Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE

Tomáš Foltýnek

Systém pro automatické a poloautomatické testování softwarových modulů

Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: Mgr. Vladan Majerech, Dr.
Studijní program: Informatika, programování

2007

Na tomto místě bych rád poděkoval Mgr. Vladanu Majerechovi, Dr. za vedení mé práce, za rady a čas, který mi během vypracovávání této práce věnoval.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 31.5.2007 Tomáš Foltýnek

Obsah

1 Úvod **6**
2 Test-Driven development **7**
2.1 Předpoklady TDD 7
2.2 Vývojový cyklus TDD 8
2.3 Výhody TDD 10
2.4 Omezení TDD 10
3 Tutoriál **11**
3.1 Zadání 11
3.1.1 Předpoklady 11
3.2 První test 12
3.2.1 Nejdříve test 12
3.2.2 Počáteční implementace 13
3.2.3 Spuštění testu a oprava implementace 13
3.3 Druhý test a úprava testovací třídy 16
3.3.1 Druhý test 16
3.3.2 Úprava testovací třídy 18
3.4 Chybové stavy 19
4 Funkcionalita převzatá z nástroje NUnit **22**
4.1 Třída Assert 22
4.1.1 Kontrola jednoduchých podmínek 23
4.1.2 Kontrola rovnosti a nerovnosti 23
4.1.3 Kontrola identity 24
4.2 Atributy 24
4.2.1 Atribut TestFixture 24
4.2.2 Atribut Test 25
4.2.3 Životní cyklus testovací třídy 25
4.2.4 Atribut SetUp 27
4.2.5 Atribut TearDown 27
4.2.6 Atribut TestFixtureSetUp 28
4.2.7 Atribut TestFixtureTearDown 28
4.2.8 Atribut ExpectedException 28
4.2.9 Atribut Explicit 30
4.2.10 Atribut Ignore 31
4.2.11 Atribut Category 32
5 Nová funkcionalita přidaná nástrojem FastUnit **34**
5.1 Testování výkonu modulů 34
5.1.1 Motivace 34
5.1.2 Uvažovaná řešení 35
5.1.3 Atribut Duration 36
5.1.4 Omezení 37
5.2 Parametrizovatelné testy 37
5.2.1 Motivace 37
5.2.2 Uvažovaná řešení 39
5.2.3 Atribut Data 40
5.2.4 Atribut Data a očekávání výjimky 41
5.2.5 Atribut Data a testování výkonu modulů 42

5.2.6 Příklad využití parametrizovatelných testů 43
5.3 Paralelní testy 43
5.3.1 Motivace 43
5.3.2 Uvažovaná řešení 45
5.3.3 Atribut ParallelTest 47
5.3.4 Atribut ParallelTest a životní cyklus testovací třídy 48
5.3.5 Atribut ParallelTest a parametrizovatelné testy 50
6 Uživatelská dokumentace 52
6.1 Aplikace s grafický uživatelským prostředím 52
6.1.1 Menu a nástrojová lišta 52
6.1.2 Průzkumník testů 53
6.1.3 Výsledkový panel 54
6.1.4 Stavový řádek 55
6.1.5 Parametry spuštění 56
6.2 Konsolová aplikace 56
6.2.1 Parametry 57
7 Programátorská dokumentace 58
7.1 Jádro 59
7.1.1 Základní vlastnosti jádra 59
7.1.2 Rozdělení jádra 60
8 Porovnání s konkurenčními nástroji 65
8.1 NUnit 65
8.2 Zanebug 65
8.3 TestDriven.NET 66
9 Závěr 67
10 Literatura 68

Název práce: Systém pro automatické a poloautomatické testování softwarových modulů
Autor: Tomáš Foltýnek
Katedra: Katedra teoretické informatiky a matematické logiky
Vedoucí bakalářské práce: Mgr. Vladan Majerech, Dr.
E-mail vedoucího: Vladan.Majerech@mff.cuni.cz

Abstrakt: Prostudovat dostupnou literaturu o TDD (Test-Driven Development). Seznámit se s existujícími systémy používanými při aplikaci TDD, zejména se systémem NUnit (www.nunit.org).
Navrhnout a implementovat systém pro testování softwarových modulů, při návrhu dosáhnout zpětné kompatibility se systémem NUnit 2.2. Systém musí dále nabídnout podporu pro testováni modulů určených pro běh v paralelním prostředí, pro testování výkonu modulů a pro detailnější analýzu chybových stavů modulů.
Implementovat konsolovou aplikaci umožňující spouštění zadaných testů na automatické bázi. Výstupem aplikace budou výsledky testů uložené v XML formátu, tak aby je bylo možno dále zpracovávat.
Implementovat aplikaci umožňující uživateli interaktivní práci s vybranými testy. Aplikace nabídne uživateli dostupné testy, umožní spuštění jednotlivých testů či množiny testů a graficky prezentuje výsledky spuštěných testů.
Text práce by měl obsahovat uživatelskou i technickou dokumentaci, k práci musí být přiloženy vytvořené moduly ve formě zdrojových souborů v jazyce C#.

Klíčová slova: Test-Driven Development, testování modulů, FastUnit

Title: Framework for the automated and semi-automated testing of software components
Author: Tomáš Foltýnek
Department: Department of Theoretical Computer Science and Mathematical Logic
Supervisor: Mgr. Vladan Majerech, Dr.
Supervisor's e-mail address: Vladan.Majerech@mff.cuni.cz

Abstract: This work is a research on Test-Driven Development (TDD), the existing literature about TDD and techniques used for implementing TDD with particular focus on the system NUnit (www.nunit.org). The primary goal was to design and implement system suitable for testing software modules, which would be backward compatible with NUnit 2.2. The designed system should implement support for testing modules that run in parallel environment, performance testing and detailed analysis of errors in modules.
The work includes a console application that allows running selected tests in automated fashion. Test results as an output of this application are stored in the XML format suitable for further processing. The work also includes a GUI application that gives users interactive access to tests. This application offers a list of available tests, lets users to run selected individual tests or their groups and presents results of running tests in the graphical form. The work should include the user guide, technical specification of the designed system and the source code of all implemented modules in the language C#.

Keywords: Test-Driven Development, unit testing, FastUnit

1 Úvod

Snem každého programátora je vyvíjet kód, který je čistý a neobsahuje chyby. Kód, který se postupem času neustále zlepšuje a zdokonaluje a v průběhu vývoje projektu je s ním čím dál větší radost pracovat. Ve většině případu je však opak pravdou. Zdá se, že nezáleží na tom, jak velké úsilí programátor při své práci vynaloží, dříve či později se kód změní v nepřehledné bludiště a každý další zásah do takového kódu se stává velice obtížným. Čím větší je projekt, tím se problém ještě znásobuje.

Test-Driven Development (TDD) upravuje proces vývoje kódu způsobem, kdy změny kódu jsou nejen možné, ale i dokonce žádoucí. Proces vývoje kódu se v podání TDD skládá ze tří základních aktivit: 1) vytvoření testu, 2) implementace funkcionality požadované testem a 3) úpravy kódu za účelem jeho zjednodušení, zprůhlednění a odstranění duplicit.

Zmíněné tři aktivity se neustále opakují. V každém kole jsou spouštěny všechny existující testy tak, aby se zajistilo, že veškerá doposud implementovaná funkcionalita je zachována. Jsou odstraněny dlouhé prodlevy mezi fázemi designu, implementací a testováním. To vše má v konečném důsledku pozitivní dopad na kvalitu designu i vlastního kódu v průběhu celého životního cyklu projektu.

To, co dělá TDD tak efektivní, je *automatizace* testů. Ve světě existuje celá řada nástrojů, které jsou dostupné zdarma a které nabízejí plnou podporu pro vývoj i automatizované provádění testů, přičemž se nejedná o žádné odlehčené verze komerčních produktů, ale o velice kvalitní nástroje vyvíjené komunitami vývojářů jako *open source* software. Nástroje jsou dostupné pro celou řadu programovacích jazyků: *Java*, *C#*, *C++*, *Perl*, *Smalltalk* a další.

V rámci této práce vznikl nástroj FastUnit. Důvodem vzniku nástroje FastUnit a návazně i celé této práce byly praktické zkušenosti autora s již existujícím nástrojem NUnit [4] při vývoji softwaru pomocí zásad TDD. Nástroj NUnit je velice užitečným nástrojem, nicméně se při jeho každodenním používání objevily některé jeho nedostatky související zejména s chybějící podporou pro tvorbu složitějších testů. Cílem nástroje FastUnit je nahradit nástroj NUnit, odstranit některé jeho nedostatky a nabídnout chybějící funkcionalitu. Byla přidána funkcionalita umožňující 1) testováni modulů určených pro běh v paralelním prostředí, 2) testování výkonu modulů a 3) použití jednoho testu pro testování více vstupních hodnot. Z důvodu existence již velkého množství testů napsaných pro nástroj NUnit byl kladen důraz na zachování zpětné kompatibility s tímto nástrojem.

Druhá kapitola této práce je věnována technice TDD, kterou se snaží přiblížit. Základní principy TDD jsou prezentovány ve třetí kapitole na příkladech vývoje jednoduchých testů za pomoci nástroje FastUnit. Čtvrtá kapitola obsahuje podrobný popis funkcionality implementované nástrojem FastUnit z důvodu zachování zpětně kompatibility s nástrojem NUnit. Pátá kapitola je věnována nové funkcionalitě přidané nástrojem FastUnit. Následuje šestá a sedmá kapitola, které obsahují uživatelskou a programátorskou dokumentaci. Závěrečná část práce je věnována srovnání nástroje FastUnit s konkurenčními nástroji.

2 Test-Driven development

O testování je přemýšleno jako o únavné a druhořadé záležitosti, kterou je možné vykonávat až v samém závěru vývojového cyklu projektu, kdy už je veškerý kód napsán a je nutné zkontrolovat jeho správnost. V situacích, kdy se projekt nevyvíjí podle očekávání a blíží se termíny odevzdání, je to právě testování, které zpravidla nejvíce utrpí. Jako poslední fáze vývojového cyklu je tedy zkráceno na minimum tak, aby byly dodrženy dohodnuté termíny.

Jednou z alternativ k tomuto způsobů myšlení je technika, ve které je testování nedílnou součástí vývojové fáze projektu. Technika, pomocí které testování nabízí prostředky pro zlepšení designu softwaru a umožňuje jeho rychlejší vývoj. Tato technika je známá jako Test-Driven Development (TDD).

Test-Driven Developement je úzce spjat s metodologiemi Extrémního programování [11], [12]. Extrémní programování jsou metodologie, ve kterých je testování a komunikace v samém středu procesu vývoje projektu. Programátoři provádějí minimální design na začátku vývoje a uvolňují kód ve více verzích a ve velmi krátkých intervalech. Časté uvolňování kódu dovoluje, aby případné chyby byly nalezeny již v počátcích projektu, kde množství práce potřebné na jejich odstranění je minimální. Design softwaru je vylepšován nepřetržitým prováděním úprav kódu v rámci postupného vývoje. V momentě kdy je zjištěno, že design neodpovídá kladeným požadavkům, je vyhrazen čas na úpravu kódu do více přehlednější, užitečnější a flexibilnější formy.

Základní myšlenkou TDD je, že než odkládat testování na konec vývojového cyklu projektu, je lepší ho přesunout na jeho začátek. Předtím než programátoři začnou vyvíjet novou funkcionalitu, navrhnou a vytvoří nejprve testy, které popisují požadovanou funkcionalitu. Až poté co jsou testy napsány, je vyvíjena vlastní funkcionalita do doby, než všechny testy budou splněny. Vývoj se odehrává ve velice rychlých iteračních cyklech obsahujících vývoj, testování a úpravu kódu.

2.1 Předpoklady TDD

Aby bylo možné uplatnit přístup požadovaný TDD je nutné, aby testování bylo úplné, bezchybné a rychlé. To obecně vyžaduje dostupnost testovacího nástroje. Pro zvýšení efektivity TDD jsou od testovacího nástroje požadovány následující vlastnosti:

1. **Snadné vytváření testů** – Programátoři budou pravděpodobněji psát testy, jestliže jejich vytváření bude jednoduché a přímočaré. Také manažeři nebudou bránit vytváření testů, pokud to bude snadná a hlavně rychlá činnost.
2. **Definice kriteria splnění testů** – Nástroj musí definovat jednoduchá kritéria, podle kterých lze určit, zda test skončil úspěšně nebo s chybou. Veškerou práci s vyhodnocováním výsledku testu musí provádět nástroj automaticky. Výsledkem, zda test skončil úspěšně, je pouze odpověď typu *ano/ne*. Není například možné, aby programátor byl nucen analyzovat výstup testu a určovat výsledek testu ručně.
3. **Spolehlivost** – Test musí skončit s chybou, pokud je něco skutečně nefunkční. Nástroj musí být spolehlivý. Opakované provedení testů musí vést ke stejným výsledkům.

4. **Snadná opakovatelnost testů** – Testy musí být možné spouštět opakovaně, jinak velice rychle ztrácí na svém významu.
5. **Rozšiřitelnost** – Testovací kritéria musí být jednoduše rozšiřitelná tak, aby je bylo možné rozšířit i pro kontrolu specifických požadavků.

2.2 Vývojový cyklus TDD

Následující popis vývojového cyklu TDD je převzat z knížky *Test-Driven Development By Example* [1], která je mnohými považována za původní zdroj konceptu TDD v jeho nynější podobě.

1. **Přidání testu**

   V TDD vývoj každé nové funkcionality začíná napsáním testu. Tento test musí skončit s chybou, neboť je napsán dříve než jím testována funkcionalita. Aby mohl být test napsán, je nejdříve nutné pochopit specifikaci a požadavky přidávané funkcionality.

   Na test lze pohlížet jako na jednu z forem specifikace. V každém kroku lze implementovat pouze funkcionalitu požadovanou testy a pracovat pouze na jednom testu v daném čase.

2. **Spuštění testů a ověření, že přidaný test skončil s chybou.**

   Spuštění testů a ověření, že přidaný test skončil s chybou potvrzuje, že kontroly prováděné během testu pracují správně a že test omylem neskončí úspěšně aniž by vyžadoval přidání kontrolované funkcionality.

   Nový test musí taktéž skončit s očekávanou chybou. Tento krok slouží pro kontrolu testu samotného.

3. **Implementace funkcionality**

   V tomto kroku je implementována nová funkcionalita, která způsobí, že test skončí úspěšně. Kód přidaný v tomto kroku nemusí být úplně propracovaný a čistý, ale musí splňovat požadovanou funkcionalitu. V této fázi je to akceptovatelné, neboť kód bude v jednom z dalších kroků dále upraven a vylepšen.

   Je důležité, aby napsaný kód byl navržen tak, aby implementoval pouze požadovanou funkcionalitu. Žádná další a tím pádem netestovaná funkcionalita by neměla být přidávána v žádné fázi vývoje.

   Poznámka autora: V praxi je v některých případech velice těžké nebo téměř nemožné předchozí pravidlo dodržet. Jedním z důvodů může být neoddělitelnost implementovaných funkcionalit, kdy vývojem jedné funkcionality je automaticky přidána i další. Dalším důvodem může být například efektivita práce. Ve všech případech je však nutné, aby v následujících cyklech byly doplněny chybějící testy.

4. **Spuštění testů a ověření, že přidaný test skončil úspěchem.**

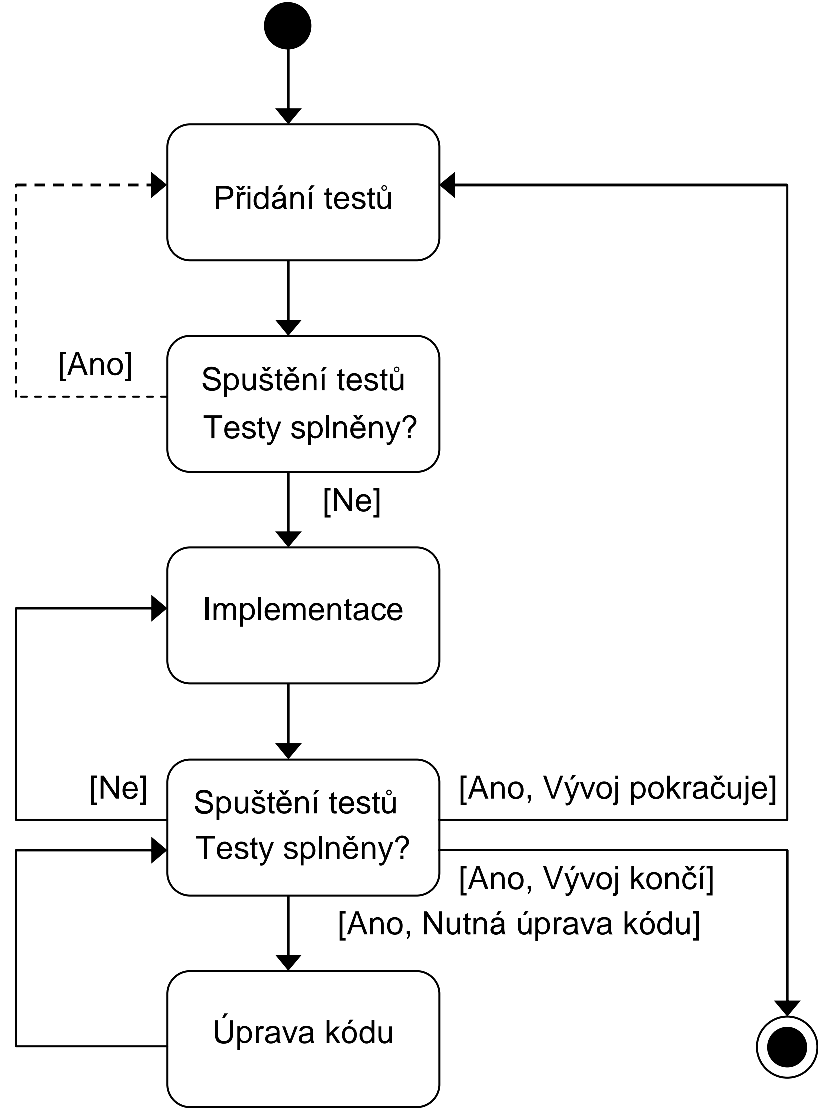
   Jestliže všechny existující testy skončí úspěšně, může si být programátor jist, že kód implementuje všechnu požadovanou funkcionalitu. Všechny testy končící úspěchem je podmínka pro závěrečný krok celého vývojového cyklu. V opačném případě je nutné se vrátit do bodu 3.

5. **Úprava kódu**

V této fázi může být kód upraven, jak je nezbytně nutné. V tomto kroku jsou doporučeny zejména úpravy kódu za účelem jeho zjednodušení, zprůhlednění a odstranění duplicit. Opakováním spuštění testů se programátor může ujistit, že úpravou kódu nebyla porušena žádná existující funkcionalita.

Cyklus je následně opakován, implementace další funkcionality je opět zahájena přidáním jejího testu. Velikosti jednotlivých kusů, po kterých je přidávána nová funkcionalita, mohou být libovolně velké. Mohou být tak malé jak programátor preferuje, nebo naopak velké, pokud se programátor cítí sebevědomě. Jestliže se však vývoj kódu vyhovující napsanému testu nevyvíjí dobře a dostatečně rychle, je možné, že zvolené množství nové funkcionality bylo příliš velké. V tomto případě je vhodné rozdělit novou funkcionalitu do více částí a ty vyvíjet a testovat samostatně.

Jednotlivé kroky vývojové cykly TDD jsou pro názornost uvedeny ve formě UML diagramu (obr. 2.1).



Obrázek 2.1: *Test-Driven Development* ve formě UML diagramu.

2.3 Výhody TDD

TDD může pomoci vyvinout software rychleji a lépe. Tato technika nabízí více než jenom možnost validace správnosti implementace, ale také řídí a směruje návrh a architekturu softwaru. Tím, že vývoj funkcionality začíná tvorbou testů, je programátor donucen přemýšlet a uvažovat o tom, jak bude daná funkcionalita používána. V první fázi je tedy programátor zaujat tvorbou správného rozhraní a nikoliv implementací.

TDD nabízí možnost postupného vývoje softwaru v malých krocích. Umožňuje programátorovi se zaměřit na jednotlivé úkoly samostatně. Hlavním cílem je vždy završit implementaci nové funkcionality úspěšným provedením testu. Umožňuje taktéž programátorovi se zaměřit na hlavní stěžejní části funkcionality - okrajové případy a ošetřování chyb nemusí být zpočátku bráno v úvahu. Testy pokrývající tyto vedlejší situace jsou implementovány odděleně.

