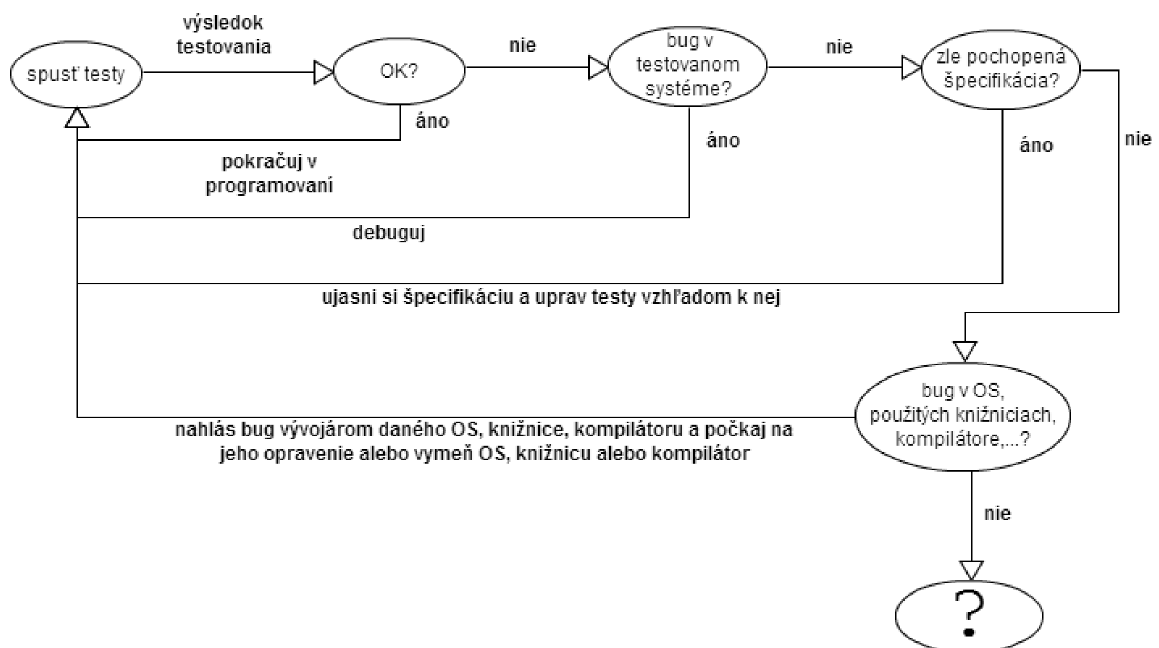
Kapitola 1

Testovanie

Moja bakalárska práca sa zaoberá testovaním výukových programov, ktorými sú jednoduché triedy a funkcie. Táto kapitola bude rozoberať metódy testovania, ktoré sa bežne používajú pri vývoji softvéru, najmä však unit testing. Kapitola čerpá hlavne z knihy The art of unit testing od Roya Osherova a z bakalárskej práce Daniela Krajča – Testovanie správnosti programového kódu v C++ pomocou unit testing frameworkov.

Ťažko by sa našiel programátor, ktorý by sa nikdy nestretol s testovaním kódu. Každý, kto programuje testuje svoj kód, aby zistil, či to, čo naprogramoval naozaj funguje tak, ako fungovať má. Pri tomto sa však ponúkajú dve dôležité otázky. Kedy program funguje tak, ako fungovať má a ako dobre odtestovať, či program funguje tak ako má. To, ako štandardne postupujeme pri testovaní popisuje nasledujúci diagram.

