3. Progettazione di un Tipo di Dato Concreto

L'obiettivo del capitolo é l'acquisizione di metodi e tecniche che rendano più semplice l'attivitá di sviluppo del software. Ci concentriamo quindi sul **processo di sviluppo** del software e non sul prodotto che esso ottiene.

Prerequisiti

Classi
Tramite la parola riservata **class** é possibile definire l'interfaccia di una classe composta dai metodi e proprietà che gli oggetti andranno a mettere a disposizione all'esterno. Decidiamo poi quali elementi saranno pubblici, quali privati e quali protetti. Vediamo un esempio.

```
class tipo {
     public:
          int var1;
          int var2;
          void funzione1();
     private:
          char var3;
          void funzione2();
     protected:
          float var4;
};
```

É buona norma inserire la definizione di una classe (interfaccia) in un file di intestazione detto **header** con estensione .h e le implementazioni dei metodi della classe nel file cpp.

Costruttore
Il costruttore é una funzione (che deve avere lo stesso nome della classe a cui appartiene) che serve quando viene dichiarato l'oggetto. Si occupa di allocare la memoria e inizializzare l'oggetto, ad esempio.

```
class tipo {
     int i;
     public:
          tipo(); // Primo semplice costruttore
};
```

```
tipo::tipo() {
     i=0;
};
```

Distruttore
Viene chiamato quano l'oggetto viene rilasciato dalla memoria. Prende il nome dalla classe preceduto da una tilde ∼. Non ha nessun valore di ritorno e non prende parametri in quanto non é il programmatore a chiamarlo direttamente.
`∼tipo();`
Il distruttore libera la memoria che era occupata dall'oggetto.

Costruttore di Copia
Costruttore utilizzato quando viene effettuata un'allocazione di memoria all'interno dell'oggetto. In questo caso é necessaria una copia dell'atto di allocazione in modo che i valori associati ai due oggetti siano effettivamente diversi. In questo modo si risolve il problema di condivisione di memoria indesiderata. Vediamo un esempio.

```
tipo::tipo(const tipo &t) {
     i=t.i; // Copia effettiva del campo i
}
```

Overload di Operatori
A livello implementativo, gli operatori sono a tutti gli effetti delle funzioni dove il nome é dato da `operator@` dove @ é il simbolo dell'operatore (+, -, *, ecc).
Il linguaggio C++ offre la possibilitá di ridefinire gli operatori in modo che possano adattarsi al meglio ai tipi definiti dall'utente.
Il problema principale riguarda la visibilitá degli operatori sovraccaricati. Alcuni dovranno essere sovraccaricati solo come mebri di classe tramite l'uso di `this`, altri invece come funzioni esterne (ad esempio gli operatori << e >>) e altri indifferentemente in uno dei due modi.

Classi Derivate
Tramite **ereditarietá** é possibile creare una classe generale (**superclasse**) che definisce le caratteristiche comuni a tutti gli oggetti ad essa correlati. Questa classe puó poi essere eridatata da una o piú classi (**sottoclassi**) che possono aggiungere solo elementi specifici senza modificare la superclasse. Il concetto si puó poi iterare a piú livelli.
Il tipo di accesso che la sottoclasse ha sugli elementi della superclasse sono definiti dagli **specificatori di accesso** (public, private, protected).

TDD - Test Driven Development
L'attivitá di progettazione e sviluppo del codice viene guidata da dei test scritti prima del codice vero e proprio che implementa interfaccia e funzionalitá. Ci troveremo dunque, ad esempio, con tre distinti file sorgente del tipo:

- `testRazionale.cc` che conterrá il codice del test che ci guiderá allo sviluppo dell'interfaccia per la classe, in questo caso, Razionale.
- `Razionale.hh` che sará l'header che contiene l'interfaccia della classe.
- `Razionale.cc` che contiene l'effettiva implementazione della classe.

L'attivitá di test serve per verificare che la classe che si sta implementando si comporti correttamente nei casi previsti dal test stesso. Bisogna sottolineare che questa operazione va effettuata in modo sistematico, in quanto é molto raro effettuare test esaustivi. Infatti, il test ci permette di individuare gli errori ma non é in grado di dimostrarne la totale assenza.