TDD zajišťuje, pokud je používáno správně, že veškerý napsaný kód je pokrytý testy. To může dát programátorovi a následně i uživatelům velkou důvěru v daný kód.

Je sice pravda, že s použitím TDD je nutné napsat mnohem více kódu než bez použití TDD (díky kódu testů), ale celkový čas na vývoj softwaru je typicky kratší. Velké množství testů pomáhá minimalizovat počet chyb ve výsledném kódu. Tvorba množství testů a jejich velice časté opakování pomáhá odhalit chyby již v počátečních fázích vývojového cyklu a brání jejich rozrůstání do zbytku systému. Eliminací chyb již v průběhu vývoje se lze vyhnout zdlouhavému a únavnému hledání chyb v závěrečných fázích projektů.

Použité TDD taktéž minimalizuje potřebu krokování běhu aplikace. Jestliže některé testy skončí neočekávaně s chybou, lze ve spolupráci se systémem spravující verze zdrojových souborů (*Version control system*) zjistit změny kódu, které vedly k tomuto neočekávanému chování a opravit je. Tento způsob hledání chyb je ve většině případu efektivnější než krokování programu.

2.4 Omezení TDD

TDD ověřuje správnost návrhu a funkcionality softwaru pouze na základě napsaných testů. Nesprávný test, který nereprezentuje požadovanou specifikaci, bude produkovat nesprávný kód. Důraz na správnost a design softwaru se přesouvá na jeho testy, neboť testy jsou to, co řídí vývoj daného softwaru. Výsledkem tohoto posunu je, že software je pouze tak dobrý, jako jsou dobré jeho testy.

Díky rychlému střídání jednotlivých kroků vývojového cyklu TDD je každý programátor odpovědný za psaní vlastních testů. Testy nelze psát ve větších skupinách dopředu ani zpětně.

Jako techniku programování lze TDD jen omezeně využít pro návrh a vývoj nových algoritmů, ačkoli lze pomocí něho verifikovat jejich správnost.

TDD je také těžké použít v situacích, například ve spojení s grafickým uživatelským prostředím nebo relační databázi, neboť tyto systémy obsahují velké množství vstupů a výstupů a nebyly navrženy pro izolované testování.

3 Tutoriál

Následující tutoriál demonstruje vývoj jednoduché funkcionality s využitím nástroje FastUnit a techniky TDD. Tutoriál je rozdělen do tří částí. V první části je předvedeno vytvoření prvního jednoduchého testu. Test při svém prvním spuštění skončí s chybou, kvůli chybné implementaci testované třídy. Implementace testovací třídy bude následně opravena a test při svém znovuspuštění skončí úspěšně. V druhé části je přidán další test a na vzniklé dvojici testů bude prezentována úprava testovací třídy spočívající v odstranění duplicitních části testů. V poslední části je předvedeno testování chybových stavů testovací třídy. Zde jsou přidány dva testy, které budou testovat reakci testované třídy na chybné parametry.

3.1 Zadání

Základní možnosti nástroje FastUnit jsou prezentovány na modifikaci známého příkladu „Hello World". Výsledkem bude vytvoření jednoduché třídy *HelloWorld* a série testů, které plně pokryjí její funkcionalitu. Od třídy *HelloWorld* bude požadována následující funkcionalita.

- Metoda *string SayHello()*, která jako návratovou hodnotu vrátí text „Hello World!".
- Metoda *string SayHelloTo(string name)*, která jako návratovou hodnotu vrátí text „Hello [name]!".

3.1.1 Předpoklady

- Nainstalované vývojové prostředí Microsoft Visual Studio 2003, .NET verze 2.0, a FastUnit
- Základní znalost práce s vývojovým prostředím Microsoft Visual Studio 2003
- Základní znalostí programovacího jazyka C#

Tutoriál předpokládá základní znalosti práce s vývojovým prostředím Microsoft Visual Studio 2003, tudíž nepopisuje kroky nutné například k vytvoření projektu, přidání referencí mezi projekty, přidání nové třídy do projektu či dalších úkonů nutných k dokončení tutoriálu. Třída *HelloWorld* bude součástí modulu *fastUnit.Samples*. Testy pro třídu *HelloWorld* budou součástí testovacího modulu *fastUnit.Samples.Tests*. V praxi se velice často používá jmenná konvence, podle které se testovací moduly pojmenovávají stejně jako testované moduly s příponou „*Tests*". Testovací modul musí obsahovat referenci na knihovnu definující rozhraní pro nástroj FastUnit - jedná se o knihovnu *nunit.framework.dll* umístěnou v instalačním adresáři nástroje FastUnit.

3.2 První test

V této části tutoriálu bude vytvořen první test pokrývající funkcionalitu metody *SayHello* třídy *HelloWorld* a bude předvedeno jeho spuštění.

3.2.1 Nejdříve test

V prvním kroku bude vytvořená testovací třída. To je demonstrováno následujícím kódem.

```
using System;
using NUnit.Framework;

namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class HelloWorldTests
    {
    }
}
```

Obrázek 3.1: Prázdná testovací třída pro třídu *HelloWorld*.

Testovací třída je obyčejná třída, která je označena atributem *TestFixture*. Testovací třída musí být veřejná, nesmí být abstraktní a musí mít bezparametrický konstruktor. Atribut *TestFixture* nemá žádné parametry a může být pouze použit na označení tříd. Cílem atributu *TestFixture* je označit třídy obsahují testy.

V dalším kroku bude vytvořen první skutečný test. To je demonstrováno následujícím kódem.

```
using System;
using NUnit.Framework;

namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class HelloWorldTests
    {
        [Test]
        public void SayHelloTest()
        {   // Step 1: Set up some objects
            HelloWorld helloWorld = new HelloWorld();

            // Step 2: Manipulate the objects
            string message = helloWorld.SayHello();
            // Step 3: Assert outcome is correct
            Assert.IsNotNull(message);
            Assert.AreEqual("Hello World!", message);

            // Step 4: Clean up objects
            helloWorld = null;
        }
    }
}
```

Obrázek 3.2: Test metody *SayHello* třídy *HelloWorld*.

Test je obyčejná metoda v rámci testovací třídy, která je označena atributem *Test*. Testovací metoda musí být veřejná, nesmí být abstraktní, nesmí vyžadovat žádné

parametry a její návratová hodnota musí být typu *void*. Atribut *Test* má jeden volitelný atribut, pomocí kterého lze k danému testu přiřadit slovní popis testu.

V ukázce lze vidět tělo testu rozdělené do čtyř kroků. Toto rozdělení ani vepsané komentáře nejsou povinné, jedná se pouze o demonstraci doporučeného uspořádaní testů.

FastUnit jako řada podobných nástrojů pro testování modulů nabízí bohatou množinu kontrolních funkcí dostupných ve třídě *Assert*. V příkladě jsou použity dvě. První *Assert.IsNotNull(object value)* ověřuje, že objekt není prázdný. Druhá *Assert.AreEqual(string expected, string actual)* ověřuje, že dvě textové hodnoty se navzájem rovnají. V tom případě to znamená, že textová hodnota navrácená metodou *SayHello* je rovna očekávané textové hodnotě „Hello World!“.

3.2.2 Počáteční implementace

První test pro třídu *HelloWorld* je již sice připraven, ale nelze jej přeložit, neboť testovaná třída ještě neexistuje. V tomto kroku bude vytvořena počáteční implementace třídy *HelloWorld*. To je demonstrováno následujícím kódem (obr 3.3).

```
using System;

namespace fastUnit.Samples
{
    public class HelloWorld
    {
        public string SayHello()
        {   return null;
        }
    }
}
```

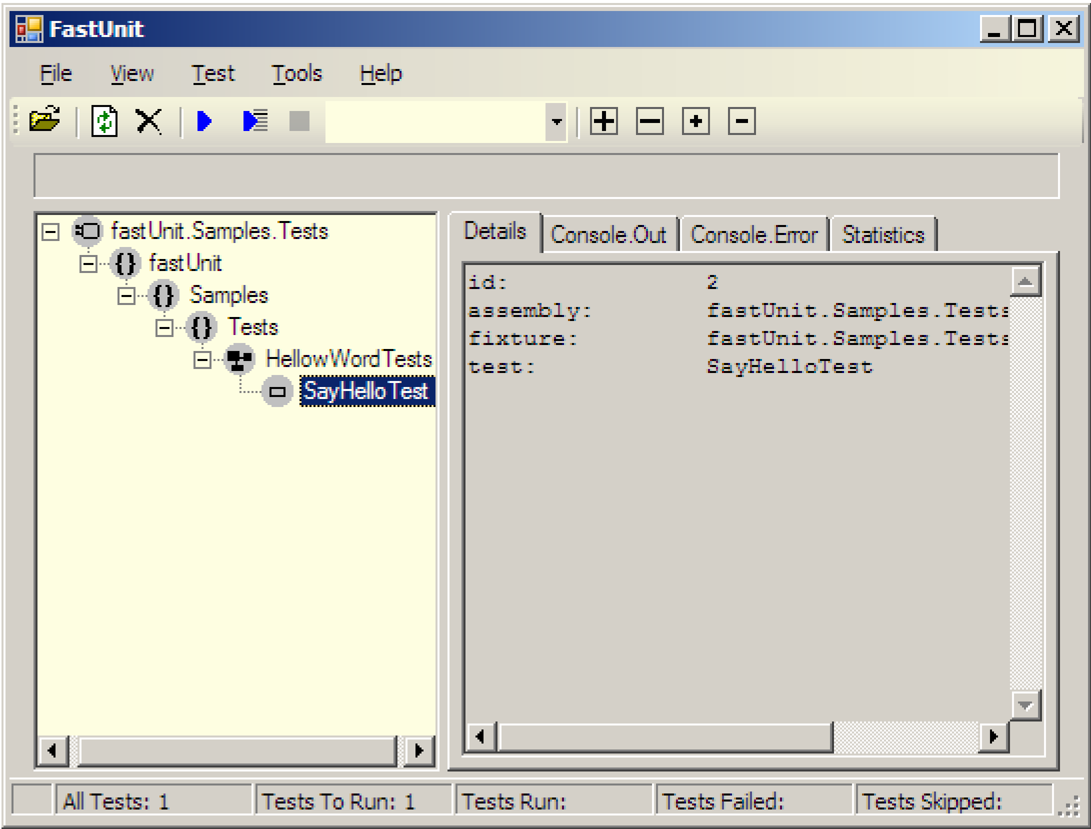
Obrázek 3.3: Počáteční implementace třídy *HelloWorld* obsahující pouze metodu *SayHello*.

Na první pohled lze vidět, že první implementace třídy *HelloWorld* je chybná a neúplná. 1) Metoda *SayHello* nevrací požadovaný text. 2) Třída neobsahuje druhou požadovanou metodu *SayHelloTo*.

V současné fázi je test, který byl vytvořen v předchozím kroku, jedinou skutečnou specifikací požadovanou po třídě *HelloWorld*. Cílem této fáze je pouze umožnit první spuštění daného testu. Další funkcionalita bude do třídy *HelloWorld* přidána až poté, co budou implementovány další testy. Na test lze pohlížet jako na jednu z forem specifikace. V každém kroku lze implementovat pouze funkcionalitu požadovanou testy a pracovat pouze na jednom testu v daném čase. Tento přístup může znít extrémně, ale pokud je důsledně dodržován, je velice efektivní.

3.2.3 Spuštění testu a oprava implementace

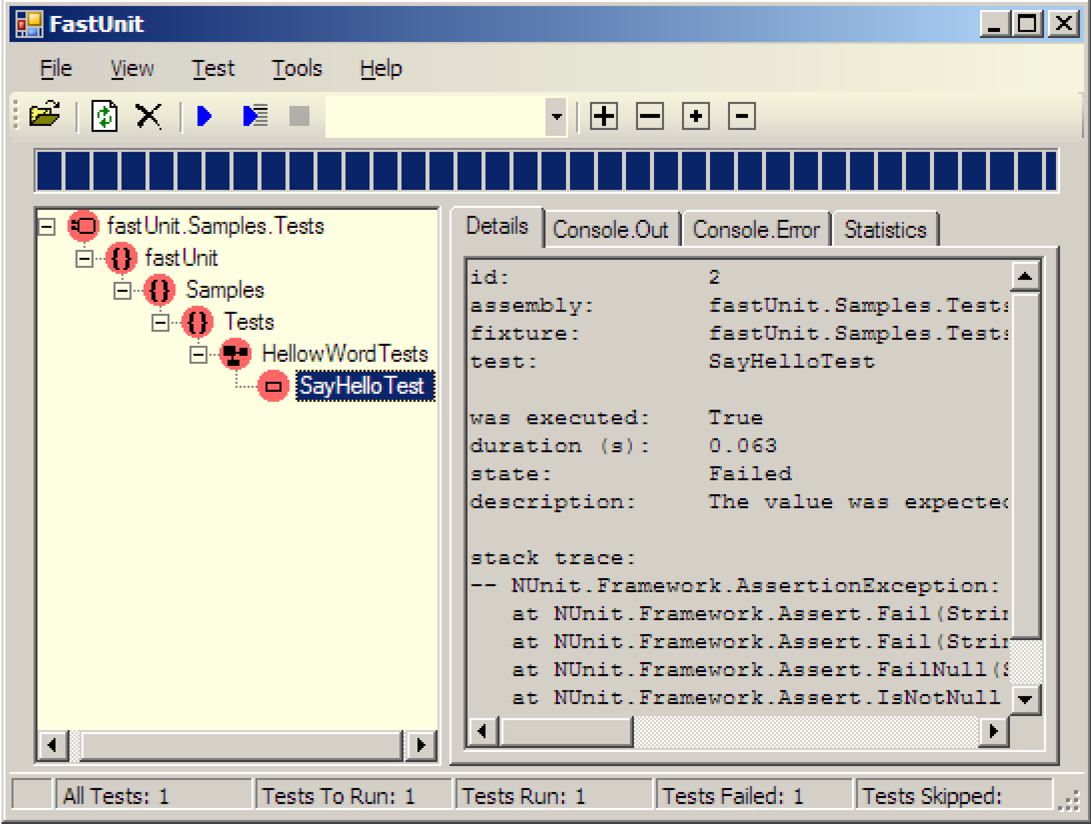
Nyní je již vše připraveno k prvnímu spuštění testu. Spuštění bude demonstrováno v aplikaci *fastUnit.GUI.exe*. Jedná se o jednu z aplikací nástroje FastUnit, která nabízí uživateli práci s testy v uživatelsky příjemném grafickém prostředí. Po spuštění aplikace se zobrazí hlavní okno, které je rozděleno vertikálně do dvou hlavních částí. Levá část slouží pro zobrazení dostupných testů. Pravá část slouží pro zobrazení detailů dostupných testů a pro prezentaci výsledků proběhlých testů.



Obázek 3.4: Aplikace FastUnit zobrazující dostupný test *SayHelloTest*.

Otevření testovacího modulu se provede příkazem *File/Open*. Po otevření testovacího modulu je obsah modulu prohledán a nalezené testy jsou zobrazeny v levé části aplikace (obr. 3.4). Dostupné testy jsou zatříděny ve stromě, který hierarchicky reprezentuje obsah testovaného modulu. Kořenový uzel představuje vlastní testovací modul. Pod kořenovým uzlem jsou zobrazeny uzly reprezentující jmenné prostory použité v testovacím modulu. Dále jsou podle svého umístění zobrazeny dostupné testovací třídy. Listy stromu tvoří jednotlivé testy.

Nyní bude přistoupeno k prvnímu spuštění testu. Test se spouští příkazem *Test/Run All*. Po doběhnutí spuštěného testu se zobrazí následující výsledek (obr. 3.5).



Obrázek 3.5: Aplikace FastUnit zobrazující první chybný výsledek testu *SayHelloTest*.

Výsledek prvního spuštění není překvapivý. Test skončil chybou. Záložka *Details* obsahuje podrobný výsledek proběhlého testu včetně popisu vyskytlé chyby.

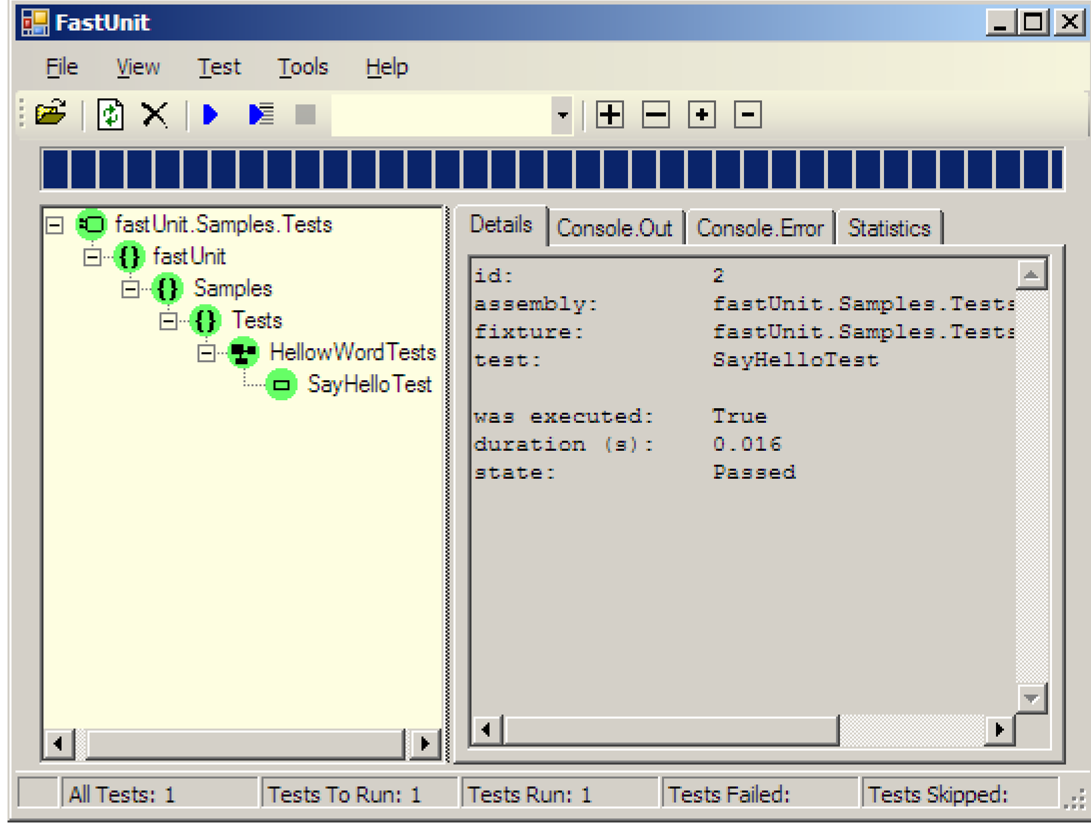
V následujícím kroku bude opravena implementace třídy *HelloWorld* tak, aby splňovala podmínky existujícího testu. To je demonstrováno následujícím kódem (obr. 3.6).

```
using System;

namespace fastUnit.Samples
{
    public class HelloWorld
    {
        public string SayHello()
        {   return "Hello World!";
        }
    }
}
```

Obrázek 3.6: Implementace třídy *HelloWorld* obsahující pouze metodu *SayHello*.

Jediná úprava, která byla provedena, je změna navrácené hodnoty z metody *SayHello* na požadovanou hodnotu. Po rekompilaci a opětovném spuštění testu se zobrazí následující výsledek (obr. 3.7).



Obrázek 3.7: Aplikace FastUnit zobrazující první úspěch testu *SayHelloTest*.

Test skončil úspěšně. V tomto bodě byla vytvořena první verze třídy *HelloWorld*, která plně implementuje funkcionalitu požadovanou existujícím testem.

3.3 Druhý test a úprava testovací třídy

V této části tutoriálu bude přidán druhý test pokrývající druhou požadovanou metodu a implementace požadované metody. Na vzniklé dvojici testů bude následně prezentována úprava testovací třídy spočívající v odstranění duplicitních částí testů.

