TEST DI PROPRIETÀ

in C++ con `rapidcheck`

9 Aprile 2020

Yuri Valentini

Sommario

- Yuri Valentini
- Test di unità
- Approccio basato su proprietà
- Introduzione a rapidcheck
- Diamond Kata
- Individuare le proprietà
- Uso avanzato di rapidcheck
- Test stateful
- Domande e risposte

Chi è Yuri Valentini

- sw embedded, biomedicale, VOIP
- C/C++, Python, C#
- TDD, XP (non windows)
- programmazione ad oggetti
- stampa 3D con OpenSCAD
- ... programmazione funzionale
- http://github.com/yuroller

Test di unità

Anatomia dei test di unità

Schema di funzionamento:

1. prepara i dati di ingresso
2. esegui una operazione sui dati
3. verifica qualcosa sul risultato

Test di unità con GoogleTest

```
#include <gtest/gtest.h>

TEST(SommaTest, UnoPiuUnoFaDue) {
    EXPECT_EQ(2, somma(1, 1));
}

TEST(SommaTest, DuePiuDueFaQuattro) {
    EXPECT_EQ(4, somma(2, 2));
}
```

Caratteristiche test di unità

- facili da comprendere/scrivere
- verificano casi specifici (esempi)
- utili per evitare regressioni

Svantaggi test di unità

- difficile capire quando si è finito
- può portare a mancanza di generalità (es. TDD)
- molti test per garantire copertura

Implementazione “fantasiosa”

Errata ma fa passare i test precedenti

```
int somma(int a, int b) {
    return a + a;
}
```

Test di proprietà

Anatomia dei test di proprietà

Schema di funzionamento:

1. per *tutti* i dati di ingresso
   che soddisfano certi criteri
2. esegui una operazione sui dati
3. verifica qualcosa sul risultato

Per test di unità il primo punto è:

- prepara i dati di ingresso

Caratteristiche test di proprietà

- coprono l'intero spazio di input
- input minimale per fallimenti
- randomizzati ma riproducibili
- spingono maggiormente a ragionare
- sono test più generali
- possono rivelare casi impensabili
- assicurano di avere capito bene i requisiti

Proprietà di somma()

- Proprietà commutativa

```
somma(a, b) = somma(b, a)
```

- Proprietà associativa

```
somma(somma(a, b), c) = somma(a, somma(b, c))
```

- Elemento neutro

```
somma(0, a) = a
```

Introduzione a rapidcheck

https://github.com/emil-e/rapidcheck

Caratteristiche rapidcheck

- Framework C++ per test basati sulle proprietà
- Ispirato a `QuickCheck` di Haskell
- Permette di scrivere le proprietà in stile imperativo
- Contiene modulo di riduzione input
- Supporto per i tipi STL (anche std::map e std::set)
- Creazione di generatori di input molto flessibile
- Test stateful
- Integrazione con altri framework di test (GoogleTest, ecc.)

Moduli di rapidcheck

Stessi moduli di QuickCheck e di molti altri framework:

- generatore
- valutatore (API per descrivere proprietà)
- riduttore di input (shrinker)

Esempio “da manuale”

```
#include <rapidcheck.h>
#include <vector>
#include <algorithm>

int main() {
  rc::check("double reversal yields the original value",
            [](const std::vector<int> &l0) {
              auto l1 = l0;
              std::reverse(std::begin(l1), std::end(l1));
              std::reverse(std::begin(l1), std::end(l1));
              RC_ASSERT(l0 == l1); // return l0 == l1;
            });

  return 0;
}
```

Precondizioni

```
rc::check([](const std::string &str) {
    RC_PRE((str.size() % 2) == 0);
    // ...
});
```

> Il valutatore rinuncia con condizioni troppo restrittive

Generatori

```
auto n = *rc::gen::arbitrary<int>();
auto i = *rc::gen::inRange(0, 10); // 0..9
```

Supporta tipi STL e tipi personalizzati

Diamond Kata

Data una lettera, creare un rombo che comincia per 'A' con la lettera fornita nel punto piu' ampio.

