TDD Calculator

Test, verifiering och certifiering

Filip Lindström
Johan Kämpe

MÖLK Utbildning

Sammanfattning

1 Inledning

1.1 Syfte

Syftet med projektet är att träna på TDD, *Test Driven Development.*
Fyra tester för en miniräknares fyra vanligaste operationer ska skapas, och funktioner ska byggas utifrån dessa fyra tester.

1.2 Länkar

Projektets GItHub-sida:
https://github.com/fille044/TDD-Calculator

2 Genomförande och resultat

2.1 Krav

Räkneoperationer som används i projektet:

- **Addition**
- **Subtraktion**
- **Multiplikation**
- **Division**

Endast heltal används för operationerna.

2.2 Utförande

2.2.1 Makefil för kompilering

En makefil skapades för att bygga programmet, antingen som ett test eller med ett "användargränssnitt".

För att bygga testet används kommandot ***make test***, annars ***make calc***.

2.2.2 Mjukvara för test

Samtliga tester är hämtade från ramverket Unity (throwtheswitch.org, 2017).
Initialt skrevs testfunktioner:

void **test_Subtraction**(void)
void **test_Addition**(void)
void **test_Division**(void)
void **test_Multiplication**(void)

De använder unity-funktionerna ***TEST_ASSERT_EQUAL*** och ***TEST_ASSERT_EQUAL_FLOAT*** för att testa miniräknarens operationer.

TEST_ASSERT_EQUAL undersöker om två heltal är lika, *TEST_ASSERT_EQUAL_FLOAT* undersöker om två flyttal är lika.

```
/* Tests multiplication in three steps */
void test_Multiplication(void)
{
    TEST_ASSERT_EQUAL(18, Multiplication(3, 6));
    TEST_ASSERT_EQUAL(81, Multiplication(9, 9));
    TEST_ASSERT_EQUAL(81096, Multiplication(124, 654));
}
```

Testfunktion för multiplikation

Initialt byggdes mock-funktioner för de fyra räkneoperationerna, som samtliga returnerade värdet 0.

```
int Subtraction(int a, int b)
{
    return 0;
}
```

Mock-funktion för subtraktion som returnerar 0

Vid kompilering och körning av testprogrammet blir då alla test *FAIL*.

```
TestCalculator.c:20:test_Addition:FAIL: Expected 36 Was 0
TestCalculator.c:28:test_Subtraction:FAIL: Expected 12 Was 0
TestCalculator.c:36:test_Multiplication:FAIL: Expected 18 Was 0
TestCalculator.c:48:test_Division:FAIL: Expected 2 Was 0
```

Funktionerna med räkneoperationer byggdes om så att testfallen uppfylldes.

```
float Division(int a, int b)
{
    return (float)a / b;
}
```

Funktion för division, som returnerar kvoten av två tal

Vid ny kompilering och körning av programmet blev då alla test *PASS*:

```
TestCalculator.c:56:test_Addition:PASS
TestCalculator.c:57:test_Subtraction:PASS
TestCalculator.c:58:test_Multiplication:PASS
TestCalculator.c:59:test_Division:PASS
```

3 Slutsats och diskussion

Testdriven utveckling används för att med större sannolikhet och självsäkerhet göra rätt direkt, och på så vis hålla en högre kvalitet under arbetets process.
Vi har i det här arbetet fått prova på just det, att redan från första raderna kod tänka i banorna kring hur vi vill att våra funktioner ska bete sig. Med hjälp av ramverket Unity har vi testat våran mjukvara med enhetstester för att säkerställa att varje funktion returnerar det värde vi från början förväntat oss.
Resultatet av att tänka testdriven utveckling har gjort att vi med relativt enkla medel kunnat producera ett säkrare användargränssnitt för en eventuell kund.

4 Referenslista

Unity. www.ThrowTheSwitch.org
Skrivet av Mark VanderVoord, Mike Karlesky, och Greg Williams. Hämtat 2017-11-15.