3.3.1 Druhý test

Druhou požadovanou metodou třídy *HelloWorld* je metoda *SayHelloTo*. V následujícím kroku bude přidán test na metodu *SayHelloTo* do testovací třídy. To je demonstrováno následujícím kódem (obr. 3.8).

```
using System;
using Nunit.Framework;

namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class HelloWorldTests
    {
        [Test]
        public void SayHelloTest()
        {   // Step 1: Set up some objects
            HelloWorld helloWorld = new HelloWorld();

            // Step 2: Manipulate the objects
            string message = helloWorld.SayHello();
            // Step 3: Assert outcome is correct
```

```
            Assert.IsNotNull(message);
            Assert.AreEqual("Hello World!", message);

            // Step 4: Clean up objects
            helloWorld = null;
        }

        [Test]
        public void SayHelloToTest()
        {   // Step 1: Set up some objects
            HelloWorld helloWorld = new HelloWorld();

            // Step 2: Manipulate the objects
            string message = helloWorld.SayHelloTo("Ema");
            // Step 3: Assert outcome is correct
            Assert.IsNotNull(message);
            Assert.AreEqual("Hello Ema!", message);

            // Step 4: Clean up objects
            helloWorld = null;
        }
    }
}
```

Obrázek 3.8: Testovací třída *HelloWorld* po doplnění testu metody *SayHelloTo*.

Druhý test je připraven. Nyní bude přidána vlastní neprázdná implementace metody *SayHelloTo*. To je demonstrováno následujícím kódem (obr. 3.9).

```
using System;

namespace fastUnit.Samples
{
    public class HelloWorld
    {
        public string SayHello()
        {   return "Hello World!";
        }

        public string SayHelloTo(string name)
        {   return string.Format("Hello {0}!", name);
        }
    }
}
```

Obrázek 3.9: Implementace třídy *HelloWorld* po doplnění metody *SayHelloTo*.

Pro urychlení prezentace byl vynechán krok s prázdnou či nefunkční implementací testované metody. Druhý test i testovaná metoda jsou připraveny. Po přeložení a spuštění oba existující testy skončí úspěšně.

Dosavadní část tutoriálu ukázala, že vytvoření testovací třídy i jednotlivých testů je jednoduchou záležitostí. Avšak rostoucí složitostí testovaných tříd poroste i složitost a pracnost testů pokrývající dané třídy.

Pro ulehčení psaní rozsáhlejších testů umožňuje nástroj FastUnit definovat metody testovací třídy, které budou spouštěny bezprostředně před a bezprostředně po spuštění jednotlivých testů. Tato funkcionalita bude prezentována v následujícím odstavci.

3.3.2 Úprava testovací třídy

Na obrázku 3.8 lze vidět podobnost obou dosavadně implementovaných testů. Kroky 1 a 4 jsou v obou případech shodné. Jedná se o kroky, ve kterých se provádí inicializace a úklid objektů potřebných pro provedení vlastního testu. V následujícím kroku bude předvedeno přesunutí těchto duplicitních částí do společných metod, které jsou automaticky spouštěny před a po spuštění jednotlivých testů. To je demonstrováno následujícím kódem (obr. 3.10).

```
using System;
using Nunit.Framework;

namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class HelloWorldTests
    {
        private HelloWorld _helloWorld = null;

        [SetUp]
        public void TestSetUp()
        {   // Step 1: Set up some objects
            _helloWorld = new HelloWorld();
        }

        [TearDown]
        public void TestTearDown()
        {   // Step 4: Clean up objects
            _helloWorld = null;
        }

        [Test]
        public void SayHelloTest()
        {   // Step 2: Manipulate the objects
            string message = _helloWorld.SayHello();
            // Step 3: Assert outcome is correct
            Assert.IsNotNull(message);
            Assert.AreEqual("Hello World!", message);
        }

        [Test]
        public void SayHelloToTest()
        {   // Step 2: Manipulate the objects
            string message = _helloWorld.SayHelloTo("Ema");
            // Step 3: Assert outcome is correct
            Assert.IsNotNull(message);
            Assert.AreEqual("Hello Ema!", message);
        }
    }
}
```

Obrázek 3.10: Implementace testovací třídy *HelloWorld* s použitím inicializační a úklidové metody.

Úprava testovací třídy zahrnovala: 1) Přesunutí lokální proměnné *helloWorld* z jednotlivých testů na úroveň třídy tak, aby i nadále byla dostupná ze všech testů. 2) Přesunutí inicializace testované třídy do společné metody *TestSetUp*. 3) Přesunutí úklidu testované třídy do společné metody *TestTearDown*.

Atributy *SetUp* a *TearDown* slouží pro označení metod, které provádějí inicializaci a úklid vnitřního stavu testovací třídy. Vnitřním stavem testovací třídy může být

cokoliv co je nutné pro provedení jednotlivých testů - například vytvořené testované objekty, předpřipravené testovací data, vytvoření spojení na databázový server, apod. Oba druhy metod, jak inicializační tak úklidové, musí být metody veřejné, nesmí být abstraktní, nesmí vyžadovat žádné parametry a jejích návratová hodnota musí být typu *void*.

3.4 Chybové stavy

V poslední části tutoriálu bude popsána implementace testů, u kterých se během jejich spuštění očekává, že testovaný kód či test samotný skončí chybou. Tato technika bude předvedena na metodě *SayHelloTo* testované třídy *HelloWorld*. Metoda na svém vstupu očekává jeden parametr typu *string* obsahující jméno, které má být vloženo do výsledného textu. Jeden z dalších požadavků, který může být kladen na metodu *SayHelloTo* je kontrola vstupního parametru. Vstupní parametr je typu *string*. Nabízí se tedy možnost kontrolovat parametr na nulovou hodnotu a na hodnotu obsahující text nulové délky.

Po přidání požadavku na kontrolu vstupních parametrů je po třídě *HelloWorld* požadována následující funkcionalita.

- Metoda *string SayHello()*, která jako návratovou hodnotu vrátí text „Hello World!“.
- Metoda *string SayHelloTo(string name)*, která jako návratovou hodnotu vrátí text „Hello [name]!“
    - V případě, že parametr *name* obsahuje nulovou hodnotu, metoda skončí s výjimkou typu *ArgumentNullException*.
    - V případě, že parametr *name* obsahuje text nulové délky, metoda skončí s výjimkou typu *ArgumentException*.

Testování kódu u kterého se očekává jeho ukončení s výjimkou, lze provést následujícími způsoby.

1. Prvním způsobem je uzavřít požadovaný kód do *try/catch* bloku a ručně testovat výskyt požadované chyby. Tento způsob není příliš přehledný, neboť vede k zanesení testovací třídy množstvím kódu, který bude znesnadňovat čtení a interpretaci vlastních testů.
2. Druhou a preferovanou možností je použití atributu *ExpectedException*. Atribut *ExpectedException* slouží pro označení testovací metody, u které se během jejího spuštění očekává výskyt výjimky. V případě, kdy specifikovaná výjimka nastane, je daná výjimka ignorována a test skončí úspěšně. V případě, kdy během spuštění testu nedojde k žádné výjimce nebo nastane jiná než očekávaná výjimka, test skončí s chybou.

Do testovací třídy bude tedy přidán jeden test pro každou novou požadovanou funkcionalitu. To je demonstrováno následujícím kódem (obr. 3.11).

```
using System;
using NUnit.Framework;

namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class HelloWorldTests
    {
        private HelloWorld _helloWorld = null;

        [SetUp]
        public void TestSetUp()
        {   // Step 1: Set up some objects
            _helloWorld = new HelloWorld();
        }

        [TearDown]
        public void TestTearDown()
        {   // Step 4: Clean up objects
            _helloWorld = null;
        }

        [Test]
        public void SayHelloTest()
        {   // Step 2: Manipulate the objects
            string message = _helloWorld.SayHello();
            // Step 3: Assert outcome is correct
            Assert.IsNotNull(message);
            Assert.AreEqual("Hello World!", message);
        }

        [Test]
        public void SayHelloToTest()
        {   // Step 2: Manipulate the objects
            string message = _helloWorld.SayHelloTo("Ema");
            // Step 3: Assert outcome is correct
            Assert.IsNotNull(message);
            Assert.AreEqual("Hello Ema!", message);
        }

        [Test]
        [ExpectedException(typeof(ArgumentNullException))]
        public void SayHelloToNullNameTest()
        {   // Step 2: Manipulate the objects
            _helloWorld.SayHelloTo(null);
        }

        [Test]
        [ExpectedException(typeof(ArgumentException))]
        public void SayHelloToEmptyNameTest()
        {   // Step 2: Manipulate the objects
            _helloWorld.SayHelloTo("");
        }
    }
}
```

Obrázek 3.11: Testovací třída *HelloWorld* po doplnění testů očekávajících výjimku.

Atributem *ExpectedException* jsou označeny obě přidané testovací metody, neboť u obou se očekává, že jejich spuštění skončí výjimkou. Parametrem atributu *ExpectedException* je v obou případech typ očekávané výjimky.

Po překompilování a spuštění testů, dva naposledy přidané testy skončí s chybou, neboť testovaný kód neskončil s požadovanou výjimkou. V posledním kroku tohoto tutoriálu zbývá upravit implementaci třídy *HelloWorld* tak, aby splňovala požadavky kladené novými testy. To je demonstrováno následujícím kódem (obr. 3.12).

```
using System;

namespace fastUnit.Samples
{
    public class HellowWorld
    {
        public string SayHello()
        {
           return "Hello World!";
        }

        public string SayHelloTo(string name)
        {
           if(name == null) throw new ArgumentNullException("name");
           if(name.Length == 0) throw new ArgumentException("name");

           return string.Format("Hello {0}!", name);
        }
    }
}
```

Obrázek 3.12: Finální implementace třídy *HelloWorld*.

Po překompilování a spuštění testů všechny testy skončí úspěšně.

4 Funkcionalita převzatá z nástroje NUnit

Základním požadavkem kladeným na nástroj FastUnit je zachování zpětné kompatibility s již existujícím testovacím nástrojem NUnit. Důvodem tohoto požadavku je nutnost, aby již existující testy vyvíjené původně pro nástroj NUnit bylo možné zcela automaticky (či pouze jen s vyvinutím minimálního úsilí) přenést a používat v nástroji FastUnit.

Tato kapitola prezentuje funkcionalitu implementovanou nástrojem FastUnit z důvodů zachování výše zmíněné kompatibility. První část popisuje třídu *Assert* nabízející množinu kontrolních funkcí, které lze použít při verifikaci testů. V dalších částech budou podrobně popsány jednotlivé atributy používané nástrojem FastUnit pro identifikaci a detailnější specifikaci testů.

4.1 Třída Assert

Nejjednodušším způsobem jak kontrolovat, zda test proběhl úspěšně, je napsat logický výraz, který bude reprezentovat výsledek daného testu. Takto napsaný výraz pak pomůže automatizovat proces rozhodování - zda daný test proběhl dle očekávání či zda skončil s chybou.

Snahou nástroje FastUnit je vytvořit plně automatické testování bez nutnosti zapojení člověka do procesu vyhodnocování výsledků. Cílem je stav, kdy stačí spustit testy a veškerou práci s prováděním a vyhodnocováním testů převezme počítač. To vede k následujícím požadavkům.

- Logický výraz reprezentující výsledek testu musí být typu *ano/ne*. Kde odpověď *ano* znamená, že všechno proběhlo úspěšně, a odpověď *ne* znamená, že se stalo něco neočekávaného.
- Odpověď na logický výraz musí být vyhodnocena počítačem zcela samostatně.

Podpora vyhodnocování logických výrazů je základem každého testovacího prostředí a FastUnit není výjimkou. FastUnit nabízí bohatou škálu kontrolních funkcí implementovaných třídou *Assert*. Úkolem těchto funkcí je kontrolovat předložené dotazy a v případě nalezení nesrovnalosti skončit s chybou. Chyba je avizována vytvořením výjimky typu *AssertException* obsahující podrobný popis nastalé situace.

Jestliže volání kontrolní funkce skončí chybou, běh testu je přerušen a chyba je reportována. Jestliže test obsahuje více volání kontrolních funkcí, tak kontrolní funkce nacházející se za kontrolní funkcí, která skončila chybou, již nebudou provedeny. Z toho důvodu je doporučeno testovat v rámci jednoho testu pouze jednu požadovanou funkcionalitu. Funkcionality, které spolu přímo nesouvisejí, by měly být testovány pomocí oddělených testů.

Každá kontrolní funkce je podporována ve třech verzích. Ve verzi bez uživatelem specifikované chybové zprávy, s jednoduchou chybovou zprávou a s chybovou zprávou a argumenty. V posledním případě je chybová zpráva naformátována pomocí předaných argumentů. Následuje ukázka metody *Assert.AreEqual* ve všech nabízených verzích. Z důvodu úspory místa budou zbylé metody prezentovány pouze v první verzi.

```
Assert.AreEqual(int expected, int actual);
Assert.AreEqual(int expected, int actual, string message);
Assert.AreEqual(int expected, int actual, string message,
                object[] parms);
```

Obrázek 4.1: Ukázka metody *Assert.AreEqual* ve všech nabízených verzích.

Kontrolní funkce jsou rozděleny do následujících skupin.

4.1.1 Kontrola jednoduchých podmínek

Metody testující jednoduché podmínky jsou pojmenovány podle podmínky, kterou testují. Jako první parametr požadují testovanou hodnotu. Následuje seznam dostupných metod.

```
Assert.IsTrue(bool condition);
Assert.IsFalse(bool condition);
Assert.IsNull(object anObject);
Assert.IsNotNull(object anObject);
```

Obrázek 4.2: Přehled kontrolních metod testujících jednoduché podmínky.

4.1.2 Kontrola rovnosti a nerovnosti

Tyto metody testují, zda se hodnoty dvou prvních parametru navzájem rovnají. Prvním parametrem je vždy očekávána hodnota. Druhý parametrem je vždy aktuální hodnota, která vznikne výpočtem testu. Metody jsou nabízeny pro všechny běžné typy hodnot, aby se zamezilo zbytečnému boxování hodnot základních typů.

```
Assert.AreEqual(bool expected, bool actual);
Assert.AreEqual(byte expected, byte actual);
Assert.AreEqual(char expected, char actual);
Assert.AreEqual(int expected, int actual);
Assert.AreEqual(uint expected, uint actual);
Assert.AreEqual(long expected, long actual);
Assert.AreEqual(ulong expected, ulong actual);
Assert.AreEqual(decimal expected, decimal actual);
Assert.AreEqual(float expected, float actual);
Assert.AreEqual(double expected, double actual);
Assert.AreEqual(object expected, object actual);

Assert.AreNotEqual(bool expected, bool actual);
Assert.AreNotEqual(byte expected, byte actual);
Assert.AreNotEqual(char expected, char actual);
Assert.AreNotEqual(int expected, int actual);
Assert.AreNotEqual(uint expected, uint actual);
Assert.AreNotEqual(long expected, long actual);
Assert.AreNotEqual(ulong expected, ulong actual);
Assert.AreNotEqual(decimal expected, decimal actual);
Assert.AreNotEqual(float expected, float actual);
Assert.AreNotEqual(double expected, double actual);
Assert.AreNotEqual(object expected, object actual);
```

Obrázek 4.3: Přehled kontrolních metod testujících rovnost a nerovnost.

Pro porovnávání čísel s desetinou čárkou jsou nabízeny i metody umožňující specifikovat toleranci, ve které budou porovnávané hodnoty považovány za rovné.

```
Assert.AreEqual(double expected, double actual, double tolerance);
Assert.AreEqual(float expected, float actual, float tolerance);
```

Obrázek 4.4: Přehled kontrolních metod porovnávající čísla s desetinou čárkou s požadovanou tolerancí.

FastUnit nabízí i porovnávání polí na rovnost. Jsou podporovány jednorozměrná, vícerozměrná i vnořená pole. Dvě pole budou shledány jako sobě rovné pomocí metody *Assert.AreEqual*, pokud mají stejnou dimenzi, stejný počet prvků a jestliže všechny korespondující prvky jsou si rovny.

4.1.3 Kontrola identity

Metody pro kontrolu identity testují, zda první dva parametry odkazují na stejný objekt.

```
Assert.AreSame(object expected, object actual);
Assert.AreNotSame(object expected, object actual);
```

Obrázek 4.5: Přehled kontrolních metod testujících identitu.

4.2 Atributy

Ve světě testovacích nástrojů se používá několik základních způsobů, jak identifikovat a detailněji specifikovat uživatelem napsané testy.

- **Jmenná konvence –** Jména tříd obsahujících testy i jména testů samotných musí splňovat vzor požadovaný nástrojem.
- **Dědičnost a polymorfismus –** Třídy obsahující testy musí být odvozené od třídy definované nástrojem. Pro identifikací jednotlivých testů se používá jmenná konvence nebo registrační metoda implementována testovací třídou.
- **Atributy –** Třídy a jednotlivé testy jsou označeny speciálními atributy.

FastUnit používá speciální atributy pro identifikaci testovacích tříd i jednotlivých testů.

4.2.1 Atribut TestFixture

Atribut *TestFixture* slouží pro označení testovacích tříd. Testovací třída je obyčejná třída, která obsahuje jednotlivé testy v podobě svých veřejných metod. Testovací třída může volitelně obsahovat inicializační a úklidové metody.

Na testovací třídy jsou kladeny následující požadavky.

- Testovací třída musí být veřejná. FastUnit hledá testovací třídy pouze mezi veřejně viditelnými třídami. V jazyce C# to znamená, že testovací třída musí být deklarovaná jako *public*. FastUnit nehledá testovací třídy mezi třídami deklarovanými jako privátní či interní.
- Testovací třída musí byt instaciovatelná neboli nesmí být abstraktní. V jazyce C# to znamená, že nesmí být deklarovaná jako *abstract*.
- Testovací třída musí mít bezparametrický konstruktor.
- Testovací třída může obsahovat maximálně po jedné metodě označené následujícími atributy: *SetUp*, *TearDown*, *FixtureSetUp* a *FixtureTearDown*.

V případě, kdy testovací třída nesplní některý z výše uvedených požadavků, bude celá testovací třída označená jako nevalidní a jakýkoliv pokus o její spuštění bude odmítnut.

```
using System;
using NUnit.Framework;
namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class TestFixture01
    {     // ...
    }
}
```

Obrázek 4.6: Ukázka použití atributu *TestFixture*.

4.2.2 Atribut Test

Atribut *Test* slouží pro označení jednotlivých testů v rámci testovací třídy. Atributem *Test* může být označena každá metoda testovací třídy splňující následující požadavky.

- Metoda je veřejná.
- Metoda nevyžaduje žádné parametry.
- Návratová hodnota metody je typu *void*.

V případě, že testovací metoda nesplní některý z výše uvedených požadavků, bude označena jako nevalidní a jakýkoliv pokus o její spuštění bude odmítnut.

FastUnit podporuje také parametrické testy, které mohou vyžadovat libovolný počet parametrů libovolných typů. Parametrické testy budou popsány v odstavci 5.2.3 zabývající se atributem *Data*.

```
using System;
using NUnit.Framework;
namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class TestFixture02
    {
        [Test(Description="My first test.")]
        public void Test01()
        {     // ...
        }
    }
}
```

Obrázek 4.7: Ukázka použití atributu *Test*.

4.2.3 Životní cyklus testovací třídy

Při návrhu a implementaci testů lze velice často v rámci struktury jednotlivých testů vypozorovat následující vzor.

1. **Inicializace –** Vytvoření testovacích nebo testovaných objektů, jejich vzájemné propojení, příprava testovacích dat apod.
2. **Test –** Provedení vlastního testu.
3. **Kontrola –** Kontrola výsledků proběhlého testu.
4. **Úklid –** Uvolnění vytvořených objektů, vyčištění testovacích dat apod.

Pokud se výše zmíněný vzor opakuje, FastUnit umožňuje přesunout opakující se části do společných inicializačních a úklidových metod definovaných na úrovni testovací třídy. Při rozhodování o tom, které části mohou být přesunuty, lze narazit na dva protichůdné požadavky.

1. **Výkon testů.** Testy by měly běžet co možná nejrychleji. Z pohledu rychlosti by bylo dobré, aby se například sdílené objekty vytvářely a uvolňovaly pouze jednou pro všechny testy.
2. **Izolace testů.** Testy by se neměly navzájem ovlivňovat. Úspěch či neúspěch jednoho testu by neměl mít za žádných okolností vliv na úspěch či neúspěch jiného testu. Pokud testy sdílejí společný objekt a jeden test změní obsah sdíleného objektu, je velice pravděpodobné, že daná změna ovlivní výsledky dalších testů.

Požadavek na izolaci jednotlivých testů má větší prioritu než požadavek na rychlost testování. To má za následek vznik následujícího pravidla:

Pokud jednotlivé testy mění stav testovaných objektů nebo testovacích dat, je nutné dané testované objekty nebo testovací data inicializovat před spuštěním každého jednotlivého testu.

FastUnit nabízí možnost na úrovni třídy definovat dvě dvojice inicializačních a úklidových metod.