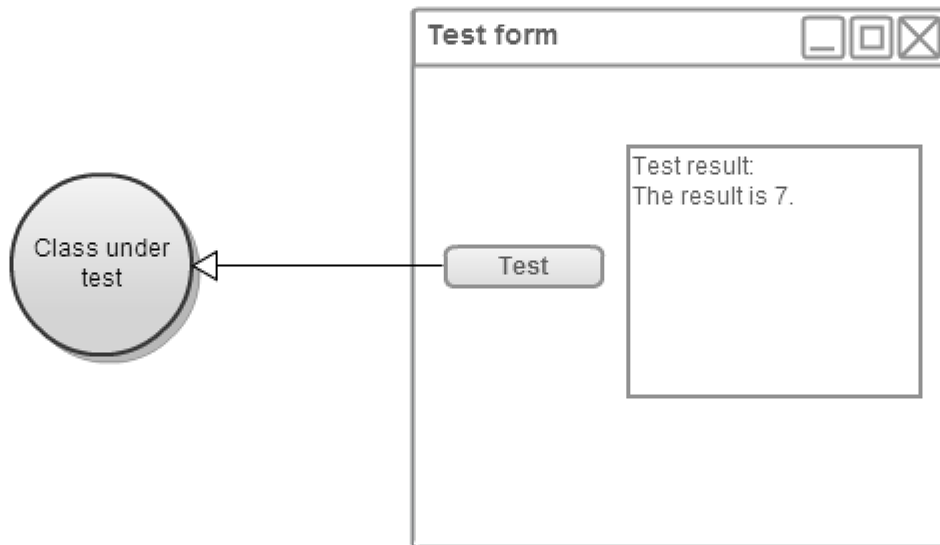


Obrázok 1.0 : Ako štandardne testujeme

Samozrejme, bežne sa spoliehame na to, že operačný systém, kompilátor a použité knižnice sú dosť spolahlivé na to, aby sme bug nehladali v nich, ale skôr vo vlastnom kóde. A toto bude aj cielom tejto bakalárskej práce. Navrhnúť a implementovať systém, ktorý bude, na základe istých vstupov od používateľa, schopný generovať testy, ktoré budú v dostatočnej miere overovať, či naprogramované funkcie a triedy pracujú tak, ako bolo zadané. Vzhľadom na to, že v tejto práci ide o testovanie konkrétnych funkcií a tried, tak použitá metóda testovania bude unit testing.

1.1 Definícia unit testingu

Unit testing nie je vo svete vývoja software žiadnou novinkou. Tento prístup k testovaniu je známy už od 70. rokov a prišiel na svet spolu s jazykom Smalltalk. Ako sa dá unit testing definovať? Unit testing je metóda testovania programového kódu pomocou unit testov. "Unit test (do slovenčiny môžeme preložiť ako test programovej jednotky) je kus kódu, štandardne metóda, ktorá volá iný kus kódu a overuje, či sa volaný kus kódu správa podľa predpokladov. Pokial sa predpoklady nepotvrdia, unit test zlyhal. Za unit je považovaná metóda alebo funkcia. Pomocou unit testingu sa testuje SUT - system under test (do slovenčiny môžme preložiť ako testovaný systém)." (Oshrova knizka, strana 4.). Je však pravdou, že programátorovi, ktorý počuje pojem unit testing v tejto chvíli prvý krát táto definícia znie ako vznešená veta, ktorá však veľa nehovorí. Pravdou však je, že každý programátor už kedysi vo svojom živote unit test napísal, či si to uvedomuje alebo nie. Každý programátor určite testoval kód ktorý práve napísal takým štýlom, že niekde v main-funkcii zavolal práve naprogramovanú funkciu alebo metódu a jej výstup nejakým spôsobom vypísal či už na konzolu, alebo do grafického formulára či, pokial sa jednalo o webovú aplikáciu, do okna prehliadača a následne skontroloval, či sa to, čo bolo vypísané, zhoduje s tým, čo očakával, že bude vypísané. Takéto testovanie bolo určite lepšie ako žiadne, a z časti sa aj spĺňalo to, čo definujeme ako unit test, pretože nejakým spôsobom bolo overované, či funkcia alebo metóda robí to, čo sme predpokladali, že robiť má, avšak stále to bolo veľmi vzdialené od toho, čo nazývame dobrý unit test. Nasledujúci diagram znázorňuje ako vačšina programátorov bežne testuje ich kód.



