15 - Programmazione per contratto

La programmazione per contratto (*Design by contract*) è un paradigma in cui una classe o una funzione viene definita tramite un contratto tra il programmatore che la implementa e quello che la utilizza.
Questo contratto specifica chiaramente:

1. **precondizioni**: le condizioni che devono essere vere prima che una funzionalità venga invocata, responsabilità dell'utente garantire che siano soddisfatte;
2. **postcondizioni**: le condizioni che devono essere vere dopo l'esecuzione della funzionalità, responsabilità dell'implementatore garantire che vengano rispettate;
3. **invarianti di classe**: proprietà che devono essere sempre vere per gli oggetti della classe, sia prima che dopo l'esecuzione di qualsiasi funzionalità.
   Questa struttura permette di rendere il comportamento di una classe prevedibile e robusto, poichè ciascuna delle parti (implementatore e utilizzatore) sa esattamente quali sono i propri obblighi e cosa aspettarsi.

Forma del contratto

Un contratto si può rappresentare come una implicazione logica:
precondizioni ⇒ postcondizioni
Questo significa che se le precondizioni sono soddisfatte, allora l'implementatore si impegna a garantire le postcondizioni. Se invece le precondizioni non sono valide, l'implementatore non ha alcun obbligo.
Spesso si includono esplicitamente anche le invarianti di classe, specificando il contratto:
precondizioni AND invarianti ⇒ postcondizioni AND invarianti
In questo caso:

- le precondizioni e postcondizioni sono definite "al netto" delle invarianti, cioè assumono che le invarianti siano già rispettate.

Esempio classe Razionale

```
Razionale operator/(const Razionale& x, const Razionale& y) {
  assert(x.check_inv() && y.check_inv()); // Controllo delle invarianti di ingresso
  assert(y != Razionale(0));              // Precondizione: y deve essere diverso da 0

  Razionale res = x;
  res /= y;

  // Invarianti in uscita (non serve verificarle per x e y, essendo const)
  assert(res.check_inv());
  return res;
}
```

- **precondizione**: `y` deve essere diverso da 0;
- **invarianti**: gli oggetti `x` e `y` devono rispettare le loro condizioni di validità interna;
- **postcondizione**: il risultato restituito deve essere corretto e mantenere le invarianti.

Contratti Narrow e Wide

Contratto Narrow (stretto)

Un contratto narrow specifica che l'implementatore fornisce la funzionalità solo se vengono rispettate determinate precondizioni. L'onere di garantire che queste siano vere è totalmente a carico dell'utilizzatore.
Caratteristiche:

- l'implementatore non deve gestire scenari in cui le precondizioni non sono valide;
- le precondizioni vengono spesso imprementate tramite asserzioni (`assert`), che funzionano solo in modalità di debug;
- più efficiente, poichè non richiede controlli aggiuntivi durante l'esecuzione.

```
Razionale operator/(const Razionale& x, const Razionale& y) {
  assert(y != Razionale(0)); // L'utilizzatore deve garantire che y ≠ 0
  Razionale res = x;
  res /= y;
  return res;
}
```

Ad esempio:

- quando si accede ad un elemento di un array, l'onere di controllare la validità dell'indice è a carico del programmatore;
- a livello di libreria standard spetta all'utente controllare che `std::vector<T>` non sia vuoto prima di eliminare l'ultimo elemento con la `pop_back()` .

Contratto wide (ampio)

Un contratto wide prevede che l'implementatore si occupi di verificare le leggittimità delle precondizioni e gestisca eventuali violazioni. Questo sposta parte dell'onere dell'utilizzatore all'implementatore.
Caratteristiche:

- l'implementatore deve gestire tutte le situazioni, inclusi casi non validi;
- i controlli devono essere espliciti e attivi anche in modalità di produzione (quindi non è possibile utilizzare le asserzioni);
- è meno efficiente e più complesso da implementare, ma riduce gli errori lato utente.

```
Razionale operator/(const Razionale& x, const Razionale& y) {
  if (y == Razionale(0)) {
    throw DivByZero(); // L'implementatore gestisce il caso di errore
  }
  Razionale res = x;
  res /= y;
  return res;
}
```

Differenze principali

contratto Narrow	contratto Wide
l'utente deve garantire le precondizioni	l'implementatore verifica le precondizioni
più efficiente	meno efficiente
controlli con `assert` (solo in debug)	controlli espliciti, sempre attivi.

Contratti nel linguaggio C++ e nelle libreria standard

Lo standard del C++ descrive ogni funzionalità del linguaggio e della libreria con un contratto implicito o esplicito, classificando il comportamento in base al livello di specifica:

Categorie di comportamento

1. **comportamento specificato (specified behavior)**:
   - il comportamento è definito esattamente dallo standard e ogni implementazione deve rispettarlo;
   - es: l'operazione di somma tra interi produce sempre il risultato atteso se non si verificano overflow.
2. **comportamento definito dall'implementazione (implementation-defined behavior)**:
   - lo standard permette all'implementazione di scegliere una modalità, ma la scelta deve essere documentata;
   - es: la dimensione dei tipi interi (`int` , `long` ...) dipende dall'implementazione ma deve essere dichiarata.
3. **comportamento non specificato (unspecified behavior)**:
   - lo standard non specifica quale comportamento debba essere adottato e l'implementazione non è tenuta a documentarlo;
   - es: l'ordine di valutazione degli argomenti in una chiamata di funzione.
4. **comportamento non definito (undefined bahavior)**:
   - si verifica quando vengono violate le precondizioni. Lo standard non specifica alcun vincolo e l'implementazione può comportarsi in modo arbitrario;
   - es: accedere a un elemento di un array fuori dai limiti, scrivere su una variabile `const` , overflow su tipi interi con segno;
   - pericoloso perchè può generare risultati imprevedibili, tra cui crash o modifiche indesiderate.
5. **comportamento locale (locale-specific behavior**:
   - dipende dalle impostazioni locali (lingua, cultura, convenzioni);
   - es: l'interpretazione dei caratteri dipende dal "character set" locale.

es. 3 prova in itinere 2016
La seguente classe presenta alcuni problemi che ne rendono l'utilizzo problematico. Individuare i problemi ed indicare una possibile soluzione (riscrivendo l'interfaccia).

```
struct Matrix {
    //...
    size_type num_rows();
    size_type num_cols();
    value_type& get (size_type row, size_type col);
    Matrix& operator-();
    Matrix& operator+=(Matrix y);
```

```
    Matrix& operator+(Matrix y);
    void print(ostream os);
    //...
};
```

correzione:

```
Struct Matrix {
    size_type num_rows() const;
    size_type num_cols() const;
    value_type& get (size_type row, size_type col); //ha senso fornire un metodo in
scrittura
    const value_type& get(size_type row, size_type col) const;
    Matrix operator-() const; //calcola un nuovo valore che è il negato della mat,
perciò restituisco per valore
    Matrix& operator+=(const Matrix& y);
    Matrix operator+(const Matrix& y) const; //calcola un nuovo valore perciò va
restituito
    void print(ostream& os) const; //const a destra per poter passare anche le mat
marcate const,
}
```

codice utente:

```
void foo(Matrix& m) {
    m.get(1,1) = 17;
    Matrix y;
    m += y; // m = m + y => ha senso tenere y const
    Matrix z;
    z = m + y;
}

void bar(const Matrix& m) {
    std::cout << m.get(1,1); //non è possibile utilizzare la terza funzione perchè
non è marcata const
}
```

11 - Risoluzione dell'overloading