- První dvojicí jsou metody, které jsou spuštěny bezprostředně před a bezprostředně po spuštění každého jednotlivého testu. Pro označení těchto metod slouží atributy *SetUp* a *TearDown*.
- Druhou dvojicí jsou metody, které jsou spuštěny jednou pro všechny testy testovací třídy. Inicializační metoda je spuštěna před spuštěním prvního testu testovací třídy. Úklidová metoda je spuštěna po skončení posledního testu testovací třídy. Pro označení těchto metod slouží atributy *TestFixtureSetUp* a *TestFixtureTearDown*.

```
using System;
using NUnit.Framework;
namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class TestFixture03
    {
        [TestFixtureSetUp]
        public void FixtureSetUp()
        {     // ...
        }

        [TestFixtureTearDown]
        public void FixtureTearDown()
        {     // ...
        }

        [SetUp]
        public void TestSetUp()
        {     // ...
        }

        [TearDown]
        public void TestTearDown()
```

```
        {   // ...
        }

        [Test]
        public void Test01()
        {   // ...
        }

        [Test]
        public void Test02()
        {   // ...
        }
    }
}
```

Obrazek 4.8: Ukázka použití atributů SetUp, TearDown, TestFixtureSetUp a TestFixtureTearDown.

V průběhu testování třídy *TestFixture03* (obr. 4.8) budou metody třídy spuštěny přesně v následujícím pořadí.

1. FixtureSetUp
2. TestSetUp
3. Test01
4. TestTearDown
5. TestSetUp
6. Test02
7. TestTearDown
8. FixtureTearDown

4.2.4 Atribut SetUp

Atribut *SetUp* slouží v rámci testovací třídy pro označení metody, která je spuštěna bezprostředně před spuštěním každého jednotlivého testu dané testovací třídy. Testovací třída může obsahovat maximálně jednu metodu označenou atributem *SetUp*. Atributem *SetUp* může být označená metoda testovací třídy splňující následující požadavky.

- Metoda je veřejná.
- Metoda nevyžaduje žádné parametry.
- Návratová hodnota metody je typu *void*.

Jestliže při spuštění metody označené atributem *SetUp* dojde k chybě, vlastní test není spuštěn a je označen jako chybný.

4.2.5 Atribut TearDown

Atribut *TearDown* slouží v rámci testovací třídy pro označení metody, která je spuštěna bezprostředně po skončení každého jednotlivého testu dané testovací třídy. Testovací třída může obsahovat maximálně jednu metodu označenou atributem *TearDown*. Atributem *TearDown* může být označená metoda testovací třídy splňující následující požadavky.

- Metoda je veřejná.
- Metoda nevyžaduje žádné parametry.
- Návratová hodnota metody je typu *void*.

Jestliže při spuštění metody označené atributem *TearDown* dojde k chybě, právě proběhlý test bude označen jako chybný. Metoda označená atributem *TearDown* je spuštěna i v případě, kdy během testu dojde k chybě. Případná další chyba vzniklá během běhu úklidové metody je již ignorována.

4.2.6 Atribut TestFixtureSetUp

Atribut *TestFixtureSetUp* slouží v rámci testovací třídy pro označení metody, která je spuštěna pouze jednou a to před spuštěním prvního testu dané testovací třídy. Takto označenou metodu je vhodné využít k inicializaci časově náročných zdrojů, které nejsou během vlastních testů modifikovány. Testovací třída může obsahovat maximálně jednu metodu označenou atributem *TestFixtureSetUp*. Atributem *TestFixtureSetUp* může být označená metoda testovací třídy splňující následující požadavky.

- Metoda je veřejná.
- Metoda nevyžaduje žádné parametry.
- Návratová hodnota metody je typu *void*.

Jestliže při spuštění metody označené atributem *TestFixtureSetUp* dojde k chybě, žádný test testovací třídy není spuštěn a testovací třída je jako celek označena za chybnou.

4.2.7 Atribut TestFixtureTearDown

Atribut *TestFixtureTearDown* slouží v rámci testovací třídy pro označení metody, která bude spuštěna pouze jednou bezprostředně po skončení posledního testu dané testovací třídy. Takto označenou metodu je vhodné využít k úklidu zdrojů, které byly inicializovány během metody označené atributem *TestFixtureSetUp*. Testovací třída může obsahovat maximálně jednu metodu označenou atributem *TestFixtureTearDown*. Atributem *TestFixtureTearDown* může být označená metoda testovací třídy splňující následující požadavky.

- Metoda je veřejná.
- Metoda nevyžaduje žádné parametry.
- Návratová hodnota metody je typu *void*.

Jestliže při spuštění metody označené atributem *TestFixtureTearDown* dojde k chybě, testovaná třída je jako celek označena za chybnou.

4.2.8 Atribut ExpectedException

Atribut *ExpectedException* slouží pro označení testů, u kterých se očekává, že jejich běh bude ukončen vytvořením výjimky. Atribut *ExpectedException* má řadu pevných i pojmenovaných parametrů, které umožňují detailnější popis očekávané výjimky.

Specifikace typu očekávané výjimky

Základním a nejvíce používaným způsobem upřesnění očekávané výjimky je specifikace jejího typu. FastUnit umožňuje zadat typ očekávané výjimky dvěma způsoby:

1. První způsob je předáním objektu reprezentující typ dané výjimky.
2. Druhý způsob je specifikování plného jména typu výjimky.

Výhodou prvního způsobu je možnost kontroly existence požadovaného typu výjimky již během kompilace testu. Naproti tomu výhodou druhého způsobu je, že nevyžaduje, aby testovací modul odkazoval přímo na modul definující očekávanou výjimku.

```
using System;
using NUnit.Framework;
namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class TestFixture04
    {
        [Test]
        [ExpectedException(typeof(InvalidOperationException))]
        public void Test01()
        {   // ...
        }

        [Test]
        [ExpectedException("System.InvalidOperationException")]
        public void Test02()
        {   // ...
        }
    }
}
```

Obrázek 4.9: Ukázka použití atributu *ExpectedException* specifikující pouze typ očekávané výjimky.

Na obrázku 4.9 oba testy očekávají vytvoření výjimky typu *InvalidOperationException*. Jestliže výjimka typu *InvalidOperationException* bude vytvořena - test skončí úspěšně. Jestliže bude vytvořena výjimka jiného typu nebo nebude vytvořena výjimka žádná - test skončí s chybou.

Při kontrole, zda očekávané a vytvořené typy výjimek si navzájem odpovídají, vyžaduje nástroj FastUnit úplnou rovnost obou typů. V opačném případě test skončí chybou. Tento způsob kontroly je odlišný od kontroly prováděné nástrojem NUnit. Nástroj NUnit umožňuje, aby vyhozená výjimka byla, buď přímo očekávaného typu nebo byla z očekávaného typu odvozena. Důvod této změny v chování nástroje FastUnit oproti nástroji NUnit vychází zcela z praxe a klade si za cíl zamezit následující situaci. Ve snaze ulehčit si práci se zjišťováním skutečných typů výjimek se píší testy, ve kterých se místo skutečných výjimek očekávají výjimky generické, v nejhorším případě pak výjimky typu *Exception*. Tento způsob psaní testů může a často vede k situaci, kdy se testy tváří jako úspěšné, ale ve skutečnosti končí s jinými výjimkami než bylo původně autory testů zamýšleno.

Specifikace zprávy očekávané výjimky

Dalším způsobem upřesnění očekávané výjimky je specifikace její chybové zprávy. Atribut *ExpectedException* umožňuje zvolit metodu kontroly textu očekávané zprávy pomocí volitelného parametru *MatchType*. Parametr *MatchType* je výčtového typu a může nabývat následujících hodnot.

- *MessageMatch.Exact* V tomto případě se kontroluje text chybové zprávy na úplnou shodu s očekávaným textem.
- MessageMatch.Contains V tomto případě se kontroluje, zda text chybové zprávy obsahuje očekávaný text.

- *MessageMatch.Regex* V tomto případě se kontroluje, zda text chybové zprávy odpovídá regulárnímu výrazu zadaného očekávaným textem.

V případě, kdy parametr *MatchType* není specifikován kontroluje se text chybové zprávy na úplnou shodu s očekávaným textem.

```
using System;
using NUnit.Framework;
namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class TestFixture05
    {
        [Test]
        [ExpectedException(typeof(InvalidOperationException),
                     "Expected message.")]
        public void Test01()
        {     // ...
        }

        [Test]
        [ExpectedException(typeof(InvalidOperationException),
                     "Fragment of expected message.",
                     MatchType=MessageMatch.Contains)]
        public void Test02()
        {     // ...
        }

        [Test]
        [ExpectedException(typeof(InvalidOperationException),
                     "Regular expression for expected message.",
                     MatchType=MessageMatch.Regex)]
        public void Test03()
        {     // ...
        }
    }
}
```

Obrázek 4.10: Ukázka použití atributu *ExpectedException* specifikující text očekávané výjimky.

4.2.9 Atribut Explicit

Jedna ze základních vlastností, která je požadována po testech, je izolovanost a nezávislost na okolním prostředí. Na druhou stranu nástroje typu FastUnit umožňují vývoj a testování i těch částí systémů, které určitým způsobem komunikují se svým okolím - například připojují se na databázový server, přistupují na lokální disk apod. Začlenění testů, během kterých testované části komunikují s okolním prostředím mezi ostatní testy, by mělo za následek vznik řady omezení, které by bránily spouštění testů nezávisle na jejich umístění. Např. požadovaný databázový server nemusí být vždy a ze všech testovacích míst přístupný, požadovaný soubor nemusí existovat apod. Jednou z možností jak řešit daný problém je vyvíjet oba druhy testů odděleně ve dvou nezávislých testovacích modulech. Tato možnost řeší problém pouze částečně. Na jedné straně umožňuje spouštění testů z jednoho testovacího modulu nezávisle na okolním prostředí a umístění. Na straně druhé však poměrně komplikuje proces vývoje testovaných částí. Je nutné udržovat dva různé testovací moduly a během vývoje pamatovat na to, že je vždy potřeba spouštět dvě sady testů umístěných ve dvou různých testovacích modulech. Řešení, které nabízí FastUnit je použití atributu *Explicit*.

Atribut *Explicit* slouží pro označení testů, které nemají být za všech okolností spouštěny automaticky, ale pouze na explicitní vyžádání uživatele. Atributem *Explicit* by měly být označeny následující testy.

- Testy, během kterých testovaný kód komunikuje s okolním prostředím.
- Dlouhotrvající testy, které není vhodné či dokonce možné z časových důvodu spouštět pravidelně.

Atributem *Explicit* lze označit jak celé testovací třídy tak i jednotlivé testy. Atribut má jeden volitelný parametr umožňující specifikovat důvod označení testu tímto atributem.

Testy označené *Explicit* atributem jsou při provádění testů ignorovány, ledaže by byly explicitně vybrány. V případě, že je test ignorován z důvodu *Explicit* atributu, tak nebude spuštěn a ani nebude obsažen ve výsledku testů.

```
using System;
using NUnit.Framework;
namespace fastUnit.Samples.Tests
{
    [TestFixture]
    [Explicit("The reason why test fixture is marked as explicit.")]
    public class ExplicitFixture
    {
        [Test]
        public void Test01()
        {     // ...
        }
    }
}
```

Obrazek 4.11: Ukázka použití *Explicit* atributu na úrovni testovací třídy.

```
using System;
using NUnit.Framework;
namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class ExplicitTests
    {
        [Test]
        [Explicit("The reason why test is marked as explicit.")]
        public void ExplicitTest()
        {     // ...
        }
    }
}
```

Obrázek 4.12: Ukázka použití *Explicit* atributu na úrovni testu.

4.2.10 Atribut Ignore

Atribut *Ignore* slouží pro označení testů, které mají být během spuštění přeskočeny. Atributu lze využít v případech, kdy je požadováno aby test či skupina testů nebyly po určitou dobu spouštěny. Typickým případem je situace, kdy daný test či skupina testu hlásí chyby, které však z nějakých důvodů nelze v krátké době odstranit.

Označení testu atribute *Ignore* by mělo být pouze dočasné. Na druhou stranu použití atributu je lepší variantou než daný test zakrýt do komentáře či úplně odstranit. Test bude nadále součástí testovacího modulu, bude kompilován s ostatními testy, bude

veden v přehledu dostupných testů, ale nebude spouštěn a ve výsledku testů bude označen jako přeskočený.

Atributem *Ignore* lze označit jak celé testovací třídy tak i jednotlivé testy. Atribut má jeden volitelný parametr umožňující specifikovat důvod označení testu tímto atributem.

Je-li test ignorován z důvodu *Ignore* atributu, tak nebude spuštěn, ale bude obsažen ve výsledku testů, ve kterém bude označen jako přeskočený.

```
using System;
using NUnit.Framework;
namespace fastUnit.Samples.Tests
{
    [TestFixture]
    [Ignore("The reason why you ignore the test fixture.")]
    public class IgnoreFixture
    {
        [Test]
        public void Test01()
        {     // ...
        }
    }
}
```

Obrazek 4.13: Ukázka použití *Ignore* atributu na úrovni testovací třídy.

```
using System;
using NUnit.Framework;
namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class IgnoreTests
    {
        [Test]
        [Ignore("The reason why you ignore the test.")]
        public void IgnoreTest()
        {     // ...
        }
    }
}
```

Obrázek 4.14: Ukázka použití *Ignore* atributu na úrovni testu.

4.2.11 Atribut Category

Při větším množství testů obsažených v jednom testovacím modulu může být užitečné rozčlenit dané testy do logických skupin. Nástroj FastUnit podporuje členění testů do skupin pomocí atributu *Category*. Atribut *Category* slouží pro zařazení testu do kategorie, která je uvedena jako parametr atributu. Atributem *Category* lze označit jak celé testovací třídy tak i jednotlivé testy. Testovací třídy i jednotlivé testy mohou být zařazeny do jedné či více kategorií. Na základě tohoto členění lze vybírat množiny testů, které budou spuštěny.

V případě, že budou při spuštění testů specifikovány kategorie testů, tak pouze ty testy, které jsou zařazeny do specifikovaných kategorií, budou spuštěny. Testy nezařazené do specifikovaných kategorií nebudou spuštěny a ani nebudou obsaženy ve výsledku testů.

```
using System;
using NUnit.Framework;
namespace fastUnit.Samples.Tests
{
    [TestFixture]
    [Category("MyCategory01")]
    public class CategorizedFixture
    {
        [Test]
        public void Test01()
        {     // ...
        }
    }
}
```

Obrazek 4.15: Ukázka použití *Category* atributu na úrovni testovací třídy.

```
using System;
using NUnit.Framework;
namespace fastUnit.Tests.Samples
{
    [TestFixture]
    public class CategorizedTests
    {
        [Test]
        [Category("MyCategory02")]
        public void Test01()
        {     // ...
        }
    }
}
```

Obrázek 4.16: Ukázka použití *Category* atributu na úrovni testu.

5 Nová funkcionalita přidaná nástrojem FastUnit

V předchozí kapitole byla prezentována základní funkcionalita podporovaná nástrojem FastUnit. Tato funkcionalita zcela vycházela z funkcionality definované nástrojem NUnit a je s ní zpětně kompatibilní.

Jedním z důvodu vzniku nástroje FastUnit bylo rozšířit funkcionalitu definovanou nástrojem NUnit o funkcionalitu novou, která by uživatelům poskytovala větší možnosti a usnadnila vývoj složitějších testů. FastUnit rozšiřuje funkcionalitu následujícími směry.

- Podpora pro testování výkonu modulů.
- Zjednodušení psaní opakujících se testů zavedením parametrizovatelných testů.
- Podpora pro paralelní testy.

5.1 Testování výkonu modulů

Většina testovacích nástrojů měří dobu trvání jednotlivých testů a naměřené hodnoty zobrazují v konečném přehledu společně s výsledky testování. Bohužel až na výjimky testovací nástroje nenabízí možnost zahrnout naměřený čas jako jednu z podmínek nutnou pro úspěšné dokončení testu. Nástroj FastUnit tuto možnost podporuje.

5.1.1 Motivace

Na začátku tohoto odstavce je nutné poznamenat, že cílem zavedení podpory pro testování (v tomto kontextu lépe řečeno kontrole) výkonu modulů nebylo vytvořit nástroj, který by sloužil pro měření či analýzu rychlosti testovaných modulů, ale nástroj, který slouží pro kontrolu jednou dosažené rychlosti. Nástroje sloužící pro měření a analýzu kódu patří do jiné skupiny nástrojů, než nástroje typu FastUnit a jsou také používány v jiných fázích vývoje softwaru - například v průběhu integračních či výkonnostních testů.

Hlavním cílem zavedení podpory pro testování výkonu modulů je nabídnout možnost rychlé a automatické detekce výrazného zpomalení testovaného kódu. Nástroj FastUnit nabídne mechanismus, pomocí kterého lze čelit situacím, kdy během úpravy existujícího kódu dojde k chybám, které mají za následek nežádoucí zpomalení testovaných částí. Požadavky na rychlost a automatičnost detekce jsou v tomto případě zásadní.

- **Rychlost** – Je nutné, aby případné chyby, v tomto případě výkonnostního rázu, byly detekovány co možná nejdříve, ideálně již ve fázi vývoje. Detekování podobných chyb v pozdějších fázích vývoje softwaru může mít mnohem závažnější důsledky, neboť chyby vedoucí k výraznému zpomalení softwaru jsou většinou architektonického charakteru a jejich odstranění bývá časově náročné.

- **Automatičnost** – Při větším množství testů není v lidských silách kontrolovat naměřené časy jednotlivých testů ručně a po každém spuštění testů. Z tohoto důvodu je nutné odstranit z procesu rozhodování o úspěšnosti testů lidský prvek. Veškeré činnosti spojené s vyhodnocováním výsledků testování musí byt plně automatizované.

5.1.2 Uvažovaná řešení

Při návrhu řešení bylo nutné odpovědět na základní otázku a sice zvolit správnou granularitu měření. V úvahu byly brány následující dvě možnosti.

1. První možností bylo zvolit co možná nejmenší granularitu a umožnit kontrolovat rychlost s velkou přesností, a to až na úroveň jednotlivých operací. Zde by se předpokládalo, že jeden test bude kontrolovat pouze jednu testovanou operaci. Rychlost jednotlivých operací by bylo možné specifikovat přímo očekávaným časem nebo počtem operací provedených za časový úsek. Například bylo uvažováno o zavedení atributu *OperationsPerSecond*, který by umožnil zadat minimální počet, kolikrát se daná operace má provést v intervalu jedné sekundy. FastUnit by změřil čas trvání jednoho testu a následně by dopočítal, kolikrát by se operace zopakovala během celé sekundy. Nevýhoda tohoto řešení spočívá především v požadavku na velkou přesnost měření, kde i malá nepřesnost výrazně ovlivní celkový výsledek. Toto řešení bylo nakonec zamítnuto.
2. Druhou možností bylo nesnažit se měřit jednotlivé operace, ale zaměřit se na déle trvající testy, které testují sady operací. Výhoda této možnosti oproti předcházející by byla ve zmírnění požadavku na přesnost měření. V rámci tohoto řešení bylo uvažováno o navržení atributu *Duration*, který by umožnil zadat očekávaný čas běhu testu. FastUnit by v průběhu testování měřil čas každého testu a ten po jeho skončení porovnal s očekávaným časem.

Pro menší nároky na přesnost měření byla zvolena druhá možnost a atribut *Duration*.

Druhým základním problémem, který bylo nutné řešit při návrhu testování výkonu modulů, byla závislost výsledků na výkonu testovacího stroje. Test, který na jednom testovacím stroji trvá deset sekund může na jiném stroji trvat 20 sekund a naopak. Jedním z uvažovaných řešení bylo zavést určitý *činitel*, který by byl počítán za běhu a který by odrážel výkon použitého testovacího stroje. V průběhu vyhodnocování testu by byl *činitel* použit pro úpravu naměřeného času před jeho porovnáním s časem očekávaným. Byly uvažovány následující dva způsoby výpočtu *činitele*.

1. Hodnota činitele se za běhu určí z aktuální hardwarové konfigurace stroje. Do výpočtu by byly zahrnuty nejdůležitější hardwarové součásti, které mají největší vliv na výkon stroje – například: typ a výkon procesoru, velikost operační paměti, rychlost pevného disku apod.

   Výhody: Rychlost výpočtu hodnoty činitele. Výpočet hodnoty by byl deterministický, tj. opakovaný výpočet i výpočet na různých strojích o stejné konfiguraci by vždy vracel stejnou hodnotu.