Obrázok 1.1: intuitívny prístup k testovaniu

1.2 Vlastnosti dobrého unit testu

Unit test by mal mať nasledujúce vlastnosti:

- Mal by byť automatizovaný a zopakovateľný
- Mal by byť jednoducho implemenovateľný
- Ak bol už raz napísaný, mal by ostať pre budúce použitie
- Hocikto z tímu vývojárov by mal byť schopný ho spustiť
- Mal by byť jednoducho spúštateľný
- Mal by bežať rýchlo

Na to aby programátor v dostatočnej miere mohol skontrolovať, či jeho unit test je dobrý, mu môže veľmi pomôcť ak sa spýta nasledujúce otázky a pravdivo si na ne odpovie:

- Môžem spustiť a prezrieť si výsledky unit testu, ktorý som napísal pred týždňom, mesiacom, či rok?
- Môže hociktorý člen týmu v ktorom pracujem spustiť a pozrieť si výsledky testu,

ktorý som napísal?

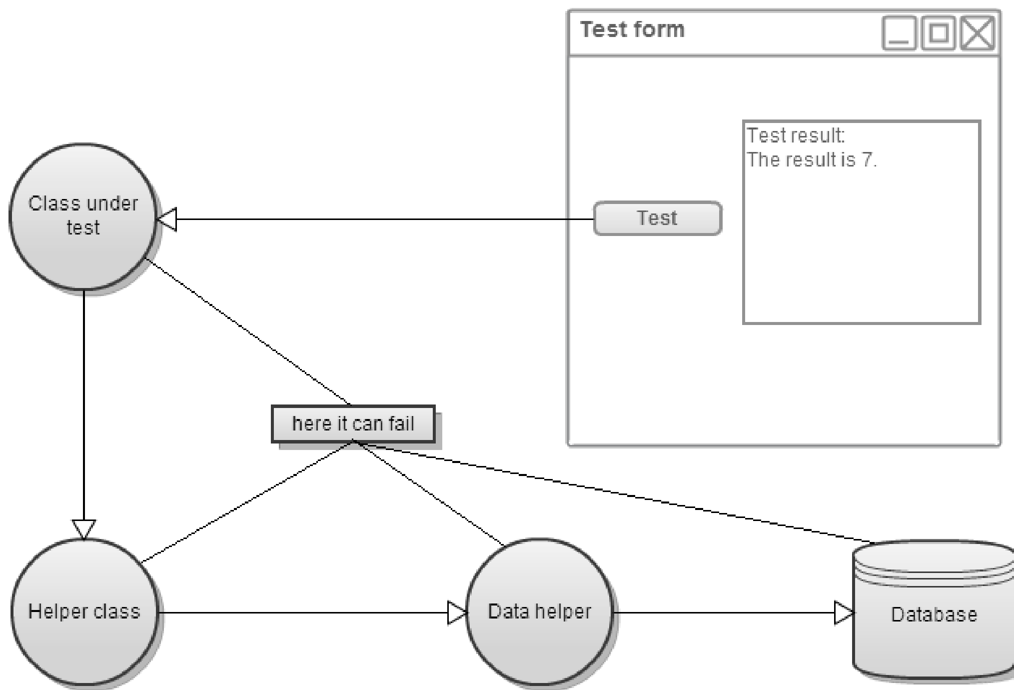
- Ak spustím všetky unit testy, ktoré som napísal, bude to trvať menej ako zopár minút? (ideálne aj menej ako zopár sekúnd)
- Môžem spustiť všetky unit testy, ktoré som napísal, na stlačenie tlačítka?
- Viem napísať nový test za menej ako zopár minút?

