Forslag om at bruge af Go og X i kurset OOPD

Oleksandr Shturmov

[date not setup]

1 Formål

I følgende foreslag omtales et sprog "X", der endnu ikke er bestemt men i sidste ende ønskes til at være enten C# eller Java; altså et rent objekt-orienteret sprog, der i skrivende stund aktivt bruges ude i erhvervslivet. X kan betragtes som både det nuværende sprog på kurset (Java), og det sprog der vil hovedsageligt benyttes i det ny foreslaget OOPD kursus (ikke nødvendigvis Java).

Fortsat fokus på et sprog der aktivt bruges i erhvervslivet giver os mulighed for at uddanne studerende til at bruge professionelle værktøjer og efter professionelle retningslinjer, men har også den ulempe at være præget af professionel inkompetence[1]. Det er i mellemtiden ikke formålet med dette forslag at behjælpe disse ulemper, men derimod at fremlægge et pædagogisk fremgangsmåde til en introduktion af studerende til imperativt og derefter objekt-orienteret programmering. Udgangspunktet er altså, at et sprog som "X", alene ikke kan anvendes effektivt til netop dette formål.

2 Foreslaget i et nøddeskal

Brug Google Go i de første par uger af kurset og brug den resterende tid på X. Eksamensopgaven må hermed bestå af 1/3 Go, og 2/3 X opgaver. Formålet er altså at bruge Go som et pædagogisk værktøj, ligesom Dr. Java[**?**] og BlueJ[**?**] har været brugt i forrige iterationer af kurset. Der argumenteres for i dette forslag at Go er et bedre udgangspunkt, givet de forventninger vi har til de studerende i hhv. starten og slutningen af kurset. Der argumenteres dernæst for at der kan forekomme en blød overgang fra Go til X ca. ved slutningen af den første 3. del af kurset.

3 Hvorfor starte med Go

Et af de mest iøjnefaldende grunde til at starte med Go er hvordan et HelloWorld program ser ud i Go:

[1] I form af f.eks. kringlet funktionalitet der overlever grundet fortsatte ønsker om bagudkompatibilitet, m.v.

```
1 package main
2
3 import "fmt"
4
5 func main() {
6   fmt.Println("Hello World!")
7 }
```

Antaget at programmet for oven er gemt som `HelloWorld.go`, kører man programmet således:

```
> 6g HelloWorld.go
> 6l HelloWorld.6
> ./6.out
Hello World!
```

Her er `6g` en oversætter, og `6l` er en sammenhæfter. De umiddelbare fordele ved sådan en start er følgende:

- Det er ikke nødvendigt at introducere en fortolker for at snakke om basal programkonstruktion da det basale program nemlig er så basal.
- Det er ikke nødvendigt at introducere klasser, static, stringe eller arrays for at snakke om et HelloWorld program.
- Et koncept som en oversætter bliver iøjnefaldende.

Det er bemærkelsesværdigt at vi mener at en introduktion af en fortolker[2] er ligefrem skadeligt for forståelsen af programmeringsparadigmet i X, der nemlig er oversætterbaseret. Vi har f.eks. i 2010 oplevet studerende der ikke vidste hvad en main funktion var ved slutningen af kurset.

Dog kan det siges at være en ulempe at man både skal oversætte og sammenhæfte sit program før man får lov at køre det. Dette kan der dog evt. korrigeres for ved at bruge et ekstra værktøj som f.eks. `godag`[3], der er en simpel udgave af noget til Go som det velkendte værktøj `ant` er for Java.

Det er dernæst oplagt at konstruere make filer (såfremt `godag` ikke benyttes). Vi mener at konstruktion af filer der eksplicit angiver rækkefølgen og afhængigheder blandt kildekodefiler er tiden værd, da efter vores erfarring er dette er noget man ofte støder ind i uanset om man programmerer Go, Java, C#, C++, el.lign.

Der er et par andre, lige så iøjnefaldende grunde som HelloWorld programmet. I Go benyttes der C-lignende strukturer. Objekter i Go er dermed, først og fremmest, strukturer, altså sammensætninger af primitive typer. Selvom Go's strukturer minder meget C's strukturer så er de dog meget nemmere at have at gøre med end i dem i C. F.eks. er følgende en sruktur for et punkt i to-dimensionelt rum:

```
1 type Point struct {
2   x, y int
3 }
```

[2] Grafisk eller kommandolinjebaseret, i som BlueJ eller Dr. Java.
[3] http://code.google.com/p/godag/.

Der er altså ikke eksplicitte klasser i Go, og Go er egentlig som udgangspunkt modul-baseret (hvilket er bl.a. grunden til at main funktionen ikke skal ind i en klasse som vi kender det fra bl.a. X). Felter (og instansmetoder) har i Go enten privat eller offentlig synlighed, alt afhængigt af om navnet er i stort eller småt.

Det andet grund er så at Go har pegere, men ikke pegeraritmetik, altså referencer, blandet med en god spildopsamler[4]. Her er et eksempel på brug af det ovennævnte struktur:

```
1 func Origo() *Point {
2   return &Point{0,0}
3 }
```

Så `*` og `&` bruges som sædvanligt i C, men en reference kan returneres uden at plads er blevet eksplicit allokeret til strukturen. Der er altså en spildopsamler der kommer ind og rydder op for de kære studerende, når "instancen" ikke længere er i brug. Syntaksen `&Point0,0`, ligner struktur-konstruktionssyntaks fra C, og i Go svarer til `new Point()`, desværre er der dog ingen måde at specificere konstruktor funktionalitet.