   Nevýhody: Výpočet by byl poměrně komplikovaný a vyžadoval by dostupnost řady statistických dat o výkonech jednotlivých hardwarových součástí. Například by byla zapotřebí tabulka srovnávající výkon existujících procesorů s vybraným referenčním procesorem apod. Další nevýhodou by bylo, že každý stroj nedosahuje teoretického výkonu, který by odpovídal jeho konfiguraci. Velice záleží na individuálním nastavení stroje a jeho celkové vyváženosti.
2. Hodnota činitele se určí na základě zkušebního kódu. Před spuštěním vlastního testu či série testů by byl spuštěn zkušební kód, pomocí kterého by byl odhadnut výkon stroje. Kód by mohl obsahovat smyčky obsahující nejpoužívanější operace – například: práci s celými čísly, práci s čísly s desetinnou tečkou, alokace paměti a jiné.

Výhody: Hodnota by lépe odrážela skutečný výkon testovacího stroje.

Nevýhody: Výpočet hodnoty není deterministický, každé spuštění testů by mohlo dát jiný výsledek. Zkušební kód by bylo nutné nechat běžet po delší dobu, aby se minimalizovali chyby způsobené nepřesností měření či působením vnějších faktorů, např. neočekávaného a nekontrolovaného spuštění *garbage collectoru*, *swapování* paměti apod. Bohužel prodloužením běhu zkušebního kódu by se celá navrhovaná funkcionalita stala v praxi nepoužitelnou. Při interaktivním testování by nutnost čekat řádově deset sekund či více před každým spuštěním testů byla pro uživatele frustrující a vedla by k tomu, že by funkcionalita nebyla používána.

Po analýze výhod a nevýhod jednotlivých způsobů určení výkonu testovacího stroje bylo nakonec rozhodnuto myšlenku *činitele* zcela opustit. Požadovaný čas testů zadaný atributem *Duration* je tedy přímo porovnáván s naměřeným časem. Velkou výhodou tohoto řešení je jeho jednoduchost, jak z pohledu implementace, tak z pohledu uživatele. Uživatel zadává přímo očekávaný čas běhu testů, nemusí ho nijak složitě přepočítávat, aby zjistil, jaká hodnota bude ve skutečnosti použita.

Problém se závislostí výsledků testů na výkonu testovacího stroje nebyl sice zcela vyřešen, na druhou stranu navrhované řešení je velice jednoduché a v praxi dostatečné. Vysvětlení:

- V rámci firem, které se zabývají vývojem softwaru na určité úrovni, existuje ve většině případů standardizace používaného hardwaru a tudíž lze vyvíjené testy nastavit pro konkrétní testovací prostředí.
- Hlavním cílem zavedení podpory pro testování výkonu modulů bylo nabídnout možnost detekce *výrazného* zpomalení testovaného kódu, nikoliv zpomalení v řádech procent či desítek procent. Je-li skutečná doba trvání testu 10 sekund, lze test označit očekávaným časem například 15 či 20 sekund. I v tomto případě test plní svou požadovanou funkci a navíc je i částečně tolerantní k rozdílným výkonům testovacích strojů.

5.1.3 Atribut Duration

Atribut *Duration* slouží pro označení testů, u kterých je nutné zajistit určitý stupeň výkonu testovaného kódu. Jestliže je test označen tímto atributem, je od testu očekáváno, že čas nutný pro provedení daného testu je menší, než čas specifikovaný atributem. Atribut *Duration* má jeden povinný parametr, který slouží ke specifikaci hodnoty požadovaného času v milisekundách.

```
using System;
using NUnit.Framework;
namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class DurationSample01
    {
        [Test]
        [Duration(5000)]
        public void Test01()
        {    // ...
        }
    }
}
```

Obrázek 5.1: Ukázka použití atributu *Duration*.

Test *Test01* na obrázku 5.1 skončí úspěšně, pokud doba jeho běhu nepřekročí pět sekund. V opačném případě test skončí chybou. Splnění časového limitu není postačující podmínkou pro úspěšné dokončení testu, nýbrž pouze podmínkou nutnou. Test musí splnit i ostatní podmínky na něj kladené pomocí ostatních atributů. Například, je-li test označen současně i atributem *ExpectedException*, tak test musí skončit v požadovaném čase a s požadovanou výjimkou, jinak bude označen za chybný.

5.1.4 Omezení

Okolnosti, které je nutné brát v úvahu při použití atributu *Duration*.

- Do času testu se započítávají i režijní náklady testovacího nástroje spojené s dynamickým spouštěním testů. Při kontrole náročnějších a déle trvajících testů je však tento vliv zanedbatelný.
- První spuštění testu je vždy pomalejší než jeho opakované spuštění, je to dáno způsobem načítání modulů a jednotlivých tříd na platformě Microsoft .NET. Příprava prvního spuštění testu, který odkazuje větší množství testovaných modulů a tříd, může zabrat nezanedbatelné množství času. V některých případech i v řádu sekund.
- Přesnost měření. Platforma Microsoft .NET nabízí standardně možnost měření času s přesností 16 milisekund. To znamená, že při kontrole testu, který ve skutečnosti trvá 4 milisekundy, může být v jednom případě naměřen čas 0 milisekund a v druhém případě naměřen čas 16 milisekund. Platforma Microsoft .NET podporuje i přesnější měření času pomocí tzv. *high performance counters*, ale režijní náklady tohoto způsobu měření jsou mnohonásobně větší než u standardně nabízeného mechanismu. Na druhou stranu cílem nástroje FastUnit není měřit trvání testů v řadech milisekund, ale v řádech sekund. Z toho důvodu je použitá přesnost měření dostačující.
- Výkon testovacího stroje. Výkon testovacího stroje je velmi důležitým faktorem, který je nutné brát v úvahu při volbě požadovaného času na provedení testu. Použití atributu *Duration* porušuje dvě ze základních podmínek kladených na testy - podmínku na úplnou nezávislost testů na okolním prostředí a podmínku na přenositelnost testů. V tomto případě má výkon testovacího stroje vliv na konečný výsledek testů. Použití atributu *Duration* je v tomto smyslu omezující.

5.2 Parametrizovatelné testy

Ve většině případu lze testy navrhnout a napsat bez nutnosti jejich parametrizace, neboli testy nevyžadují žádné vstupní parametry. Existují však situace, kdy bylo vhodné mít možnost do testů vstupní parametry předat. Jedná se například o situaci, kdy je testován nějaký algoritmus na množinách vstupních parametrů a jsou očekávány množiny výstupních hodnot.

5.2.1 Motivace

Důvody, které vedly k zavedení parametrizovatelných testů budou prezentovány na následujícím příkladě. Je dána třída *DiscountCalculator*, jejíž úkolem je počítání slev při odběru zboží nad určitou hodnotu. Při odběru do 1000,- Kč by neměla být udělena žádna sleva, při odběru nad 1000,- Kč (včetně) by měla být udělená sleva

5% a při odběru nad 5000,-Kč (včetně) by měla být udělena sleva 10%. Při zadání záporné hodnoty zboží by měla být vyhozena výjimka typu *ArgumentException*.

S využitím pouze funkcionality podporované nástrojem NUnit, tj. bez možnosti předat do testů vstupní parametry, je nutné napsat nejméně čtyři samostatné testy, jeden test pro každou z požadovaných limitních hodnot třídy *DiscountCalculator* (obr 5.2).

```
using System;
using NUnit.Framework;
namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class DiscountCalculatorTests01
    {
        [Test]
        public void DiscountCalculatorTest01()
        {   DiscountCalculator calc = new DiscountCalculator();
            Assert.AreEqual(0, calc.GetPercentage(999));
        }

        [Test]
        public void DiscountCalculatorTest02()
        {   DiscountCalculator calc = new DiscountCalculator();
            Assert.AreEqual(5, calc.GetPercentage(1000));
        }

        [Test]
        public void DiscountCalculatorTest03()
        {   DiscountCalculator calc = new DiscountCalculator();
            Assert.AreEqual(10, calc.GetPercentage(5000));
        }

        [Test]
        [ExpectedException(typeof(ArgumentException))]
        public void DiscountCalculatorTest04()
        {   DiscountCalculator calc = new DiscountCalculator();
            calc.GetPercentage(-1);
        }
    }
}
```

Obrázek 5.2: Ukázka testů třídy *DiscountCalculator* bez využití parametrizovatelných testů.

Testy vytvořené pro třídu *DiscountCalculator* (obr 5.2) jsou sice formálně v pořádku a plně funkční, ale způsob jejich zápisu má řadu vad.

- Testy obsahují řadu duplicitního kódu.
- Kód testů je poměrně těžký na údržbu. Při změně rozhraní třídy je nutné kód měnit na více místech.
- Testy jsou nepřehledné – jak množstvím, tak i nic neříkajícími názvy.

Cílem nástroje FastUnit bylo definovat mechanismy, které by umožnily testy podobné testům pro třídu *DiscountCalculator* (obr 5.2) napsat jednodušším a přehlednějším způsobem. Hlavní požadavky byly:

- Odstranění duplicit kódu.
- Zjednodušení údržby testů.
- Zkrácení a zpřehlednění testovacího kódu.

5.2.2 Uvažovaná řešení

Prvním uvažovaným řešením bylo nepřidávat žádnou novou funkcionalitu a pouze využít prostředků nabízených programovacím jazykem C#. Testovací kód by bylo možné přepsat následujícími dvěma způsoby:

1. Společné části testů přesunout do sdílené privátní metody testovací třídy. V jednotlivých testech pak pouze delegovat volání do privátní metody s požadovanými parametry.

   Výhody: Toto řešení je jednoduché a v řadě případů splní svůj účel. Kód je již kratší a bylo odstraněno velké množství duplicitního kódu. Zjednodušila se údržba testů.

   Nevýhody: Testovací kód obsahuje stále ještě mnoho duplicitních částí – například: definice jednotlivých testů a delegování volání do privátní metody. Kód je již sice kratší, ale stále není přehledný.

2. Společné části testů přesunout do sdílené privátní metody testovací třídy a jednotlivé testy sloučit do jednoho velkého testu.

   Výhody: Kód je již přehledný a byla odstraněna řada duplicitního kódu nutného pro definování jednotlivých testů.

   Nevýhody: Spojení testů porušuje jeden ze základních požadavků TDD, a sice že rozdílné funkcionality by měly být testovány v oddělených testech. Požadavek má svůj praktický význam: dojde-li během provádění k chybě, zbývající část testu již nebude provedena. To znamená, že nebude otestována celá funkcionalita. Jeden společný test taky nelze použít na testování hodnot, u kterých se očekává vyhození výjimky. Podobně jako v předchozím případě by zbývající část funkcionality nebyla otestována.

Výše uvedená řešení se sice ukázala jako jednoduchá a nevyžadovala přidání žádné nové funkcionality, nicméně plně neřešila kladené požadavky. Bylo tedy rozhodnuto o nutnosti rozšířit definici testů o možnost předávat vstupní parametry do testovacích metod. Základní otázkou bylo, jakým způsobem umožnit uživateli zadávat hodnoty, které budou předány do testovací metody při jejím spuštění.

1. Prvním uvažovaným řešením bylo zavést registrační metodu, pomocí které by testovací třída předala nástroji informace o svých parametrizovatelných testech a u každého testu seznam hodnot.

   Výhody: Předávané hodnoty by mohly byt konstruovány dynamicky při běhu testů.

   Nevýhody: Nevýhody, kvůli kterým bylo toto řešení zamítnuto, byly dvě. První nevýhodou bylo, že definice hodnot by byla držena odděleně od definice vlastního testu, což by mělo za následek zhoršení čitelnosti kódu. Druhou nevýhodou bylo, že by toto řešení zavedlo nový mechanismus pro definování testů a sice registrační metody. FastUnit doposud používal pro definici testů výhradně atributy.

2. Druhým uvažovaným řešením bylo zavést nový atribut *Data*. Atributem data by se označovaly jednotlivé parametrizovatelné testy a sloužil by pro zadávání hodnot, které se předají testovací metodě při jejím spuštění.

   Výhody: Využívá již používaných mechanismů pro definování testů, nezavádí žádný nový. Definice testu i definice testovaných hodnot je držena u sebe.

Nevýhody: Předávané hodnoty nelze vytvářet dynamicky, ale musí být známe již během překladu.

Po analýze výhod a nevýhod uvažovaných řešení bylo nakonec rozhodnuto pro variantu s atributem *Data*.

5.2.3 Atribut Data

Atribut *Data* slouží pro předání parametrů do jednotlivých testů v čase jejich spuštění. Jeden test může být označen více atributy *Data*, každý atribut pak odpovídá jednomu spuštění daného testu se specifikovanými parametry. Atributem *Data* může být označená každá metoda testovací třídy splňující následující požadavky.

- Metoda je veřejná.
- Parametry vyžadované metodou si odpovídají s parametry specifikovanými u atributu.
- Návratová hodnota metody je typu *void*.
- Metoda je označena atributem *Test* nebo *ParallelTest*. Atribut *ParallelTest* bude vysvětlen v odstavci 5.3.3, který je věnován paralelním testům.

Jestliže parametry specifikované u atributu *Data* neodpovídají parametrům vyžadovaných metodou, bude pokus o spuštění testu s danými parametry zamítnut a test bude označen ve výsledku testů jako chybný.

```
using System;
using NUnit.Framework;
namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class DataAttributeSample01
    {
        [Test]
        [Data(true, 2, 3, "text1")]
        [Data(true, 2, 6.1, "text2")]
        [Data(false, 5, -34.4, "text3")]
        public void Test(bool a, int b, double c, string d)
        {     // ...
        }
    }
}
```

Obrázek 5.3: Ukázka použití atributu *Data*.

Při spuštění testovací třídy *DataAttributeSample01* (obr 5.3) bude test *Test* spuštěn celkem třikrát, jednou pro každý atribut *Data*.

Způsoby specifikování parametrů

Atribut *Data* akceptuje neomezený počet parametrů. Této vlastnosti bylo dosaženo použitím konstrukce *params object[] args* při deklaraci parametru v konstruktoru atributu. Atribut *Data* tedy umožňuje při svém použití specifikovat nula a více parametrů, které mohou být zadány jednotlivě nebo najednou v podobě jednoho pole typu *object*. Zápisy atributu *Data* na obrázku 5.4 jsou ekvivalentní.

```
[Data(1, 2, 3, 4, "text1", "text2", "text3", "text4")]
[Data(new object[] {1, 2, 3, 4, "text1", "text2", "text3", text4"})]
```

Obrázek 5.4: Ukázka možností způsobu zadání parametru atributu *Data*.

5.2.4 Atribut Data a očekávání výjimky

Nástroj FastUnit umožňuje použití parametrizovatelných testů i v případě, kdy je při spuštění testů s parametry očekávána výjimka. Atribut *Data* obsahuje tři volitelné pojmenované parametry sloužící pro specifikaci očekávané výjimky (parametry *ExpectedException*, *ExpectedMessage* a *MatchType*). Význam jednotlivých pojmenovaných parametrů i způsob jejich použití je stejný jako v případě atributu *ExpectedException* a je popsán v odstavci 4.2.8.

```
using System;
using NUnit.Framework;
namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class DataAttributeSample02
    {
        [Test]
        [Data(1000, 10)]
        [Data(-1, 0, ExpectedException=typeof(ArgumentException))]
        [Data(-2, 0, ExpectedException=typeof(ArgumentException),
                     ExpectedMessage="less than zero",
                     MatchType=MessageMatch.Contains)]
        public void Test(int invoice, int expectedDiscount)
        {     // ...
        }
    }
}
```

Obrázek 5.5: Ukázka specifikace očekávané výjimky v rámci atributu *Data*.

Atribut *Data* umožňuje, aby v rámci jednoho testu bylo definováno více dat, u kterých je očekávána výjimka (obr. 5.5). Taktéž očekávané výjimky pro jednotlivá data se mohou lišit. Na druhou stranu, pokud je stejná výjimka očekávána u všech zadaných dat, lze označit test jako celek atributem *ExpectedException* a specifikovaná výjimka pak bude očekávána u všech zadaných dat (obr. 5.6).

```
using System;
using NUnit.Framework;
namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class DataAttributeSample03
    {
        [Test]
        [ExpectedException(typeof(ArgumentException))]
        [Data(-1, 0)]
        [Data(-2, 0)]
        public void Test(int invoice, int expectedDiscount)
        {     // ...
        }
    }
}
```

Obrázek 5.6: Ukázka použití atributu *Data* v kombinaci s atributem *ExpectedException*.

V případě, kdy v rámci jednoho testu je očekávaná výjimka určená jak atributem *ExpectedException* tak atributem *Data*, je při vyhodnocování testů použita definice výjimky zadaná atributem *Data*.

5.2.5 Atribut Data a testování výkonu modulů

Nástroj FastUnti umožňuje použití parametrizovatelných testů i v kombinaci s testováním výkonů modulů, tj. v situaci kdy je u testu kontrolováno jeho skončení v požadovaném čase. Atribut *Data* obsahuje volitelný pojmenovaný parametr *Duration*, který slouží ke specifikaci hodnoty požadovaného času v milisekundách. Význam i způsob použití parametru *Duration* je stejný jako v případě atributu *Duration* a je popsán v odstavci 5.1.3.

```
using System;
using NUnit.Framework;
namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class DataAttributeSample04
    {
        [Test]
        [Data(1000, 10, Duration=1000)]
        [Data(2000, 20, Duration=2000)]
        public void Test(int invoice, int expectedDiscount)
        {    // ...
        }
    }
}
```

Obrázek 5.7: Ukázka specifikace očekávaného výkonu modulu v rámci atributu *Data*.

Atribut *Data* umožňuje, aby v rámci jednoho testu bylo definováno více dat, u kterých je kontrolován čas běhu testu (obr. 5.7). Taktéž požadované časy se mohou lišit pro jednotlivá data. Na druhou stranu, pokud je u jednotlivých dat požadován stejný čas běhu testů, lze označit test jako celek atributem *Duration* a zadaný čas pak bude použit při kontrole časů běhu testů pro všechna data (obr. 5.8).

```
using System;
using NUnit.Framework;
namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class DataAttributeSample05
    {
        [Test]
        [Duration(1000)]
        [Data(1000, 10)]
        [Data(2000, 20)]
        public void Test(int invoice, int expectedDiscount)
        {    // ...
        }
    }
}
```

Obrázek 5.8: Ukázka použití atributu *Data* v kombinaci s atributem *Duration*.

V případě, kdy v rámci jednoho testu je požadovaný čas určen jak atributem *Duration*, tak atributem *Data*, je při vyhodnocování testu použita hodnota definovaná atributem *Data*.

5.2.6 Příklad využití parametrizovatelných testů

Využití parametrizovatelných testů bude prezentováno v ukázce testů pro třídu *DiscountCalculator*, která byla popsána v odstavci 5.2.1. Všechny čtyři testy uvedené v ukázce na obrázku 5.2 lze přepsat jako jeden parametrizovatelný test, který je označen čtyřmi atributy *Data*. Výsledek je prezentován na obrázku 5.9 .

```
using System;
using NUnit.Framework;
namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class DiscountCalculator02
    {
       [Test]
       [Data(999, 0)]
       [Data(1000, 5)]
       [Data(5000, 10)]
       [Data(-1, 0, ExpectedException=typeof(ArgumentException))]
       public void Test(int invoice, int percentage)
       {   DiscountCalculator calc = new DiscountCalculator();
           Assert.AreEqual(invoice, calc.GetPercentage(percentage));
       }
    }
}
```

Obrázek 5.9: Ukázka testů třídy *DiscountCalculator* s využitím parametrizace testů.

5.3 Paralelní testy

Při vývoji modulů, které jsou určeny pro běh v paralelním prostředí, je nutné mít k dispozici nástroje, které dokáží vyvíjené moduly adekvátním způsobem otestovat. V tomto případě již nestačí nástroje, které umožňují otestovat testovaný modul pouze v rámci testů, které jsou prováděny sekvenčně za sebou v jednom vlákně. Vyvíjené moduly musí být testovány způsobem, který co možná nejvíce odpovídá jejich budoucímu nasazení. To znamená, že jsou potřeba nástroje, pomocí kterých je možné vytvořit testy, které budou spouštěny paralelně.

5.3.1 Motivace

V současné době bohužel testovací nástroje nenabízejí žádnou nebo když už, tak pouze velice malou podporu pro testování modulů určených pro běh v paralelním prostředí. Podle množství a typu podpory lze existující nástroje rozdělit do tří skupin.