Ak existuje otázka, na ktorú si odpovedal "nie", je pravdepodobné, že tvoj prístup, ktorým testuješ v skutočnosti nie je unit testing ale integration testing.

1.3 Integration testing, v čom je iný?

Pokazilo sa auto. Ako zistí človek v čom je problém, ak takéto auto chce opraviť? Auto ako také sa skladá z mnohých častí, napríklad kolies, náprav, motora a iných. A tieto časti sa skladajú s ďalších menších častí. Ich integráciou získame auto, na ktorom môžeme jazdiť. Pokiaľ všetky časti fungujú ako majú, auto je v poriadku. Pokial je auto pokazené, je isté, že niektorá z týchto častí nie je v poriadku. Otázka je však, ktorá časť to je a čo treba spraviť na to, aby auto opať fungovalo tak, ako má?

Podobné to je so softvérom. Vačsina vývojárov testuje svoj kód pomocou výslednej funkčnosti či nefunkčosti užívateľského rozhrania. Kliknutím na tlačítko zavolá sériu funkcií či metód v množstve tried, ktoré majú spolupracovať a dať ten správny výsledok. Takýto prístup k testovaniu, keď sa testuje funkcionalita celej množiny unitov (modulov, tried či funkcií), ktoré majú spolupracovať a dávať výsledok sa nazýva integration testing. Pokial sa správny výsledok nedostaví, je isté že spolupráca unitov, z ktorých systém pozostáva zlyhala ako celok. Otázkou však je, čo konkrétne zlyhalo a ako to treba opraviť, aby celý systém fungoval tak, ako má. Výhodou unit testingu oproti integration testingu je, že ak testujeme každý unit zvlášť, tak pri zlyhaní konkrétneho testu vieme kde presne hľadať chybu a čo treba opraviť.



Obrázok 1.2: Diagram demonštruje, že pri integration testingu môže problém nastať na mnohých miestach.

1.4 Príklad jednoduchého unit testu

Aj keď existuje množstvo unit test framworkov, unit testy sa dajú písať a spúšťať aj bez nich. V tejto časti sa pozrieme na to, ako by taký unit test mohol vyzerať. Predpokladajme, že naším zadaním je naprogramovať jednoduchú funkciu, ktorá má brať na vstupe reťazce "one", "two" a "three" a vrátiť zodpovedajúce celé číslo. Po ťažkých hodinách presedených nad kódom sa nám podarilo napísať tento kód:

Kód 1.4.1: parseNumber.cpp

```
#include "numberParser.h"

int parseNumber(string numStr){
	if(numStr == "one"){
		return 1;
	}
	else if(numStr == "two"){
		return 2;
```

```
	}
	else{
		return -1;
	}
}
```

V tomto momente pre ujasnenie pojmov je dobré si pripomenúť, že funkcia **int parseNumber**(string numStr) ktorú sme práve naprogramovali je presne taký unit (programová jednotka), o akom sa celý čas bavíme, takže bude vhodné aby sme ju odtestovali pomocou unit testingu. Testovací kód bude vyzerať nasledovne:

Kód 1.4.2: parseNumberTests.cpp

```
#include "numberParser.h"
#include <iostream>

int main(){
	bool allTestsOk = true;

	if(parseNumber("one") != 1){
		allTestsOk = false;
		cout << "one should be parsed as 1" << endl;
	}
	if(parseNumber("two") != 2){
		allTestsOk = false;
		cout << "two should be parsed as 2" << endl;
	}
	if(parseNumber("three") != 3){
		allTestsOk = false;
		cout << "three should be parsed as 3" << endl;
	}

	if(allTestsOk){
		cout << "Final result - " << "All tests run 
successfully" << endl;
	}
	else{
		cout << "Final result - "  << "Something went wrong" << 
endl;
	}

	return 0;
}
```

Po tom, ako spustíme naše testy, zistíme, že výsledok testovania je nasledovný:

```
three should be parsed as 3
Final result - Something went wrong
```

