Programmazione II

1. Introduzione (PDJ 1)

Dragan Ahmetovic

Conoscenze pregresse

Cosa avete imparato in *Programmazione I*?

- Basi di programmazione:
  - Come funziona un calcolatore e come interpreta le istruzioni
  - Cosa sono i linguaggi di programmazione, quali tipi ci sono e come funzionano
  - Quali costrutti ci sono in un linguaggio di programmazione e come si utilizzano
- Concetti applicati in un linguaggio specifico (GO)
  - ma validi per (quasi tutti) **linguaggi di programmazione moderni**
- Obiettivo: insegnare gli **strumenti universali** usati nella programmazione
  - L'appropriatezza dipende dal contesto e dall'abilità di usarli **nel modo giusto**

Conoscenze pregresse

Che tipo di programmazione abbiamo studiato in *Programmazione I*?

Esistono diversi paradigmi di programmazione. In *Programmazione I* avete studiato:

- **Programmazione Imperativa**:
  - Sequenza di operazioni (istruzioni) **in uno specifico ordine**
- **Programmazione Procedurale**:
  - Istruzioni strutturate in **procedure indipendenti, riusabili e parametrizzate**

Dalla programmazione imperativa a quella procedurale

Esempio: Programmazione Imperativa

```
1  package main
2  import "fmt"
3
4  func main() {
5    var eta1, eta2, eta3 int
6    fmt.Println("Eta persona 1 = ")
7    fmt.Scanln(&eta1)
8    fmt.Println("Eta persona 2 = ")
9    fmt.Scanln(&eta2)
10   fmt.Println("Eta persona 3 = ")
11   fmt.Scanln(&eta3)
12
13   var media float64 = float64(eta1 + eta2 + eta3) / 3
14   fmt.Println("Media eta =", media)
15
16   eta1 += 10
17   eta2 += 10
18   eta3 += 10
19
20   var media10anni float64 = float64(eta1 + eta2 + eta3) / 3
21   fmt.Println("Media eta fra 10 anni =", media10anni)
22 }
```

Dalla programmazione imperativa a quella procedurale

Limiti della programmazione imperativa:

- Al crescere della complessità, una singola sequenza di istruzioni diventa ingestibile.
- Problemi chiave:
  - **Comprensione del codice** - difficile seguire il flusso di esecuzione di un codice imperativo lungo
  - **Scalabilità** - l'aumentare della complessità (es: molti livelli di selezione) complica la gestione
  - **Riusabilità** - bisogna riscrivere codice simile ogni volta che dobbiamo risolvere un problema simile
  - **Modificabilità** - modifiche possono avere effetti indesiderati in altre parti del codice

Soluzione: programmazione procedurale

- **Decomposizione** del codice in **procedure**, unità logiche isolate, autonome e generalizzabili
  - Come fare questa decomposizione?
- Evoluzione del **paradigma di programmazione**, cioè del modo di scrivere il codice.
  - Avere un linguaggio che supporta il nuovo paradigma non basta - bisogna saperlo applicare!

Dalla programmazione imperativa a quella procedurale

Esempio: Programmazione Procedurale

```
1  package main
2  import "fmt"
3
4  func avg(eta []int) (result int) {
5    for i := range eta {
6      result += eta[i]
7    }
8    return result/len(eta)
9  }
10
11 func read(c int) []int {
12   o := make([]int, c)
13   for i := range o {
14     fmt.Printf("Eta persona %d = \n", i)
15     fmt.Scanln(&o[i])
16   }
17   return o
18 }
19
20 func incrementSlice(s []int, p int) []int {
21   o := make([]int, 0, len(s))
22   for _,e := range s {
23     o = append(o, e+p)
24   }
25   return o
26 }
27
28 func main() {
29   eta := read(3)
30   fmt.Println("Media eta = ", avg(eta))
31   etaplus10 := incrementSlice(eta, 10)
32   fmt.Println("Media eta fra 10 anni = ", avg(etaplus10))
33 }
```

Dalla programmazione imperativa a quella procedurale

Problemi aperti nella programmazione procedurale

- Svariate criticità nel caso del codice molto articolato e scritto da più persone:
  - Capire come usare il codice degli altri in maniera corretta
  - Far capire agli altri come usare il proprio codice in maniera corretta
  - Evitare di duplicare codice / favorire riuso di componenti
  - Mantenere sempre coerente lo stato (di tutte le componenti) del sistema
- Strutturare codice molto articolato e di grandi dimensioni
- Risolvere in maniera efficace problemi ricorrenti
- Scrivere codice parallelizzabile e performante su dati di grandi dimensioni

Programmazione II costruisce sulle basi fornite in precedenza, estendendole con ulteriori paradigmi e tecniche di sviluppo **fondamentali**, al fine di rendere più facile la soluzione dei problemi individuati.

Argomenti del corso

Un argomento principali:

- **Programmazione a oggetti**
  Focus: imparare un paradigma di programmazione che permetta di scrivere codice a prova di errore, facilmente riutilizzabile da altri senza doverlo conoscere approfonditamente.
- (Cenni di) **Software engineering**
  Focus: imparare tecniche per risolvere in maniera efficace problemi ricorrenti, strutturare progetti complessi e con molti sviluppatori.
- (Cenni di) **Programmazione funzionale**
  Focus: imparare un paradigma di programmazione adatto per svolgere operazioni scomponibili e parallelizzabili.

