Laboration 4 : Enhetstestning med Catch

Metoder och verktyg i mjukvaruprojekt

Syfte:
Du ska praktiskt tillämpa enhetstestning av en klass.
Du ska praktiskt tillämpa enklare testdriven utveckling.

Innehåll

Laboration 4 – En enkel kalkylator....2
Implementation av kalkylatorn....2
Steg 1 - Infix ==> PostFix....2
Steg 2 Beräkning av uttryckets värde....3
Krav på implementationen....4
Krav på MathExpression....5
Arbetets utförande....6
Projektstruktur....6
Redovisning....7

Laboration 4 – En enkel kalkylator

I den här uppgiften ska du utveckla en enkel kalkylator som klarar de fyra räknesätten och en parentesnivå. I arbetet ska du försöka tillämpa en enklare form av TDD, dvs. ett utvecklingssätt där du driver arbetet framåt genom att skriva enhetstester innan implementationerna finns. Skriv successiva testfall som, när testen lyckas, har fört implementationen framåt. Utöka testsviten och svårigheterna successivt. Börja med ”dummy”-implementationer som gör att du klarar kompileringen men inte testen. Gör sedan enkla implementationer, låt testen misslyckas och jobba sedan vidare med implementationen tills testen klaras. När din implementation klarar alla tester går du tillbaka och ser om du kan förbättra/förenkla koden, dvs. en refactoring.

Implementation av kalkylatorn

Kalkylatorn implementeras angreppssätt är att först överföra uttrycket från infix-form till postfix-form och därefter utföra beräkningen med hjälp av RPN (Reverse Polish Notation).

Steg 1 - Infix ==> PostFix

Transformering av uttrycket från infix-form där operatorerna står mellan sina operander till postfix-form där operatorn kommer efter operanderna.

Input: infix-uttrycket	Output: postfix-uttrycket
5*(27+3*7) + 22	5 27 3 7 * + * 22 +

Metoden kan beskrivas på följande sätt:

Parsa (läs) input vänster->höger och överför till output.

- en operand (tal) överförs direkt till output

- en operator (+,*, osv) måste sparas på en stack tills dess operander har processats.

Om en operator som ska pushas på stacken har samma eller lägre prioritet än operatorn som ligger överst på stacken så poppas den operatorn av stacken och läggs till output innan den nya pushas. Detta inträffar t.ex. i uttrycket 4*6+5. När + påträffas innehåller output 4 6 och toppen på stacken innehåller operatorn *. Eftersom * har högre prioritet än + så poppas * av stacken och läggs till output innan + pushas på stacken. När den sista symbolen (5) har överförts till output så poppas det som finns på stacken och läggs till output.

Postfix-uttrycket i output blir alltså 4 6 * 5 +

Om uttrycket istället hade varit 4+6*5 så hade + varit kvar på stacken när * pushas.

Alltså:

```
4*6+5 ger output 4 6 * 5 +
4+6*5 ger output 4 6 5 * +
```

Ett annat exempel:

```
6+5-4 ger 6 5 + 4 - medan
6-5+4 ger 6 5 – 4 +
```

eftersom + och – har samma prioritet.

En vänster-parentes pushas direkt på stacken, oavsett vad som råkar ligga överst på stacken. När motsvarande höger parentes påträffas så poppas allt som ligger på stacken till output framtill vänsterparentesen. När den påträffas så poppas den av stacken men överförs inte till output.

Uttrycket (4+6)*5 kommer då att ge postfix-uttrycket 4 6 + 5 *

I postfix-formen är alla parenteserna borta och uttryckets värde kan enkelt beräknas med hjälp av en stack.

Steg 2 Beräkning av uttryckets värde.

Postfix-uttrycket behandlas från vänster till höger enligt följande

- varje operand (tal) som påträffas pushas på en stack

- när en operator påträffas poppas de två operanderna av stacken och operatorn tillämpas på

operanderna

- resultatet pushas på stacken

När postfix-uttrycket är beräknat finns resultatet i toppen på stacken (som för övrigt ska vara tom).