Z čoho nám je jasné, že reťazec "three" parsujeme spôsobom, ktorý nezodpovedá zadaniu úlohy, takže vieme kde presne hľadať chybu a opraviť ju nebude ťažké. V tomto učebnicovom prípade by to samozrejme nebolo ťažké ani bez unit testov, ale v praxi sa stretávame veľmi často so situáciami, keď hodiny sedíme nad kódom, ktorý z nejakého dôvodu nepracuje tak, ako pracovať má a urputne hľadáme to miesto, ktoré spôsobuje chybu. Takýmto spôsobom testovania budeme však schopný jdnoznačne a rýchlo identifikovať miesto, ktoré chybu spôsobuje a čas budeme môcť venovať na opravenie logiky kódu, ktorý je chybný.

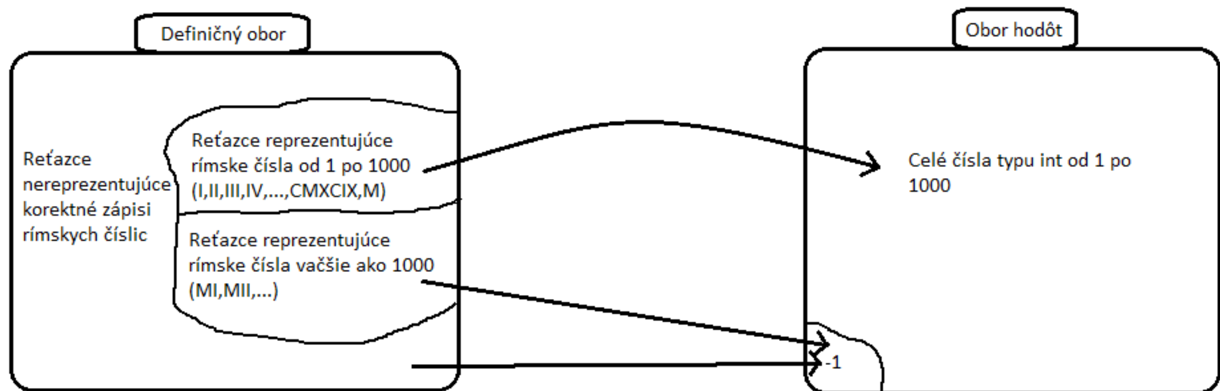
1.5 Čo brať do úvahy pri písaní testov

Keď príde na písanie testov, prichádza aj otázka, čo všetko treba odtestovať na to, aby sa dalo povedať, že program funguje správne. Najdôležitejšie, čo si musíme uvedomiť, že to čo testujeme, sú funkcie vo veľmi podobnom zmysle v akom ich poznáme v matematike a teda majú nejaký definičný obor a zároveň možné návratové hodnoty tvoria istý obor hodnôt tejto funkcie. Ako príklad uvažujme, že úlohou je naprogramovať funkciu, ktorá má previesť číslo zapísané v rímskych čísliciach na integer. Celá špecifikácia vraví, že funkcia má byť schopná prevádzať čísla, ktorých hodnota je 1 až 1000 a pri vstupe, ktorý buď nie je korektne zapísané rímske číslo, alebo je korektne zapísané číslo mimo tohoto rozmedzia má vracať hodnotu -1. Nech hlavička tejto funkcie vyzerá nasledovne:

```
int fromRomanToInt(string romanNo).
```