Invariante di Classe
Proprietá che deve essere soddisfatta in qualunque momento dalla rappresentazione scelta per il tipo di dato. Questa invariante viene codificata all'interno di un metodo cosí che poi tramite le **asserzioni** possiamo verificare a nostro piacimento, in un punto qualsiasi del codice, che questa sia verificata.
Bisogna quindi fare attenzione a non fornire all'utente che utilizza la classe l'occasione di rompere l'invariante.
Questo puó accadere inavvertitamente, ad esempio creando metodi pubblici che rompono l'ivariante che vanno in overloading con metodi pubblici giá esistenti e verificati. L'attivitá di progettazione prevede infatti che l'aggiunta di un nuovo metodo venga valutata per capire se da un accesso incontrollato alla rappresentazione interna e, di conseguenza, mette a rischio l'invariante.

Programmazione per Contratto
Prevede che ogni volta che un utente fa accesso a un oggetto di un tipo di dato deve rispettare il contratto scritto da chi ha sviluppato quel determinato oggetto.
Questo contratto stabilisce le **precondizioni** che l'utente deve soddisfare per poter invocare quella funzionalitá e quali sono le **postcondizioni** che l'implementatore deve invece garantire una volta usata la funzionalitá. Spesso si rendono esplicite le condizioni sulle invarianti e si ottiene un'implicazione logica del tipo:

```
precondizioni AND invarianti => postcondizioni AND invarianti
```

Ovviamente, se l'utente non soddisfa le precondizioni allora l'implicazione sará sempre vera e nemmeno l'implementatore dovrá soddisfare le postcondizioni.
Vediamo un esempio pratico.

```
// Esempio sull'opratore di divisione tra oggetti di tipo Razionale

Razionale operator/(const Razionale& x, const Razionale& y) {
     assert(x.check_inv() && y.check_inv()); // Invarianti in ingresso
     assert(y != Razionale(0)); // Precondizione

     Razionale res = x;
     res /= y;

     assert(res.check_inv()); // Invarianti in uscita e postcondizione
     return res;
}
```

Contratto Narrow
Detto anche contratto **stretto**, in questo caso non tutte le combinazioni degli argomenti di input sono lecite. Nell'esempio di prima non era lecita la combinazione dove il secondo argomento y fosse uguale a zero, infatti si trattava di un contratto narrow.
In questo tipo di contratti l'implementatore si impegna a fornire la funzionalitá solo quando questa assume un senso (e quindi a valori forniti in input legittimi). Sará dunque l'utilizzatore a decidere se tale legittimitá sará verificata o meno. In C++ questo tipo di contratto é molto frequente sia a livello di linguaggio stretto (ad esempio sta al programmatore verificare la validitá dell'indice di un array) che a livello di libreria standard.

Contratto Wide
In questo caso sta all'implementatore assicurarsi e gestire quelle situazioni in cui l'utilizzatore inserisce dei valori erronei (nell'esempio di prima, y = 0). Si tratta quindi di spostare una parte del contratto dalle precondizioni alle postcondizioni.
Vediamo il contro-esempio.

```
Razionale operator/(const Razionale& x, const Razionale& y) {
     assert(x.check_inv() && y.check_inv()); // Invarianti in ingresso

     // Controllo postcondizione:  non esiste piú l'asserzione che
     richiede y diverso da zero.  Infatti non possiamo piú, come implementatori,
     pensare che l'utente ci abbia fornito i valori corretti.  In caso
     di contratto wide dobbiamo controllare se y é uguale a zero e
     in caso comportarci di conseguenza.
     if(y == Razionale(0))
          throw DivByZero;

     Razionale res = x;
```

```
    res /= y;

    assert(res.check_inv()); // Invarianti in uscita
    return res;
}
```

In generale, i contratti wide sono piú onerosi per l'implementatore in termini di efficienza e di righe di codice.

Contratti in C++
Nello standard del C++ viene definito per ogni costrutto quale contratto deve rispettare, e di conseguenza l'eventuale comportamento (**behavior**) che l'implementazione deve assumere quando questo contratto viene infranto.
Questi behavior vengono classificati nelle seguenti categorie:

- *Specificato*, il comportamento viene descritto dallo standard e l'implementazione é tenuta a seguire questa prescrizione.
- *Implementation-defined*, ogni implementazione puó scegliere liberamente come sviluppare la funzionalitá ma ha l'obbligo di documentare la scelta fatta.
- *Non specificato*, comportamento che dipende dall'implementazione che peró, in questo caso, **NON** deve documentare la scelta fatta.
- *Non definito*, comportamento che si verifica quando l'utente viola una precondizione, l'implementazione quindi non ha piú nessun obbligo nei confronti dell'utilizzatore. Il risultato viene detto undefined-behavior (**UB**) e l'implementazione puó comportarsi in qualsiasi modo.