- **Žádná podpora –** Do této skupiny patří nástroje, které nenabízejí žádnou podporu pro tvorbu paralelní testů. Mezi tyto nástroje lze zařadit např. NUnit [4], Zanebug [6] a csUnit [8]. Při použití těchto nástrojů pro tvorbu paralelních testů je nutné naimplementovat vlastní mechanismus, který zajistí spuštění požadovaných testů ve více vláknech, počká na dokončení jednotlivých testů a výsledek jako celek oznámí testovacímu nástroji.
- **Podpora pomocí přídavných modulů** – Do této skupiny patří nástroje, které sice nenabízejí vlastní podporu pro tvorbu paralelních testů, ale umožňují tuto podporu přidat pomocí přídavných modulů. Mezi tyto nástroje lze zařadit například JUnit [10]. Použití těchto nástrojů pro tvorbu paralelních testů je již mnohem snazší, než v případě první skupiny, ale stále je dost komplikované a přidává závislost na softwaru třetí strany.

- **Základní podpora** – Do této skupiny patří nástroje, které již nabízejí vlastní podporu pro psaní paralelních testů. Mezi tyto nástroje patří například MbUnit [5] a TestNG [9]. Podpora nabízená těmito nástroji je ovšem většinou základní a je vhodná pro tvorbu pouze jednoduchých testů. Při tvorbě složitějších paralelních testů je opět nutné přidávat vlastní mechanismy.

Hlavní motivací vzniku nástroje FastUnit bylo vytvořit testovací nástroj, který by umožňoval testování modulů v paralelním prostředí a nabízel i dostatečné prostředky, které by usnadňovaly vývoj i složitějších paralelních testů. V praxi se při tvorbě paralelních testů v existujících nástrojích nejčastěji naráželo na následující dvě omezení:

- Stávající řešení nenabízely vlastní mechanismus, který by umožnil jednoduše odlišit jednotlivé instance běžících metod.
- Stávající řešení neumožňovaly spustit více druhů testovacích metod najednou. Bylo možné spustit pouze jednu testovací metodu opakovaně.

Výše zmíněná omezení sice nebyla v praxi nepřekonatelná, ale vyžadovala vývoj vlastních mechanismů, které by je dokázaly obejít. Cílem nástroje FastUnit bylo tedy definovat vlastní mechanismy, které by požadovanou funkcionalitu nabízely uživateli. Hlavní požadavky kladené na přidanou funkcionalitu byly:

- Možnost identifikovat jednotlivé instance běžících metod.
- Možnost spustit více druhů testovacích metod najednou.

Příklad podpory paralelních testů konkurenčními nástroji

Nástroj MbUnit podobně jako nástroj FastUnit používá pro definici a popis testů atributy. Za účelem podpory paralelních testů zavedl nástroj MbUnit atribut *ThreadedRepeat* (obr. 5.10). Je-li test označen tímto atributem, je během provádění testů spuštěn ve více vláknech paralelně. Atribut *ThreadedRepeat* má jeden povinný parametr, kterým je možno zadat počet požadovaných vláken.

```
using System;
using MbUnit.Framework;
namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class ThreadedRepeatSample
    {
        [Test]
        [ThreadedRepeat(10)]
        public void Test01()
        {    // ...
        }
    }
}
```

Obrázek 5.10: Ukázka definice paralelních testů v nástroji MbUnit.

Při spuštění testu *Test01* (obr. 5.10) bude testovací metoda spuštěna paralelně v deseti vláknech.

5.3.2 Uvažovaná řešení

Návrh a vývoj paralelních testů byl rozdělen do tří fází. První fáze se zaměřila na návrh základní definice paralelního testu. Cílem této fáze bylo získat podporu pro paralelní testy, která by byla srovnatelná s podporou nabízenou ostatními existujícími nástroji. Další fáze byly následně věnovány snaze rozšířit základní definici a přidat novou požadovanou funkcionalitu.

Fáze 1 – základní definice

Při návrhu mechanismu, jak definovat paralelní testy se nástroj FastUnit inspiroval atributem *ThreadedRepeat* nástroje NUnit (obr. 5.10). Byl zaveden atribut *ParallelTest*, který obsahuje jeden povinný parametr *threadCount*. Parametr *threadCount* určuje počet vláken, ve kterých se má označená metoda spustit.

V této fázi vývoje byl hlavní rozdíl atributu *ParallelTest* od atributu *ThreadedRepeat* v tom, že atribut *ParallelTest* slouží pro označení paralelního testu samostatně bez nutnosti kombinace s atributem *Test*. Důvodem tohoto rozdílu byla snaha vytvořit atribut, který by byl nezávislý na atributu *Test* a který by bylo možné v dalších fázích vývoje dále rozšiřovat.

Důvody, pro které bylo zvoleno řešení za pomocí atributu *ParallelTest* a nebyly už hledány jiné možnosti, byly následující:

1. Jednalo se o jednoduché řešení které plně umožňovalo požadovanou funkcionalitu.
2. Bylo to řešení, které pro definici paralelního testu používalo atribut.

Fáze 2 – identifikace jednotlivých instancí běžících metod

Důvody, které vedly ke snaze o zavedení této funkcionality jsou vysvětleny v části 5.3.5, která je věnována dokumentaci výsledného řešení.

V průběhu návrhu této funkcionality byly uvažovány následující řešení.

1. První uvažovanou možností bylo nerozšiřovat stávající definici paralelních testů, ale pouze využít již existujících mechanismů. Konkrétně se uvažovalo o pojmenovávání pracovních vláken, ve kterých jsou spuštěny jednotlivé instance testovacích metod. Pracovní vlákna by byla pojmenována jednoznačně – například pořadovým číslem instance v rámci paralelního testu. Takto pojmenována vlákna by následně mohla být použita pro identifikaci jednotlivých instancí testovacích metod.

   Výhody: Jednoduché řešení, které plně splňuje požadavek.

   Nevýhody: Zneužívá cizích mechanismů, v tomto případě pojmenování vláken.

2. Druhým zvažovaným řešením bylo rozšířit definici testovací metody pro paralelní testy o jeden povinný parametr. Do přidaného parametru by se při spuštění metody předával jednoznačný identifikátor – například pořadové číslo instance testovací metody v rámci paralelního testu.

   Výhody: Opět jednoduché řešení, které již využívá pouze vlastních mechanismů.

   Nevýhodu: Parametr je předáván vždy, i když by nebyl uživatelem vyžadován.

3. Předchozí uvažované řešení s rozšířením definice testovací metody o jeden parametr pak vedlo k myšlence využít již existující atribut *Data*. Atribut *Data* by

bylo možné použít i pro definici paralelního testu a sloužil by pro zadávání dat, která mají být předána do instance testovací metody při jejím spuštění.

Výhody: Atribut *Data* již existoval a nebylo tedy nutné definovat nový atribut. Možnost předat libovolný počet parametrů libovolných typů nabízí mnohem silnější funkcionalitu než byla původně vyžadována. Atribut *Data* nemusí sloužit pouze pro identifikaci instance testovací metody, ale je plně na uživateli, která data si předá do spuštěné instance.

Nevýhody: Atribut *Data* se definuje ručně. V případě identifikace většího množství instancí testovací metody by bylo nutné ručně zadat jednoznačné identifikátory pomocí atributu *Data*.

Po přezkoumání všech výhod a nevýhod bylo jednoznačně vybráno řešení s atributem *Data*, neboť toto řešení nabízí pro uživatele mnohem větší možnosti.

Fáze 3 – spuštění více druhů testovacích metod najednou

V průběhu návrhu této funkcionality byly uvažovány následující řešení.

1. Prvním uvažovaným řešením bylo použití jmenné konvence. Jména testovacích metod, které by měly být spuštěny společně, by měly odpovídat nějakému společnému vzoru. Tato myšlenka nebyla dále ani rozvíjena pro níže popsané nevýhody.

   Nevýhody: Jednalo by se o zavedení nového způsobu definice testů. FastUnit doposud nepoužíval pro identifikací testů jmennou konvenci a pojmenovávání testovacích metod nechával čistě na uživateli.

2. Druhou uvažovanou možností bylo zavedení registrační metody. Testovací třída by musela poskytnout testovacímu nástroji seznam svých testovacích metod určených pro paralelní testy a jejich rozdělení do skupin.

   Výhody: Členění testovacích metod do jednotlivých paralelních testů by bylo drženo na jednom místě.

   Nevýhody: Nevýhody tohoto řešení jsou obdobné jako v předcházejícím případě. FastUnit nepoužívá pro identifikaci testů registračních metod, ale výhradně atributů. Bylo by zbytečné zavádět registrační metody jenom kvůli této funkcionalitě.

3. Třetí a již slibnější uvažovanou možností bylo zavedení nového atributu. Nový atribut by sloužil pro určení skupiny paralelních testů, ve které by měla být označená metoda spuštěna.

   Výhody: Jednoduché řešení, které splnilo požadovanou funkcionalitu. V případě, kdy by měla být testovací metoda spuštěna samostatně, nebylo by třeba danou metodu značit tímto atributem.

   Nevýhody: Zavádí nový atribut.

4. Poslední uvažovanou možností bylo rozšíření atributu *ParalelTest* o jeden nepovinný parametr *testName*, kterým by se určovala skupina paralelních testů, ve které by měla být označená metoda spuštěna. V případě, kdyby parametr *testName* nebyl zadán, testovací metoda by byla spuštěna samostatně v rámci svého vlastního paralelního testu. Jednalo se o možnost, která by řešila požadovanou funkcionalitu obdobným způsobem jako předchozí možnost, ale bez nutnosti zavedení nového atributu.

Výhody: Nezavádí nový atribut. Parametr *testName* je nepovinný, tj. není ho nutné uvést v případě, kdy by měla být testovací metoda spuštěna samostatně.

Po analýze výhod a nevýhod uvažovaných řešení bylo nakonec rozhodnuto pro variantu s parametrem *testName*.

5.3.3 Atribut ParallelTest

Atribut *ParallelTest* slouží pro označení testů, které mají být spuštěny paralelně ve více vláknech. Atribut má jeden povinný parametr *threadCount* a jeden nepovinný parametr *testName*.

- *threadCount*: Počet vláken ve kterých má být testovaná metoda spuštěna paralelně.
- *testName*: Logické jméno paralelního testu. Slouží pro seskupování testovacích metod, které jsou označeny atributem *ParallelTest* do logických skupin v rámci jedné testovací třídy. Testy patřící do stejné skupiny budou spuštěny současně jako jeden paralelní test. Není-li parametr zadán, je testovací metoda spuštěna samostatně v rámci svého vlastního paralelního testu.

Atributem *ParallelTest* může být označena každá metoda testovací třídy splňující následující požadavky.

- Metoda je veřejná.
- Metoda nevyžaduje žádné parametry. Atribut *ParallelTest* lze kombinovat i s atributem *Data*. (Tato možnost bude prezentována v odstavci 5.3.5).
- Návratová hodnota metody je typu *void*.
- Metoda není označena ani jedním z atributů *Test*, *SetUp*, *TearDown*, *TestFixtureSetUp* a *TestFixtureTearDown*.

V případě, že metoda nesplní některý z výše uvedených požadavků, tak bude označena jako nevalidní a jakýkoliv pokus o její spuštění bude odmítnut.

```
using System;
using NUnit.Framework;
namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class ParallelTestSample01
    {
        [ParallelTest(5, "Test")]
        public void Test01()
        {     // ...
        }
    }
}
```

Obrázek 5.11: Ukázka jednoduchého paralelního testu.

Testovací třída *ParallelTestSample01* zobrazená na obrázku 5.11 obsahuje jeden paralelní test *Test*. Při jeho spuštění bude testovací metoda *Test01* spuštěna v pěti vláknech paralelně.

Parametr *testName* slouží pro seskupování jednotlivých testovacích metod označených atributem *ParallelTest* do logických skupin. Testovací metody patřící do stejné skupiny tvoří jeden paralelní test. Při spuštění paralelního testu jsou všechny

jeho jednotlivé testovací metody spuštěny paralelně v požadovaném počtu vláken. Příklad využití parametru *testName* bude prezentováno na následujícím příkladu (obr. 5.12).

```
using System;
using NUnit.Framework;
namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class ParallelTestSample02
    {

        [ParallelTest(3, "TestA")]
        public void Test01()
        {     // ...
        }

        [ParallelTest(4, "TestA")]
        public void Test02()
        {     // ...
        }

        [ParallelTest(5, "TestB")]
        public void Test03()
        {     // ...
        }

        [ParallelTest(6, "TestB")]
        public void Test04()
        {     // ...
        }
    }
}
```

Obrázek 5.12: Ukázka rozdělení paralelních testů do skupin.

Testovací třída *ParallelTestSample02* zobrazená na obrázku 5.12 obsahuje dva paralelní testy, paralelní test *TestA* a paralelní test *TestB*.

- Paralelní test *TestA* se skládá ze dvou testovacích metod - *Test01* a *Test02*. Při spuštění testu *TestA* bude vytvořeno celkem sedm paralelních vláken, ve třech z nich bude spuštěna testovací metoda *Test01* a ve čtyřech z nich bude spuštěna testovací metoda *Test04*.
- Paralelní test *TestB* se skládá ze dvou testovacích metod - *Test03* a *Test04*. Při spuštění testu *TestB* bude vytvořeno celkem jedenáct paralelních vláken, v pěti z nich bude spuštěna testovací metoda *Test03* a v šesti z nich bude spuštěna testovací metoda *Test04*.

5.3.4 Atribut ParallelTest a životní cyklus testovací třídy

Zavedením paralelních testů se mírně zvýšila složitost životního cyklu testovací třídy, především z pohledu inicializačních a úklidových metod. V tomto odstavci budou upřesněny významy jednotlivých inicializačních a úklidových metod v návaznosti na paralelní testy. Dále budou detailně popsány fáze, které se odehrávají během provádění testů obsahující paralelní testy.

Atribut TestFixtureSetUp

Význam ani čas spuštění metody označené atributem *TestFixtureSetUp* se nemění a zůstává stejný, jak byl popsán v odstavci 4.2.5.

Atribut TestFixtureTearDown

Význam ani čas spuštění metody označené atributem *TestFixtureTearDown* se nemění a zůstává stejný, jak byl popsán v odstavci 4.2.6.

Atribut SetUp

Význam atributu *SetUp* zůstává stejný, jak byl popsán v odstavci 4.2.3. Dochází pouze k jeho upřesnění ve spojitosti s paralelními testy.

Metoda označena atributem *SetUp* je spuštěna pouze jednou před spuštěním každého paralelního testu. Jinak řečeno, metoda není provedena před spuštěním každé jednotlivé testovací metody paralelního testu, ale pouze jednou před paralelním testem jako celkem.

Dojde-li během běhu metody označené atributem *SetUp* k chybě, jednotlivé testovací metody paralelního testu již nebudou spuštěny a paralelní test jako celek bude označen jako chybný.

Atribut TearDown

Význam atributu *TearDown* zůstává stejný, jak bylo popsáno v odstavci 4.2.4. Dochází pouze k jeho upřesnění ve spojitosti s paralelními testy.

Metoda označena atributem *TearDown* je spuštěna pouze jednou pro každý paralelní test po jeho skončení. Metoda je spuštěna až poté, kdy skončí běh všech jednotlivých testovacích metod paralelního testu.

Dojde-li během metody označené atributem *TearDown* k chybě, právě proběhlý test bude jako celek označen jako chybný.

```
using System;
using NUnit.Framework;
namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class ParallelTestSample03
    {
        [TestFixtureSetUp]
        public void TestFixtureSetUp()
        {     // ...
        }

        [TestFixtureTearDown]
        public void TestFixtureTearDown()
        {     // ...
        }

        [SetUp]
        public void TestSetUp()
        {     // ...
        }

        [TearDown]
        public void TestTearDown()
```

```
        {    // ...
        }

        [ParallelTest(4, "Test")]
        public void Test01()
        {    // ...
        }
    }
}
```

Obrázek 5.13: Testovací třída sloužící pro demonstraci životního cyklu třídy ve spojení s paralelními testy.

Na testovací třídě *ParallelTestSample03* (obr. 5.13) budou prezentovány jednotlivé fáze spouštění paralelních testů. Třída obsahuje všechny podporované inicializační a úklidové metody a dále jeden paralelní test *Test*. Spuštění testů testovací třídy *ParallelTestSample03* bude probíhat v následujících krocích.

1. Vytvoření testovací třídy *ParallelTestSample03*.
2. Provedení inicializace na úrovni třídy - spuštění metody *TestFixtureSetUp*.
3. V této fázi je testovací třída připravená a začíná spouštění jednotlivých testů.
4. Začátek testu *Test*.
5. Provedení inicializace na úrovni testu - spuštění metody *TestSetUp*.
6. Vytvoření požadovaného počtu vláken pro všechny jednotlivé testovací metody, které mají být v rámci testu spuštěny paralelně. V tomto případě budou vytvořeny čtyři vlákna pro testovací metodu *Test01*.
7. Spuštění vláken s jednotlivými testy.
8. Čekání na dokončení všech jednotlivých testů.
9. Provedení úklidu na úrovní testu - spuštěním metody *TestTearDown*.
10. Konec testu *Test*.
11. V této fázi skončilo testování dostupných testů.
12. Provedení úklidu na úrovni třídy - spuštění metody *TestFixtureTearDown*.
13. Uvolnění testovací třídy *ParallelTestSample03*.

5.3.5 Atribut ParallelTest a parametrizovatelné testy

Při spuštění testovacích metod v rámci paralelního testu je velice často nutné identifikovat jednotlivé instance běžících metod mezi sebou. Důvodem tohoto požadavku může být například potřeba rozlišení dat, se kterými má konkrétní instance pracovat. V současné době patrně neexistuje nástroj, který by výše zmíněnou funkcionalitu nabízel vlastními prostředky. FastUnit požadovanou funkcionalitu podporuje a dále rozšiřuje.

FastUnit nabízí možnost předání požadovaných dat do jednotlivých metod při jejich spuštění v rámci paralelního testu. Pomocí tohoto mechanismu lze například docílit výše zmíněné identifikace jednotlivých instancí spuštěných metod. Této funkcionality je dosaženo použitím atributu *Data* při definici paralelního testu.

Atribut *Data* se chová ve spojení s atributem *ParallelTest* obdobně jako při spojení s atributem *Test*, které bylo popsáno a odstavci 5.2.3. Atribut *Data* v tomto případě slouží pro předání parametrů do instance testovací metody v čase jejího spuštění. Jedna testovací metoda může být označena více atributy *Data*, každý atribut pak odpovídá jedné instanci dané metody, která bude spuštěna v rámci paralelního testu s požadovanými parametry.

V případě použití atributu *Data* v rámci definice paralelního testu, je parametr *threadCount* atributu *ParallelTest* ignorován. Počet vláken, ve kterých bude testovaná metoda spuštěna, je určen počtem výskytu atributu *Data*.

```
using System;
using NUnit.Framework;
namespace fastUnit.Samples.Tests
{
    [TestFixture]
    public class ParallelTestSample04
    {
        [ParallelTest(0, "Test")]
        [Data(10)]
        [Data(20)]
        [Data(30)]
        public void Test01(int count)
        {    // ...
        }
    }
}
```

Obrázek 5.14: Ukázka definice paralelního testu ve spojení s atributem *Data*.