Teda definičným oborom tejto funkcie by mala byť množina všetkých stringov a oborom hodnôt by mala byť množina všetkých prirodzených čísel od 1 po 1000 spolu s číslom -1. Čo sa týka definičného oboru, vzhľadom na to, že vačšina programovacích jazykov je typovaná, a pri písaní tej ktorej funkcie v hlavičke funkcie uvádza programátor typ jej parametrov, tak definičný obor funkcie bude vždy množina všetkých kombinácií hodnôt, ktoré sú pre dané parametre prípustné. Už v prípade našej funkcie fromRomanToInt je jasné, že odtestovať všetky možné vstupy nie je reálne, kedže množina všetkých reťazcov je nekonečná, alebo limitovaná jedine dostupnou pamaťou v počítači, čo je stále príliš veľa

možných kombinácií znakov. Ako teda vybrať vhodnú podmnožinu z definičného oboru, ktorá bude dostatočne reprezentatívna na to, aby odtestovala túto funkciu? V prvom rade si musíme uvedomiť, že náš obor hodnôt je vzhľadom na našu funkciu nejak logicky rozdelený. Ako, to ilustruje nasledujúci obrázok:



Obrázok 1.5. : logické rozdelenie definičného oboru funkcie fromRomanToInt

Logicky môžeme rozdeliť definičný obor funkcie fromRomanToInt na tri časti a to na reťazce reprezentujúce rímske čísla od 1 po 1000, reťazce reprezentujúce rímske čísla vačšie ako 1000 a reťazce, ktoré nereprezentujú korektné zápisy rímskych číslic. Pre každú logickú podmnožinu definičného oboru predpokladáme nejaké správanie, charakteristické pre danú podmnožinu. To, čo robíme pri testovaní je, že sa snažíme vybrať zopár reprezentantov z každej logickej podmnožiny definičného oboru, zavolať funkciu, ktorá dostane týchto reprezentantov ako parameter a následne overiť, či výsledok je taký, ako určuje špecifikácia. To, kto sú vhodný reprezentanti je skôr filozofická ako technická debata, ale predsa je tu ešte pár myšlienok, na ktorých sa vačšina programátorov zhodne, že je dobré, ak sa aplikujú. A to, že funkciu treba testovať vzhľadom na vstupy ktoré sú:

- **očakávané**
  vstupy, ktoré sú bežné vzhľadom na špecifikáciu funkcie. Sú to vstupy, pri ktorých funkcia naozaj vykonáva to, pre čo bola programované. Napríklad naša funkcia fromRomanToInt bola programovaná pre to, aby prevádzala znakové reťazce na celé čísla a očakávanými vstupmi sú všetky reťazce reprezentujúce rímske čísla.

Ako reprezentantov by sme mohli vybrať napríklad reťace "III","DCCCXX","MMCD".

- **podobajúce sa na očakávané**
  vstupy, ktoré sú nejakým spôsobom podobné s očakávanými vstupmi. Pri funkcii fromRomanToInt by to boli reťazce, ktoré síce nepredstavujú korektné zápisi rímskych číslic, ale obsahujú iba korektné rímske číslice, napríklad reťazce "DDCM","IIIXVD","IXXDCC".
- **hraničné**
  vstupy, ktoré sú na hranici dvoch logických podmnožín definičného oboru. Hraničné hodnoty na hraniciach medzi nekorektnými zápismi rímskych číslic a korektnými zápismi je ťažko určiť, ale hraničnými hodnotami na hraniciach medzi zápismi číslic od 1 po 1000 a zápismi číslic vačších ako 1000 sú celkom jasne reťazce "M" a "MI". Tieto reťazce je dobré uvažovať ako vstupy pre testy
- **špeciálne**
  okrem hraničných hodnôt je dobré uvažovať aj špeciálne hodnoty vstupov. Obvykle sú špeciálne hodnoty malou podmnožinou množiny očakávaných hodnôt. Pri našom príklade môžeme uvažovať ako špeciálne tie reťazce, ktoré tým, že majú vo svojom zápise menšiu číslicu pred vačšou naznačujú, že pri prevádzaní treba hodnotu menšej odčítať od hodnoty vačšej na to aby sme dostali správny výsledok. Sú to napríklad reťazce "IX","CXLIX","XIV". Za špeciálnu hodnotu môžeme takisto považovať pri reťazcoch prázdny reťazec, pri parametroch číselného typu hodnoty 0,1,-1 a hodnoty typu POSITIVE_INFINITY a NEGATIVE_INFINITY a pri parametroch typu double a float hodnoty typu Not a Number.
- **zlé vstupy**
  za zlé vstupy považujeme také vstupy, ktoré funkcia nemá vedieť nijak zmysluplne spracovávať, iba nejakým spôsobom má informovať o tom, že tento vstup nevie spracovať a to buď vrátením špeciálnej hodnoty, ako v prípade funkcie fromRomanToInt alebo hodením výnimky. Vždy je dobré odtestovať aj správanie funkcie pri zlých vstupoch. V prípade našej funkcie sú zlými vstupmi všetky reťazce, ktoré nereprezentujú korektný zápis rímskeho čísla. Napríklad sú to