Ex 1.

```
Infix (4+6)*5 ==> postfix 4 6 + 5 *
```

Beräkning:

4 pushas, 6 pushas , "+" påträffas, 6 poppas, 4 poppas, 4+6 = 10 pushas, 5 pushas, * påträffas,

5 poppas, 10 poppas, 10*5 = 50 pushas. SLUT.

Ordningen på operanderna är viktig, den sist poppade blir vänster operand.

Ex 2.

```
Infix 1-2-3 ==> postfix 1 2 - 3 -
```

Beräkning:

1 pushas, 2 pushas, ”-” påträffas, 2 poppas, 1 poppas, 1-2 = -1 pushas, 3 pushas, ”-” påträffas,

3 poppas, -1 poppas -1-3 = -4 pushas. SLUT.

Krav på implementationen

Kalkylatorn ska använda sig av klassen MathExpression som implementerar följande publika interface:

```
 /*
 * Klass för lagring och beräkning av enkla matematiska uttryck.
 */
class MathExpression
{
public:
    /*
     * Skapar ett MathExpression från en sträng.
     */
    MathExpression(const std::string &expression);
    /*
     *Returnerar en sträng som är identisk med den som gavs till konstruktorn.
     */
    std::string infixNotation() const;
    /*
     * Returnerar uttrycket på postfix from
     */
    std::string postfixNotation() const;
    /*
     * Beräknar postfix uttrycket och returnerar svaret.
     */
    double calculate() const;
    /*
     * Returnerar falskt om uttrycket är ogiltigt.
     */
    bool isValid() const;
    /*
     * Returnerar ett felmeddelande om uttrycket är falskt.
     */
    std::string errorMessage() const;
    /*
     * Ersätter befintligt uttryck med expression (nollställer objektet).
     */
    MathExpression& operator=(const std::string& expression);
private:
...
};
```

Observera! Att ni får varken lägga till, ändra eller ta bort från det publika intefacet. (däremot får ni lägga till privata medlemsfunktioner och medlemsvariabler). Detta är för att testerna ska skrivas mot det publika interfacet som det är skrivet ovan.

Målsättningen är att kalkylatorn ska klara sammansatta uttryck med de fyra räknesätten och enkla parenteser. T.ex 3*(5+3*6)-5+2*(9/2-3).

Krav på MathExpression

- Efter ett anrop till konstruktion ska man anropa isValid, och endast om den returnerar sant är det definierat vad som händer när man anropar övriga metoder. Med undantag för errorMessage som ska anropas om isValid är falskt.
- Exempel på hur isValid ska tolka lite olika uttryck.

  -följande ska valideras sant:

  ```
  (32+5)-(2+1)
  5*(21+1)-1
  (2)+(3)
  2
  (35)
  ```

  -följande ska valideras falskt:

  ```
  HelloWorld!
  35+
  (1*2)(3+56)
  (2+3
  2 1 3
  2++8
  ```

- errorMessage ska skriva ut en sträng med hjälpsamma meningar om varför uttrycket inte validerades korrekt. Observera att om isValid returnerar sant ska anrop av alla metoder vara säkra och kunna anropas i godtycklig ordning (errorMessage bör returnera en tom sträng).
- Vid användning av tilldelningsoperatorn ska all internt data nollställas och det nya uttrycket valideras.
- Inga minnesläckor (använd memstat)

Till er hjälp får ni en färdig kodbas med ett färdigt interaktivt testprogram det enda ni behöver göra är att fylla i de tomma test filerna och implementations filen.