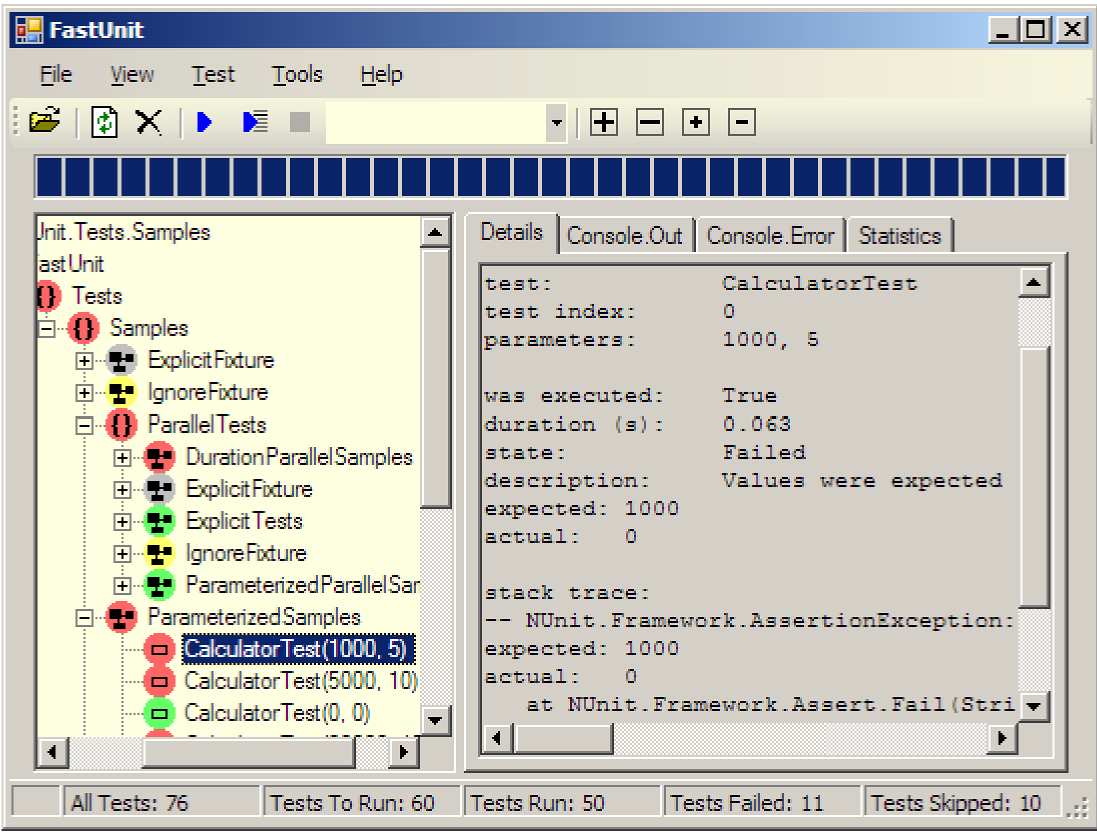
Testovací třída *ParallelTestSample04* zobrazená na obrázku 5.14 obsahuje jeden paralelní test *Test*. Při jeho spuštění bude testovací metoda *Test01* spuštěna paralelně ve třech vláknech. V prvním vlákně bude testovací metodě předána hodnota 10, v druhém vlákně hodnota 20 a ve třetím vlákně hodnota 30.

6 Uživatelská dokumentace

Nástroj FastUnit nabízí dva způsoby práce s testy. Prvním způsobem je použití aplikace s grafickým uživatelským prostředím (*fastUnit.GUI.exe*), která umožňuje interaktivní práci s testy a nabízí grafickou prezentaci výsledků testů. Druhým způsobem je využití konsolové aplikace (*fastUnit.Cmd.exe*), která je rychlejší a umožňuje výsledek uložit ve formě XML dokumentu. Druhá metoda však není interaktivní.

6.1 Aplikace s grafický uživatelským prostředím

Aplikace *fastUnit.GUI.exe* nabízí uživateli plně interaktivní práci s testy v příjemném grafickém prostředí. Dostupné testy jsou zobrazeny způsobem, který připomíná zobrazení souborů a adresářů v souborových průzkumnících. Zobrazení testů nabízí vizuální indikaci úspěchu nebo neúspěchu testů, jednotlivé stavy testů jsou barevně rozlišeny. Prostředí dále dovoluje spouštět jednotlivé testy nebo skupiny vybraných testů samostatně.



Obrázek 6.1: Ukázka aplikace *fastUnit.GUI.exe* zobrazující výsledky právě provedených testů.

6.1.1 Menu a nástrojová lišta

V horní části aplikace se nachází menu, které obsahuje všechny funkce programu. Nástrojová lišta ležící pod menu umožňuje rychlý přístup k téměř všem položkám v menu. Každá položka menu i nástrojové lišty obsahuje krátký popisek prováděné operace, který se zobrazuje po najetí myší nad položku.

Seznam všech funkcí programu:

Ikona	Název	Popis
	File / Open	Otevře dialog pro nalezení testovacího modulu. Po vybrání a otevření testovacího modulu bude jeho obsah načten a zobrazen.
	File / Reload	Znovu otevře načtený testovací modul.
	File / Close	Uvolní načtený testovací modul.
	File / Exit	Ukončí běh celé aplikace.
	View / Expand All	Rozbalí kompletní strom testů až na úroveň jednotlivých testů.
	View / Collapse All	Zabalí strom testů až do úrovně modulu.
	View / Expand To Fixtures	Rozbalí strom testů na úrovně testovacích tříd. Pokud jsou některé testovací třídy již rozbaleny, jsou ponechány v rozbaleném stavu.
	View / Collapse Fixtures	Zabalí strom testů do úrovně testovacích tříd. Pokud je již část stromu zabalena na vyšší úrovni než je úroveň testovacích tříd, je daná část ponechána v zabaleném stavu.
	Test / Run All	Spustí všechny testy. V případě že je vybrána kategorie testů, spustí pouze testy zařazené do vybrané kategorie.
	Test / Run Selected	Spustí pouze testy vybrané ve stromu testů.
	Test / Stop	Zastaví právě běžící testy.
	Tools / Label Output	Je-li funkce aktivní je ve výstupu generovaného testy označen začátek a konec jednotlivých testů.
	Category	Je-li vybrána neprázdná kategorie jsou při spuštění všech testů spuštěny pouze ty testy, které jsou zařazeny do vybrané kategorie.

6.1.2 Průzkumník testů

V levé části aplikace se nachází průzkumník testů. Jedná se o komponentu, ve které je zobrazen obsah načteného testovacího modulu. Dostupné testy jsou zatříděny ve stromě, který hierarchicky reprezentuje obsah testovaného modulu. Kořenový uzel představuje vlastní testovací modul. Pod kořenovým uzlem jsou zobrazeny uzly reprezentující jmenné prostory použité v testovacím modulu. Dále jsou podle svého umístění zobrazeny dostupné testovací třídy. Listy stromu tvoří jednotlivé testy.

Průzkumník testů rozlišuje význam jednotlivých uzlů stromu pomocí ikon.

Ikona	Popis
	Testovací modul.
	Jmenný prostor použitý v testovacím modulu.
	Testovací třída.
	Test
	Paralelní test.

Průzkumník testů slouží taktéž k indikaci stavu jednotlivých testů. Stav jednotlivého uzlu stromu je zobrazen barevným podbarvením ikony uzlu. Následuje seznam a význam jednotlivých barev.

Ikona	Barva	Popis
	Šedá	Test nebyl spuštěn. Po načtení testovacího modulu jsou všechny testy ve stavu „nespuštěn“.
	Modrá	Test je právě spuštěn nebo-li test právě probíhá.
	Zelená	Test skončil úspěšně.
	Žlutá	Test byl při spuštění ignorován.
	Červená	Test skončil s chybou nebo definice testu je nevalidní.

6.1.3 Výsledkový panel

V pravé části aplikace se nachází panel, který zobrazuje detailnější informace o testech a jejich výsledcích. Panel se skládá ze čtyř záložek, každá záložka poskytuje jiný druh informace.

Záložka Details

Záložka *Details* zobrazuje podrobné informace o uzlu, který je aktuálně vybrán v průzkumníku testů. Formát výpisu je textový a obsahuje pouze relevantní informace pro vybraný uzel s ohledem na jeho typ a stav. Informace, které nejsou relevantní pro daný uzel nebo jsou nevyplněné, nejsou v přehledu zobrazeny.

Název	Popis
Id	Interní identifikační číslo testu.
Assembly	Jméno testovacího modulu
Fixture	Jméno testovací třídy.
Test Group	Název logické skupiny testů. Záznam má význam pouze pro paralelní testy.
Test	Název testu.
Test Index	Index testu v rámci logické skupiny. Záznam má význam pouze pro paralelní testy.
Description	Uživatelem zadaný slovní popis testu.
Categories	Čárkou oddělený seznam kategorií do kterých je test zařazen.
Explicit	Příznak signalizující zda je test označen atributem Explicit.
Explicit Reason	Uživatelem zadaný slovní popis důvodu proč je test označen atributem Explicit.
Was Executed	Příznak signalizující zda byl test spuštěn. Záznam může nabývat následujících hodnot: True = Test byl spuštěn. False = Test nebyl spuštěn.
Duration (s)	Délka trvání testu v sekundách.
State	Výsledek testu. Záznam může nabývat následujících hodnot: Passed = Test skončil s úspěšně. Failed = Test skončil s chybou. Ignored = Test byl ignorován.
State Description	Textový popis stavu. Např. obsahuje text chyby vyskytlé během testu.
Stack Trace	Detail vyskytlé chyby včetně výpisu zásobníku.

Záložka Console.Out

Záložka Console.Out slouží pro zobrazení výstupu generovaného prováděnými testy.

Záložka Console.Error

Záložka Console.Error slouží pro zobrazení chybového výstupu generovaného prováděnými testy.

Záložka Statistics

Záložka Statistics obsahuje v souhrnné formě informace o dokončených testech. U dokončených testů jsou v sloupečcích zobrazeny následující informace:

Název	**Popis**
Fixture	Název testovací třídy.
Group	Název logické skupiny testů. Sloupec má význam pouze pro paralelní testy.
Test	Název testu.
Index	Index testu v rámci logické skupiny. Sloupec má význam pouze pro paralelní testy.
State	Výsledek testu. Sloupec může nabývat následujících hodnot: *Passed* = Test skončil úspěšně. *Failed* = Test skončil s chybou. *Ignored* = Test byl ignorován.
Execute	Příznak signalizující zda byl test spuštěn. Sloupec může nabývat následujících hodnot: *True* = Test byl spuštěn. *False* = Test nebyl spuštěn.
Duration (s)	Délka trvání testu v sekundách.

6.1.4 Stavový řádek

V dolní části aplikace se nachází stavový řádek, který zobrazuje aktuální informace o stavu aplikace. Každá položka stavového řádku obsahuje krátký popisek zobrazované informace, který je aktivován po najetí myší nad položku.

Význam jednotlivých položek stavového řádku:

Název	**Popis**
	Informace o aktuálně prováděné operaci.
All Tests	Počet všech dostupných testů.
Tests To Run	Počet vybraných testů pro spuštění.
Tests Run	Počet již spuštěných testů.
Tests Failed	Počet testů, které skončily chybou.
Tests Skipped	Počet testů, které byly ignorovány.

6.1.5 Parametry spuštění

Aplikace *fastUnit.GUI.exe* podporuje řadu parametrů, které lze zadat při spuštění a které výrazně usnadňují a urychlují práci s aplikací.

Seznam podporovaných parametrů:

Parametr	Popis
-asm assembly	Jméno testovacího modulu, který je načten po spuštění aplikace.
-autoRun	Automaticky spustí načtený testovací modul po spuštění aplikace.
-expandAll	Rozbalí celý strom testů.
-expandToFixtures	Rozbalí strom testů na úroveň testovacích tříd.
-label	Ve výstupu generovaného testy se vyznačí začátek a konec jednotlivých testů.

6.2 Konsolová aplikace

Konsolová aplikace je textová aplikace, která může být použita při testování modulů, není-li zapotřebí grafické prezentace výsledků testů. Aplikace je vhodná pro zautomatizování testů a integraci s jinými systémy používanými během procesu vývoje softwaru. Aplikace podporuje uložení výsledku proběhlých testů ve formátu XML, což umožňuje výsledek dále zpracovávat.

```
C:\WINXP\system32\cmd.exe
o exception was received.

**** Fixture "fastUnit.Samples.Tests.TestFixture05" passed.
**** Fixture "fastUnit.Samples.Tests.TestFixture06" started.
***** Test "fastUnit.Samples.Tests.TestFixture06.Test()" started.
***** Test "fastUnit.Samples.Tests.TestFixture06.Test()" passed.
***** Test "fastUnit.Samples.Tests.TestFixture06.Test()" started.
***** Test "fastUnit.Samples.Tests.TestFixture06.Test()" passed.
***** Test "fastUnit.Samples.Tests.TestFixture06.Test()" started.
***** Test "fastUnit.Samples.Tests.TestFixture06.Test()" passed.
**** Fixture "fastUnit.Samples.Tests.TestFixture06" passed.
**** Fixture "fastUnit.Samples.Tests.TestFixture07" started.
***** Test "fastUnit.Samples.Tests.TestFixture07.Test()" started.
***** Test "fastUnit.Samples.Tests.TestFixture07.Test()" passed.
***** Test "fastUnit.Samples.Tests.TestFixture07.Test()" started.
***** Test "fastUnit.Samples.Tests.TestFixture07.Test()" passed.
**** Fixture "fastUnit.Samples.Tests.TestFixture07" passed.
*** Assembly finished: assembly "fastUnit.Samples.Tests".

D:\Work\TTD\fastUnit\bin\Debug>_
```

Obrázek 6.2: Ukázka výstupu programu fastUnit.Cmd.exe .

Příklad na obrázku 6.2 ukazuje výstup generovaný programem *fastUnit.Cmd.exe* při spuštění testů. Příklad byl prezentován na testech testovacího modulu *fastUnit.Samples.Tests.dll*, který je součástí distribuce nástroje FastUnit.

6.2.1 Parametry

Konsolová aplikace podporuje sadu parametrů, které slouží pro nastavení běhu aplikace. Seznam dostupných parametrů lze získat příkazem *fastUnit.Cmd.exe –help*.

```
C:\WINXP\system32\cmd.exe
D:\Work\TTD\fastUnit\bin\Debug>fastUnit.Cmd.exe -help
Runs a set of unit tests from the console.

fastUnit.Cmd.exe [-asm assembly] [-xml filename]
                 [-output filename] [-error filename]
                 [-label]
                 [-fixture fixture fullname]
                 [-cat category1[,category2[...]]]

Options:
   -asm       Test assembly.
   -xml       Name of XML output file.
   -output    Name of file to receive test output.
   -error     Name of file to receive test error output.
   -label     Label each test in test output.
   -fixture   Fixture to test.
   -cat       Comma separated list of categories to test.
   -help      Display help only.

D:\Work\TTD\fastUnit\bin\Debug>
```

Obrázek 6.3: Výpis nápovědy programu *fastUnit.Cmd.exe*.

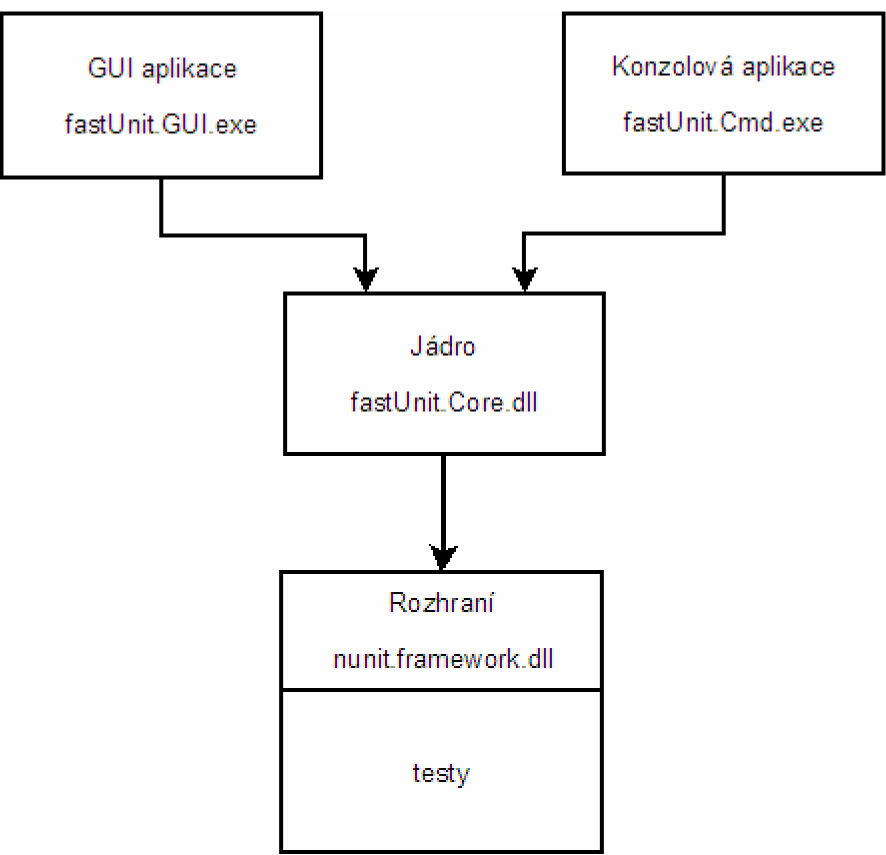
Obrázek 6.3 zobrazuje seznam všech podporovaných parametrů. Význam jednotlivých parametrů je následující.

Parametr	Popis parametru
-asm	Jméno testovacího modulu. Spustí testy nacházející se v zadaném modulu.
-xml	Jméno souboru kam má být uložen výsledek testů ve formátu XML.
-output	Jméno souboru kam má být přesměrován výstup generovaný testy. Není-li parametr zadán je výstup testů přesměrován na standardní výstup aplikace *fastUnit.Cmd.exe*.
-error	Jméno souboru kam má být přesměrován chybový výstup generovaný testy. Není-li parametr zadán je chybový výstup testů přesměrován na standardní chybový výstup aplikace *fastUnit.Cmd.exe*.
-label	Ve výstupu testů vyznačí začátek a konec jednotlivých testů.
-fixture	Jméno testovací třídy. Je-li parametr zadán, jsou spuštěny pouze ty testy, které se nacházejí v zadané testovací třídě. Není-li parametr zadán jsou spuštěny všechny testy nacházející se v testovacím modulu.
-cat	Čárkou oddělený seznam kategorií testů. Pouze testy náležící do zadaných kategorií jsou spuštěny. Není-li parametr zadán, jsou spuštěny všechny testy nacházející se v testovacím modulu.
-help	Zobrazí nápovědu.

7 Programátorská dokumentace

Nástroj FastUnit je napsán pro platformu *Microsoft .NET* v programovacím jazyce *C#*. Nástroj FastUnit si klade za cíl se stát obstojnou alternativou k dnes velice rozšířenému a používanému nástroji NUnit. Volba platformy i programovacího jazyka byla teda rozhodnuta nutností zachovat zpětnou kompatibilitu s nástrojem NUnit.

Architektura nástroje FastUnit je modulární a je rozdělena do čtyř hlavních částí. Každá část plní svou předepsanou úlohu a přes podporované rozhraní komunikuje s ostatními částmi systému.



Obrázek 8.1: Architektura nástroje FastUnit.

Obrázek 8.1 reprezentuje celkovou architekturu nástroje FastUnit, zobrazuje jednotlivé části a jejich vzájemné propojení. Ve středu architektury se nachází *jádro* celého systému - knihovna *fastUnit.Core.dll*. Knihovna zajišťuje kompletní funkcionalitu týkající se práce s testovacími moduly a vlastního testování. Na vrcholu celé architektury jsou umístěny dvě aplikace nabízené nástrojem FastUnit: *GUI aplikace* a *konzolová aplikace*. Obě aplikace používají pro práci s testy služeb *jádra*, na které delegují své požadavky. Každá z aplikací nabízí vlastní způsob práce s testy a prezentaci jejích výsledků. Ve spodní části architektury se nachází knihovna *nunit.framework.dll*, která definuje veřejné *rozhraní* sloužící pro definici testů. Nejzajímavější částí celé architektury je *jádro*, které bude popsáno v následujícím odstavci.

7.1 Jádro

Základním stavebním kamenem nástroje FastUnit je jeho *jádro*, které je umístěno v knihovně *fastUnit.Core.dll*. *Jádro* implementuje veškerou funkcionalitu spojenou s načítáním a správou testovacích modulů a s prováděním a vyhodnocováním testů. Implementována funkcionalita je pak využívaná jednotlivými aplikacemi nástroje FastUnit. Nabízenou funkcionalitu lze rozdělit do dvou skupin podle svého zaměření.

Operace sloužící pro práci a správu testovacích modulů:

- Načtení testovacího modulu.
- Vyhledání testovacích tříd a jednotlivých testů obsažených v testovacím modulu.
- Poskytování seznamu dostupných testů.
- Poskytování statistických údajů o načteném testovacím modulu.

Operace sloužící pro testování:

- Kontrola běhu testů: spouštěni testů, zastavování či přerušování běhu testů a ošetřování chyb vzniklých během testů.
- Vyhodnocování testů.
- Poskytování výsledků proběhlých testů.
- Průběžné informování o stavu probíhajících testů.