Per esempio: crea_rombo('C') fornisce

```
"  A  "
" B B "
"C   C"
" B B "
"  A  "
```

Individuare le proprietà

- `Presentazione` di Scott Wlaschin (pagine 73-89)
- `Video` di Andrea Leopardi (dal minuto 14)

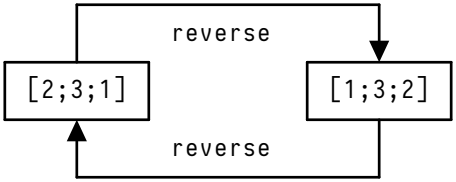
Funzioni simmetriche

```
decode(encode(term)) == term;
```

Esempi:

- serializzazione/deserializzazione di oggetti (JSON, XML, ecc.)
- escape/unescape di stringhe (URLEncode, ecc.)

Funzioni reversibili



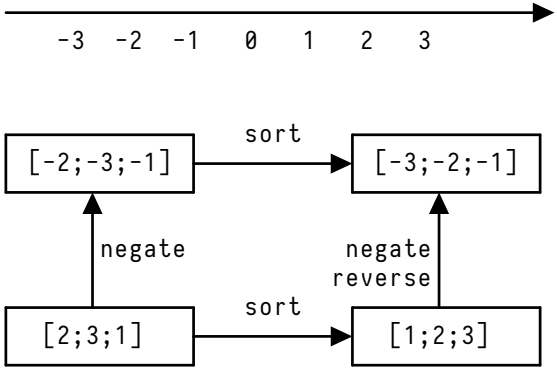
Invarianti

Esempi:

- sort() non cambia il numero di elementi
- gli elementi dopo sort() sono quelli di input
- un albero binario bilanciato, rimane tale dopo inserimenti

Percorsi differenti, stessa destinazione

Per testare sort() sfrutto la simmetria dei punti sull'origine



Proprietà matematiche

Posso sfruttare:

- commutatività
- associatività
- idempotenza (es. sort(), find_if())
- proprietà funzionali (es. leggi delle monadi e funtori)
- induzione strutturale es.:
  1. dopo il sort() qualunque slice è ordinato
  2. relazione fra i nodi di un albero bilanciato ordinato

Problemi asimmetrici

Difficili da implementare, ma facili da verificare

```
std::vector<int> fattorizza(int n) {
    // implementazione
    return ...;
}
```

Verifico che il prodotto in uscita sia il numero in input

Oracolo

Implementazione corretta già esistente:

- da un vecchio sistema da reimplementare
- con algoritmo poco performante (es. brute force, complessità esponziale)

Verifico che:

```
vecchia_implementazione(input) == nuova_implementazione(input)
```

Smoke test

Prendere confidenza con un metodo di un sistema esistente:

- verificare che il risultato sia sempre non null
- restituisce valori in un range

> Qui non verifichiamo la correttezza, ma rafforziamo le nostre assunzioni

Uso avanzato di rapidcheck

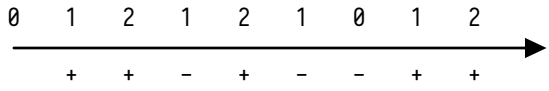
Configurazione con variabile d'ambiente RC_PARAMS:

- `seed=value` riprodurre le condizioni di test
- `reproduce=state_string` per fare debug del caso fallito
- `max_success=value` (default=100) quanti input generare

> Nel build server è utile alzare `max_success` per avere una copertura più completa

Test stateful

```
class contatore {
    int acc_ = 0;
public:
    int acc() const { return acc_; }
    void inc() { acc_++; }
    void dec() { acc_--; }
};
```



rapidcheck può generare sequenze di comandi

Domande e risposte

Grazie

> *Be lazy! Don't write tests, generate them!*

— Scott Wlaschin