reťazce "Jozef","xdekxczep","slovenske mamicky77". Vymýšľať sa ich dá naozaj do bludu.

Na základe tohoto rozdelenia vstupov a pri dobrom výbere reprezentantov jednotlivých kategórií sme už schopný napísať testy, ktoré v dostatočnej miere odtestujú, či nami naprogramovaná funkcia spĺňa špecifikáciu.

1.6 Unit testing frameworky

V predchádzajúcom príklade sme ukázali, ako sa dá triviálne testovať funkcia, ktorá vracia int a následne ako testovací kód zjednodušiť a spraviť ho prehľadnejším. Lenže v reálnom svete toho budeme potrebovať testovať o mnoho viac a písať si vlastné triedy na testovanie by nebola zrovna najpraktickejšia voľba. Tu nám prichádzajú na pomoc unit testing frameworky, ktoré implementujú množstvo rutín potrebných pre unit testing a tak pomáhajú programátorom s unit testingom. Prvý unit testing framework implementoval Kent Beck pre jazyk Smalltalk. Framework dostal meno SUnit (zo SmalltalkUnit). Neskôr boli na rovnakej architektúre implementované frameworky pre veľa bežne používaných jazykov. JUnit pre Javu, CppUnit pre C++ a mnohé iné. Vďaka ich pomenovaniu dostala tátot architektúra pomenovanie xUnit architektúra a všetky takéto frameworky sa súhrnne označujú ako xUnit.

1.6.1 xUnit architektúra

Všetky frameworky postavené na xUnit architektúre zdieľajú tieto základné komponenty:

- **Assert**
  V každom xUnite by mal byť programátor schopný assertov. Assert je procedúra, ktorá sa štandardne pri implementácii testovacích metód používa na volanie konkrétnych testov, ktoré ak vrátia true, tak program ide ďalej a nič sa nedeje, ale ak nejaký test vráti false, tak sa daná testovacia metóda, v ktorej sa assert nachádza, ukončí a nejakým spôsobom, či už výpisom na konzolu, do súboru, alebo inak, je programátorovi oznámené ktorá procedúra a na ktorom teste zlyhala.
- **Test case**

Najzákladnejšia trieda, ktorá je určená na implementáciu jedného jednoduchého testu, tak, že sa definuje jej potomok a overriduje metóda runTest(). Štandardne sa však veľmi nepoužíva, skôr sa používa trieda Test fixture.

