktor `F`, ki sprejme strukturo `A`, ki zadošča signaturi `S`, in vrne strukturo `B` zapišemo takole:

```
module F(A : S) =
struct
  ⟨definicija strukture B⟩
end
```

V `algebra.ml` je primer preprostega funktorja. Bolj uporaben primer sledi.

Primer: prioritetne vrste

Prioritetna vrsta je podatkovna struktura, v katero dodajamo elemente, ven pa jih jemljemo glede na njihovo *prioriteto*. Zapišimo specifikacijo:

Signatura:

- podatkovni tip `element`
- operacija `priority : element → int`
- podatkovni tip `queue`
- konstanta `empty : queue`
- operacija `put : element → queue → queue`
- operacija `get : queue → element option * queue`

Aksiomov ne bomo pisali, ker bi morali v tem primeru spoznati bolj zahtevne jezike za specifikacijo, ki presegajo okvir te lekcije. Neformalno pa lahko opišemo zahteve za prioritetno vrsto:

- `element` je tip elementov, ki jih hranimo v vrsti
- `priority x` vrne prioriteto elementa `x`, ki je celo število. Manjše število pomeni "prej na vrsti"

- `queue` je tip prioritetnih vrst
- `empty` je prazna prioritetna vrsta, ki ne vsebuje elementov
- `put x q` vstavi element `x` v vrsto `q` glede na njegovo prioriteto in vrne tako dobljeno vrsto
- `get q` vrne `(Some x, q')` kjer je `x` element iz `q` z najnižjo prioriteto in `q'` vrsta `q` brez `x` Operacija `get` vrne `(None, q)`, če je `q` prazna vrsta.

Implementacija v OCamlu

Oglejmo si implementacijo v OCamlu (datoteka `priority_queue.ml`).

Implementacija v Javi

Oglejmo si še implementacijo v Javi. V tem jeziku je bolj naravno narediti vrste kot objekte, ki se spreminjajo. Torej spremenimo specifikacijo.

Signatura:

- podatkovni tip `element`
- operacija `priority : element → int`
- podatkovni tip `queue`
- operacija `empty : unit → queue`
- operacija `is_empty : queue → bool`
- operacija `put : element → queue → unit`
- operacija `get : queue → element option`

Zahteve so podobne kot prej, le da operacije `empty`, `put` in `get` delujejo nekoliko drugače:

- `empty ()` vrne nov primerek (objekt) prazne vrste
- `put x q` vstavi `x` v vrsti `q` in s tem *spremeni* `q`
- `get q` vrne prvi `x` v vrsti `q` in s tem *spremeni* `q`

Primer: množice

Na vajah boste na dva načina implementirali končne množice. Oglejmo si OCaml signaturo, ki jih opisuje (datoteka `set.ml`).