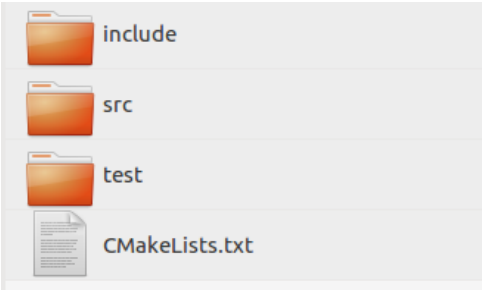
Arbetets utförande

Uppgiften kan med fördel utföras i grupp om max 3 personer. Idealet är att du samarbetar med minst en och att ni har någon kommunikationskanal så att ni i realtid kan diskutera under utvecklingens gång, dvs. tillämpa någon form av parprogrammering som är vanligt inom de agila metoderna. Exempel på olika typer av verktyg : https://trello.com/, https://www.lucidchart.com/ och https://slack.com/. Komihåg att alla i gruppen ska utföra regelbundna "Commits" så att det går att följa utvecklingen och se vem som gjort vad genom att se historiken i Git. Alla medlemmar i gruppen måste bidra med deras användarkonto, ni kan alltså **INTE** göra grupparbetet med en dator och en Bitbucket användare.

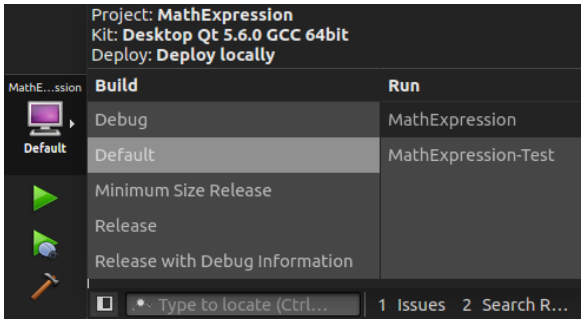
Projektstruktur

Gör en "Fork" av följande "Repository"
https://bitbucket.org/MikaelNilsson_Miun/mathexpression
Mer information om vad en "Fork" är hittar ni här :
https://confluence.atlassian.com/bitbucket/forking-a-repository-221449527.html
Det är endast en i gruppen som ska göra detta, och sedan skapa och bjuda in övriga medlemmar och metoder_och_verktyg användaren till ert team. (Tänk på lämpliga rättigheter som användarna bör ha).
Mer information om hur man skapar och administrerar ett team på Bitbucket hittar ni här :
https://confluence.atlassian.com/bitbucket/bitbucket-cloud-teams-321853005.html

Projektstrukturen skiljer sig något från tidigare projekt. Så här ser rotkatalogen ut:



I rotkatalogen hittar man byggskriptet. I include katalogen finns alla headerfiler. Src innehåller alla implementationsfiler och test katalogen innehåller alla enhetstester. Er uppgift är att implementera MathExpression.cpp och skriva enhetstester.



För att bygga och köra det interaktiva testprogrammet bygger ni MathExpression, och för att bygga och köra enhetstesterna kör ni MathExpression-Test. Testprogrammet är ett exempel på en REPL (Read-Eval-Print-Loop) och den skriver ut på formatet 'Infix' ==> 'Postfix' = 'resultat' eller ett felmeddelande vid syntax fel. Om ni kör programmet i från QT Creator måste ni köra det i en separat terminal, kryssa i Run in Terminal under Project Run.

Observera! Innan ni har implementerat alla medlemsfunktioner för MathExpression kommer det inte fungera att bygga koden verken för enhetstesterna eller testprogrammet.

Redovisning

Bjud in metoder_och_verktyg användaren till ert Team.

Utöver detta ska var och en göra en enskild rapport som ni skickar in via Moddle som ska innehålla :

- Kortfattat vilka delar av det kodmässiga arbetet som man utfört.
- En beskrivning av arbetet, hur du/ni har bedrivit arbetet och vilka verktyg som har använts (även för kommunikation)
- En diskussion om arbetssättet, resultatet och dina erfarenheter av uppgiften
- Länk till "Repository" på Bitbucket