Fordelen er således at de mest enkelte strukturer kan introduceres nemt og hurtigt og en pæn linje kan tegnes mellem reference og værdityper, eftersom referencer angives eksplicit. Alt dette kan gøres uden at begrave de studerende i hukkomelsesstyring som ellers ville have været tilfældet havde det nu været C/C++ vi beskæftigede os med.

En sidste fordel der er værd at nævne er hvor pædagogisk velplaceret syntaksen er for at deklarere instansmetoder. Først og fremmest er der naturligvis mulighed for at smide en reference som en argument til en funktion. Objektorienteret programming, især i form af punktum-syntaks kan hermed introduceres som et værktøj til at spare en eksplicit angivelse af et reference argument.

En instansmetode defineres således:

```
1 func (*Point point) AddX(value int) int {
2   point.x += value
3 }
```

Og kaldes således:

```
1 p = Orego()
2 p.AddX(5)
```

Der er hermed nemt at tegne en linje mellem statiske og instansmetoder. Dernæst har instansmetoder, ligesom strukturens felter kan have enten privat eller offentlig synlighed alt afhængigt om navnet starter med stort eller småt.

4 Hvorfor ikke fortsætte med Go

Ud over det basale empiriske programmering brydder Go mange af de aspekter der ellers har været undervist, og bør undervises i et kursus med navnet

[4] Den er "god" pga. noget der nævnes om lidt.

"objekt-orienteret programmering og design". Specifikt har Go hverken indbygget understøttelse af type-hierarkier og parametrisk polimorfi. Det første er noget Go's udviklere står imod og det andet har ikke fundet sin vej ind i sproget endnu[**?**]. Det ville uden tvivl være nødvendigt at overveje at skifte helt til Go når, og hvis sproget får parametrisk polimorfi.

Derudover er der umiddelbart følgende mangler:

- Ingen syntaktisk sukker for struktur konstruktører.
- Ingen metode overlasning.
- Ingen dynamic dispatch.
- Interfaces fungerer på en grundlæggende anderledes måde.

5 Overgang til X

Følgende koncepter kan som anvist for oven introduceres vha. Go på en evolutionær[5] måde:

1. main funktionen
2. oversætter
3. modul afhængighedsstyring
4. statiske metoder
5. strukturer / objekter
6. primitive datatyper inkl. stringe
7. privat / offentlig synlighed af instansvariable og metoder
8. arrays (under spørgsmål[6])
9. if, for løkker
10. konstanter
11. simple datastrukturer

Ved slutningen af denne periode med Go, kan de studerende forventes at kunne skrive f.eks. en hægtet liste samt dynamisk array strukturer uden besvær.

Med disse værktøjer i værktøjsskuffen kan Java introduceres på praktisk talt 3-5 slides. I følgende sektioner, lad kodesekvenserne være placeret i to kolonner for hhv. Go og Java i hver slide.

[5] I som at alt hvad der foregår i en kodefil kan forklares med det samme, dvs. at der ikke er nogen overflødelige tegn og ord involveret.

[6] Der er nogle syntaktiske vanskeligheder omkring arrays i Go, der er endnu ikke fundet en god intuitiv afgrænsning så de "minder om" arrays som vi ser i X.

5.1 En klasse i Go vs. en klasse i Java

```
type Point struct {
  x, y int
  Z int
}

func (point *Point) Move(x, y int) {
  point.x += x
  point.y += y
  point.Z = point.x * point.y
}
```

```
public class Point
{
  private int x, y;
  public int Z;

  public Move(int x, int y)
  {
    this.x += x;
    this.y += y;
    Z = this.x * this.y;
  }
}
```

- Klassespecifikation vs. metodespecifikation.
- `point` vs. `this`.
- `Z` kan tilgås, så `this` kan være gratis.

5.2 En main funktion i Go vs. en main funktion i Java

```go
package main

func main()
{
  // ...
}
```

```java
public class Main
{
  public static void main(String[] arguments)
  {
    // ...
  }
}
```

- `main` skal være i en klasse.
- `main` skal have en denne signatur med argument array'et angivet.

5.3 Initializering i Go vs. Java

```
type Point struct {
  x, y int
}

func main()
{
  point = new(Point) // sets all values to 0.
}
```

```
public class Point
{
  private int x, y;
}

public class Main
{
  public static void main(String[] arguments)
  {
    Point point = new Point(); // sets all values to 0.
  }
}
```

- automatisk tilføjelse af en konstruktør.

5.4 Initializering i Go vs. Java # 2

```
type Point struct {
  x, y int
}

func main()
{
  point = &Point{3,5}
}
```

```
public class Point
{
  private int x, y;

  public Point(int x, int y) // "constructor"
  {
    this.x = x;
    this.y = y;
  }
}

public class Main
{
  public static void main(String[] arguments)
  {
    Point point = new Point(3,5);
  }
}
```

- mulighed for at specificere konstruktører.
- udelukkelse af tomme konstruktører ved tilstedeværelse af parametriserede konstruktører.
- evt. at en tom konstruktør (uden kode) vil igen tillade os at bygge objektet således at alle værdier sættes til 0. Der kan evt. beskrives fremgangsmåden for en klasses initializering fuldt ud, hvorved der evt. kan nævnes hvorfor man ikke skal intializere sine variabler i toppen af klassen.