Decomposizione

Per progettare programmi non banali, serve scomporre il problema in sottoproblemi

I limiti sia della Programmazione Imperativa, sia di quella Procedurale sono legati alla difficoltà di strutturare il problema in parti più facili da progettare, implementare e manutenere.

La **Decomposizione** è la suddivisione (fattorizzazione) di un **problema complesso** in **sottoproblemi** (moduli) separabili che siano più comprensibili, più facili da implementare e manutenere, anche in maniera indipendente.

- I sottoproblemi dovrebbero essere logicamente coesi e svolgibili indipendentemente.
- Combinando le soluzioni dei sottoproblemi si dovrebbe poter risolvere il problema originale.
- I sottoproblemi dovrebbero essere allo stesso livello di dettaglio.

Astrazione

Il modo per modellare il processo di Decomposizione è mediante **Astrazione**

L'Astrazione è l'identificazione delle caratteristiche rilevanti del problema, slegate dalla loro effettiva implementazione.

- Può essere visto come un mapping molti -> uno al fine di definire un meccanismo generale per risolvere una serie di problemi simili.
- Es: In matematica le operazioni sono delle astrazioni che ci permettono di svolgere calcoli con diversi valori degli operandi

Meccanismi di astrazione:

- **Astrazione per parametrizzazione** - genero entità che dipendono e lavorano su dei parametri, al posto dei quali potranno essere utilizzati diversi valori effettivi
- **Astrazione per specifica** - dichiaro, in maniera esplicita o implicita, il modo in cui queste entità devono essere utilizzate (senza doverne conoscere l'implementazione)

Astrazione

Esempio di astrazione per parametrizzazione

Posso estrarre blocchi logici e indipendenti di codice, ad esempio:

```
1 func at(b,a) float {
2   return b*a/2
3 }
```

In seguito posso svolgere le operazioni richiamandole coi parametri specifici:

```
1 a = at(4,5)
```

Astrazione per parametrizzazione definisce le entità coinvolte nella computazione ma non indica l'intenzione della computazione

Astrazione

Esempio di astrazione per specifica

Bisogna descrivere l'obiettivo della computazione, spiegando cosa richiede (precondizione) e cosa ottiene (postcondizione)

```
1   func at(b,a) float {
2   //REQUIRES: b, a > 0  (precondizione)
3   //EFFECTS: returns the area of the triangle given its base b and height a (postcondizione)
4       ...
5   }
```

Spesso, l'utilizzo dei nomi appropriati fornisce già un buon livello di astrazione per specifica

```
1 func area_triangle(base,height) float {
2   return base*height/2
3 }
```

Astrazione

Tipi di astrazione

Il processo di astrazione è utilizzato a diversi livelli. Usando i meccanismi di astrazione (e gli strumenti forniti dal linguaggio di programmazione) si possono quindi definire diversi tipi di astrazione. Noi ne vedremo 4:

- Astrazione procedurale
- Astrazione dei (tipi di) dati
- Gerarchia dei tipi di dati
- Astrazione di iterazione

Astrazione

Astrazione procedurale

Astrazione che caratterizza la definizione di **sottoprogammi** (metodi, funzioni).

Mediante astrazione per parametrizzazione generalizzo un'operazione a una procedura con gli operandi trasformati in parametri

Mediante astrazione per specifica definisco cosa fa la procedura senza dover spiegare come è implementata

Questo tipo di astrazione è supportato nei linguaggi procedurali

Astrazione

Astrazione dei (tipi di) dati

Astrazione che caratterizza la definizione di nuovi **tipi di dato** (pensate alle **struct** in go), con i loro **comportamenti** (in OOP, i tipi di dato possono avere dei comportamenti - funzioni che dipendono dal valore del dato)

Mediante astrazione per parametrizzazione generalizzo i comportamenti del tipo di dato

Mediante astrazione per specifica definisco cosa modella il tipo di dato e i suoi comportamenti, senza spiegare come è costruito il tipo di dato o come sono implementati i suoi comportamenti.

Questo tipo di astrazione è supportato in molti linguaggi procedurali e nei linguaggi orientati a oggetti

Astrazione

Gerarchia dei tipi di dati

Astrazione che caratterizza la creazione dei nuovi tipi di dato **partendo da tipi di dato esistenti** ed estendendone le capacità.

Mediante astrazione per parametrizzazione posso (ri-)definire i comportamenti del nuovo tipo di dato, possibilmente riutilizzando quelli del dato "Genitore"

Mediante astrazione per specifica definisco le differenze e le similitudini nel comportamento, senza spiegare come queste sono implementate.

Questo tipo di astrazione è supportato nei linguaggi orientati a oggetti

Astrazione

Astrazione di iterazione

Astrazione che permette di accedere a una sequenza di dati qualunque

Mediante astrazione per parametrizzazione definisco i metodi per iterare sulla sequenza di dati (come chiedere il prossimo dato? come capire se ci sono altri dati?)

Mediante astrazione per specifica definisco il comportamento dell'iterazione (in che modo e ordine mi arrivano gli elementi), senza spiegare come questo processo è implementato o come è memorizzata la sequenza di dati.

Questo tipo di astrazione è possibile implementarlo nei linguaggi orientati a oggetti, alcuni li forniscono direttamente nelle primitive del linguaggio

Programmazione II

1. Introduzione (PDJ 1)

Dragan Ahmetovic