- **Test fixture**
  Trieda, pomocou ktorej sa štandardne implementuje jadro unit testov, teda konkrétne asserty. Kedže funkcie alebo triedy, ktoré testujeme potrebujú k svojej funkčnosti často množstvo inštancií iných tried, Test fixture poskytuje možnosť nastaviť kontext, v ktorom budú jednotlivé testy bežať. Štandardne sa teda definuje potomok triedy Test fixture v ktorom sa zadefinujú metódy, ktoré slúžia ako konkrétne testy, teda v nich sú volané konkrétne asserty a navyše sa v tejto triede overridujú metódy setUp() a tearDown(). Metóda setUp() funguje tak, že kód v nej sa vykoná pred každým volaním metódy, ktorá implementuje konkrétne asserty, napríklad vytvorí objekty potrebné k spusteniu testov, následne sa zavolá metóda, ktorá implementuje konkrétne asserty a po jej ukončení sa zavolá metóda tearDown(), ktorá nejakým spôsobom uprace všetko, čo chceme aby bolo po testovaní dané do pôvodného stavu, napríklad zruší objekty vytvorené pomocou metódy setUp().
- **Test runner**
  Trieda, ktorej úlohou je spúšťat testy a informovať programátora o výsledkoch testovania. Štandardne sa používa tak, že cez metódu addTest() sa inštancii triedy Test runner posunú referencie na metódy implementujúce testovanie v potomkovi triedy Test Fixture, inštancia triedy Test runner následne postupne spúšťa tieto metódy, pričom pred každým volaním testovacej metódy zavolá metódu setUp(), definovanú pre daného potomka Test Fixture, ktorého testovacia metóda sa volá a po skončení testovacej metódy zavolá metódu tearDown(). Po tom ako skončia všetky testovacie metódy je programátor informovaný o výsledkoch testovania.
- **Test suite**
  Ako už názov napovedá (po anglicky suite znamená súprava), táto trieda je abstrakciou pre krabicu obsahujúcu viacero testov a slúži na sprehľadnenie testovacieho kódu. Jej inštancie dokážu v sebe uchovávať informácie o viacerých

testovacích metódach. Štandardne sa používa tak, že v potomkovi Test Fixture definujeme metódu, ktorá vracia inštanciu tejto triedy, pričom do nej vložíme referencie na všetky testovacie metódy, ktoré sme v potomkovi Test Fixture implementovali a následne sa táto inštancia posunie inštancii triedy Test runner a sa už postará o to, aby všetky testy zbehli a programátor bol informovaný o ich výsledkoch.

V tejto baklárskej práci bude použitý framework CppUnit, ktorý je tiež postavený na xUnit architektúre. Hlavnou výhodou CppUnitu je, že výstup z testovania môže byť aj vo formáte XML, ktorý využijem vo webovej časti tejto aplikácie.

1.7 Podobné systémy

Systém, ktorý inšpiroval túto bakalársku prácu bol naprogramovaný v rámci bakalárskej práce Daniela Krajča - Testovanie správnosti programového kódu v C++ pomocou unit testing frameworkov a umožňuje zadávateľovi cvičenia cez webové rozhranie vytvoriť testy, ktoré budú testovať správnosť vypracovaného cvičenia. Následne môže študent, ktorý naprogramoval dané cvičenie nahrať súbor so svojím zdrojovým kódom na server, kde je tento kód skompilovaný a odtestovaný, pričom je študent informovaný o tom, ako testy dopadli. Testy však niesú nijak generované a zadávateľ je nútený každý jeden test explicitne zadať. Systém, ktorý bude výsledkom tejto práce by mal byť schopný na základe istého množstva vstupných údajov, ako napríklad hlavičky funkcií, ktoré majú byť testované, informácie o logickom rozdelení definičného oboru funkcie a podobných, automaticky generovať testy pre dané zadanie a následne by mal byť takisto schopný pomocou vygenerovaných testov odtestovať vypracované zadanie. Študentské zadania budú v tomto systéme provnávané s master-verziou zadania, ktorú bude musieť vyučujúci pri vytváraní cvičenia vypracovať a jej zdrojový kód pri nahrávaní cvičenia do systému odovzdať spolu s ostatnými potrebnými informáciami k cvičeniu. Krajčov systém takisto neimplementuje meranie efektívnosti jednotlivých riešení. Tento systém bude schopný merať čas behu jednotivých riešení a teda študent bude môcť vidieť aké efektívne je jeho riešenie v provnaní s ostatnými odovzdanými riešeniami.