7.1.1 Základní vlastnosti jádra

Aplikační domény

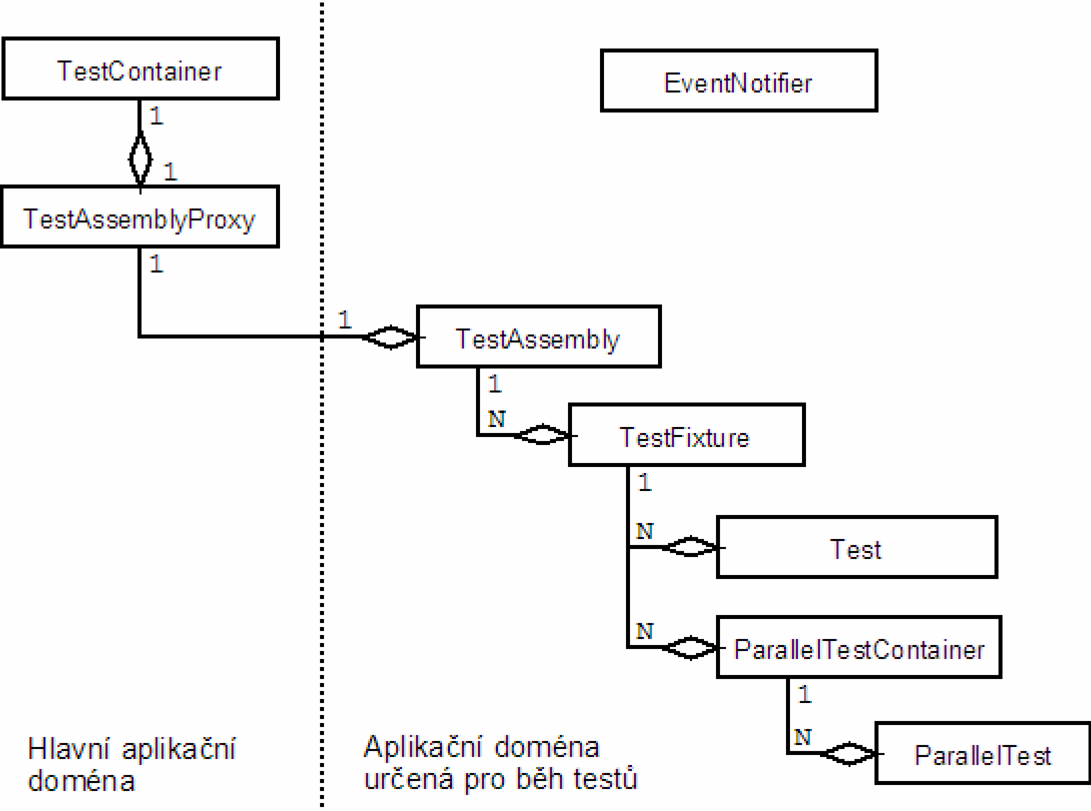
V rámci aplikace je načtený testovací modul umístěn v samostatné aplikační doméně, která je nezávislá na hlavní aplikační doméně celé aplikace. Důvodem umístění modulu do oddělené aplikační domény, je snaha minimalizovat vzájemné ovlivňování testovací aplikace a běžících testů. Při načítání testovacího modulu je vždy vytvořena nová aplikační doména, aplikační domény vytvořené pro předchozí testovací moduly se znovu nepoužívají. Při uzavírání testovacího modulu je odstraněna celá jeho aplikační doména, tím je zajištěno korektní uvolnění všech zdrojů vytvořených testovacím modulem.

Asynchronní provádění testů

Testovaní načteného testovacího modulu je v jádře implantováno jako neblokující (asynchronní) operace. Při zpracovávání požadavku na spuštění testů je vytvořeno nové pracovní vlákno, ve kterém je prováděno vlastní testování. Po vytvoření pracovního vlákna a předání požadavku je řízení okamžitě vráceno zpět vyšší vrstvě.

Provádění testů je poměrně časově náročná operace, v závislosti na počtu jednotlivých testů a jejich náročnosti může trvat i řádově desítky minut. Důvod asynchronního provádění testů je snaha neblokovat pracovní vlákno vyšší vrstvy po dobu běhu testů. Například v případě *GUI aplikace* by zablokovaným vláknem bylo hlavní vlákno aplikace, které má na starosti interakci s uživatelem.

7.1.2 Rozdělení jádra



Obrázek 8.2: Architektura jádra nástroje FastUnit.

Obrázek 8.2 zobrazuje architekturu *jádra* nástroje FastUnit. Jsou zde zobrazeny hlavní implementační třídy a jejich vzájemné vazby. Obrázek taktéž znázorňuje rozmístění jednotlivých tříd v rámci aplikačních domén. V následujícím textu budou popsány jednotlivé hlavní třídy, bude zmíněn jejich význam a hlavní odpovědnosti. Nejzajímavější vlastností celého *jádra* je jeho asynchronnost a práce s vlákny. Třídy, které tuto činnost zajišťují budou popsány detailněji.

Třída TestContainer

Vstupním bodem *jádra* je třída *TestContainer*. Přes její rozhraní je zpřístupněna veškerá funkcionalita jádra vyšším vrstvám. Třída má tři hlavní úkoly. 1) Správu otevřených testovacích modulů. To obsahuje načítání modulů a jejich organizaci v paměti *jádra*. Dále na straně hlavní aplikační domény udržuje a zpřístupňuje seznamy dostupných testů. 2) Řízení procesu testování. Kontroluje pouze stavy testovacích modulů, požadavky dále deleguje do nižších vrstev. 3) Během testování průběžně sbírá výsledky testů, které jsou po skončení testování zpřístupněny. Třída *TestContainer* taktéž umožňuje se vyšším vrstvám zaregistrovat a dostávat asynchronní notifikace o právě probíhajících testech a jejich výsledcích.

Třída TestAssemblyProxy

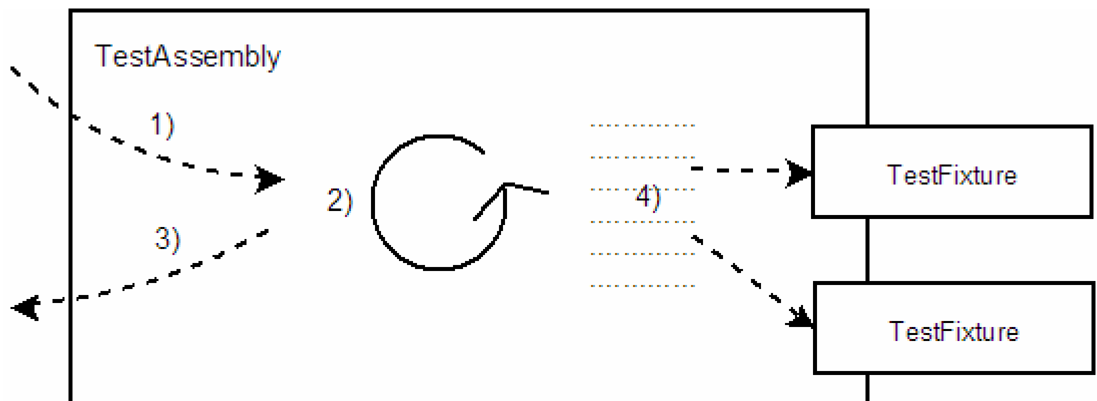
Hlavní funkcí třídy *TestAssemblyProxy* je umožnit komunikaci mezi třídami *TestContainer* a *TestAssembly*, která probíhá přes rozhraní dvou aplikačních domén. Třída *TestAssemblyProxy* implementuje obálku nad třídou *TestAssembly* a nabízí její funkcionalitu v rámci hlavní aplikační domény.

Druhou funkcí třídy *TestAssemblyProxy* je správa životního cyklu aplikační domény určené pro běh testů. Při načítání testovacího modulu je třída zodpovědná za vytvoření nové aplikační domény, vytvoření instance třídy *TestAssembly* a udržování komunikace s vytvořenou instancí. Během zavírání testovacího modulu je zodpovědná za ukončení komunikace s instancí třídy *TestAssembly* a uvolnění celé její aplikační domény z paměti *jádra*.

Třída TestAssembly

Třída *TestAssembly* je vstupním bodem do části *jádra*, které se nachází v aplikační doméně určené pro běh testů. Třída *TestAssembly* reprezentuje jeden testovací modul. V inicializační fázi je zodpovědná za načtení modulu do paměti, prohledání jeho obsahu a vytvoření hierarchie reprezentující nalezené testy.

Během testování je třída zodpovědná za následující operace. 1) Vytvoření nového vlákna pro běh testů a zajištění asynchronní komunikace s tímto vláknem. 2) Přesměrování standardních výstupů podle požadavků vyšších vrstev. 3) Spuštění jednotlivých testovacích tříd. Zmíněné operace jsou pro přehlednost uvedeny na obrázku 8.3.



Obrázek 8.3: Operace třídy *TestAssembly* prováděné během spouštění testů.

Na obrázku 8.3 jsou zobrazeny jednotlivé kroky vykonávané třídou *TestAssembly* během testování. Následuje popis jednotlivých kroků.

1. Přijmutí požadavku na spuštění testů.
2. Vytvoření pracovního vlákna, předání parametrů upřesňující běh testů do kontextu pracovního vlákna a jeho spouštění.
3. Ukončení zpracování požadavku a navracení kontroly vyšší vrstvě.
4. Asynchronní provádění testů v pracovním vlákně. Tento krok je dále rozdělen do následujících fází.
    a. Přesměrování standardního a chybového výstupu dle požadavku vyšší vrstvy.
    b. Sekvenční spouštění testů obsažených v jednotlivých testovacích třídách. Jednotlivé testy jsou již prováděny synchronně a vždy se čeká na jejich dokončení.
    c. Vrácení standardního a chybového výstupu do původního stavu.

Třída TestFixture

Třída *TestFixture* reprezentuje jednu testovací třídu obsaženou v testovacím modulu. Během inicializační fáze prohledává testovací třídu a identifikuje jednotlivé testy a inicializační metody. Během provádění testů je zodpovědná za vytvoření instance

testovací třídy a její inicializaci na úrovni třídy. Za spuštění jednotlivých testů. Po skončení testů je zodpovědná za úklid testovací třídy a její uvolnění.

Třída Test

Třída *Test* reprezentuje jeden synchronní test. Během inicializační fáze prohledává atributy testovací metody a zjišťuje detailní požadavky kladené na daný test. Během fáze provádění testu je zodpovědná za inicializaci testovací třídy na úrovni testu, spuštění testovací metody, provedení úklidu testovací třídy na úrovni testu a vyhodnocení výsledku testu.

Třída ParallelTestContainer

Třída *ParallelTestContainer* reprezentuje jeden paralelní test. Během inicializační fáze prohledává definici testovací metody označené atributem *ParallelTest*. Na jejím základě pak vytváří instance reprezentující jednotlivé běhy dané testovací metody v rámci paralelního testu.

Během fáze testování je zodpovědná za 1) inicializaci testovací třídy na úrovni testu, 2) paralelní spuštění jednotlivých instancí testovacích metod, 3) počkání na dokončení všech spuštěných testovacích metod, 4) provedení úklidu testovací třídy na úrovni testu a 5) vyhodnocení paralelního testu jako celku.

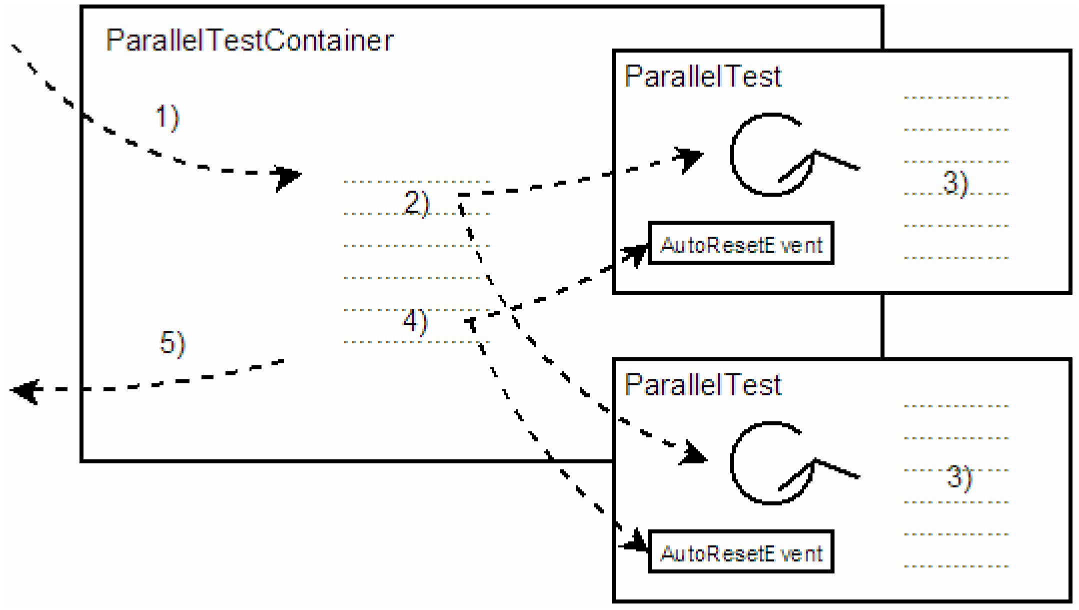
Třída *ParallelTestContainer* je úzce spjata se třídou *ParallelTest*.

Třída ParallelTest

Třída *ParallelTest* reprezentuje jednu instanci testovací metody, která je součástí paralelního testu. Během inicializační fáze neprovádí žádnou vlastní práci, pouze přejímá parametry zjištěné třídou *ParallelTestContainer*.

Během fáze běhu testů je zodpovědná za 1) vytvoření pracovního vlákna, 2) spuštění reprezentované instance testovací metody a vyhodnocení jejího výsledku a 3) ukončení a uvolnění pracovního vlákna.

Operace tříd *ParallelTestContainer* a *ParallelTest* během testování jsou pro přehlednost uvedeny na obrázku 8.4.



Obrázek 8.4: Operace tříd *ParallelTestContainer* a *ParallelTest* během spouštění testů.

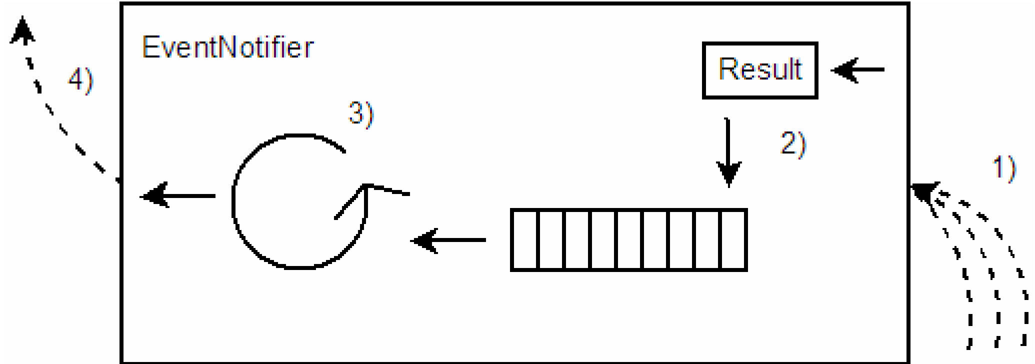
Na obrázku 8.4 jsou zobrazeny jednotlivé kroky vykonávané třídami *ParallelTestContainer* a *ParallelTest* během testování. Následuje popis jednotlivých kroků.

1. Přijmutí požadavku na spuštění testu.
2. Příprava a spuštění vnořených testů. Tento krok je rozdělen do následujících fází:
    a. Inicializace testovací třídy na úrovni testu, tj. spuštění metody označené atributem *SetUp*.
    b. Vytvoření pracovních vláken ve všech vnořených testech.
    c. Spuštění vnořených testů.
3. Paralelní provádění jednotlivých vnořených testů v rámci tříd *ParallelTest*.
4. Ukončení testu a úklid. Tento krok je rozdělen do následujících fází:
    a. Počkání na dokončení všech vnořených testů.
    b. Ukončení a uvolnění všech pracovních vláken.
    c. Provedení úklidu testovací třídy na úrovni testu, tj. spuštění metody označené atributem *TearDown*.
    d. Vyhodnocení paralelního testu jako celku a notifikace výsledku.
5. Ukončení testu a navracení kontroly vyšší vrstvě.

Třída EventNotifier

Třída *EventNotifier* slouží pro přeposílání zpráv o průběhu prováděných testů vyšším vrstvám. Všechny objekty zapojené v procesu testování průběžně informují o svém aktuálním stavu. Třída *EventNotifier* nabízí asynchronní způsob přeposílání notifikací. Důvodem je snaha neblokovat provádění vlastního testování a tím zkreslovat či dokonce ovlivňovat jeho výsledek.

Jednotlivé operace třídy *EventNotifier* během přeposílání zpráv o průběhu testování jsou pro přehlednost uvedeny na obrázku 8.5.



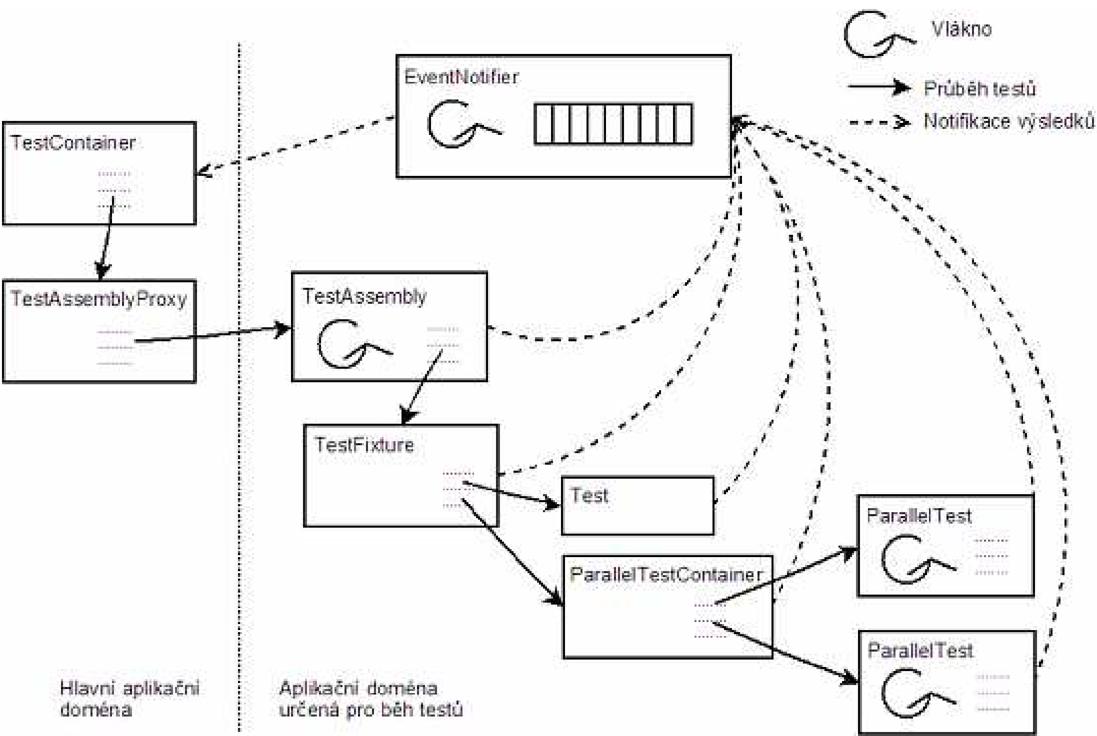
Obrázek 8.5: Operace třídy *EventNotifier* během přeposílání výsledků testů.

Na obrázku 8.5 jsou zobrazeny jednotlivé kroky vykonávané třídou *EventNotifier* během přeposílání zpráv o průběhu testů. Následuje popis jednotlivých kroků.

1. Přijmutí informace o aktuálním stavu testovaného objektu. Např. test byl spuštěn, test skončil úspěšně, test skončil s chybou apod.
2. Zpracování požadavku. Vytvoření specializované instance třídy *Result* reprezentující zprávu a vložení instance do fronty. Je-li nutné je probuzeno pracovní vlákno.
3. Vyzvednutí objektu reprezentujícího zprávu z fronty v pracovním vlákně třídy *EventNotifier*.

4. Oznámení zprávy testovaného objektu v pracovním vlákně třídy *EventNotifier*.

Celý průběh testování je zrekapitulován na obrázku 8.6.



Obrázek 8.6: Celkový pohled na průběh testů s vyznačením jednotlivých pracovních vláken.

Na obrázku 8.6 je zobrazen kompletní průběh testování prováděný *jádrem* nástroje FastUnit. Jsou na něm uvedeny všechny hlavní implementační třídy a jejich vzájemná interakce. Obrázek také znázorňuje rozmístění všech pracovních vláken, která jsou během provádění testů vytvářena.

8 Porovnání s konkurenčními nástroji

Existuje mnoho nástrojů, které umožňují testování softwarových modulů podobným způsobem jako nástroj FastUnit, tj. pomocí testů napsaných uživatelem. Pro porovnání s nástrojem FastUnit byly vybrány tři nástroje, které jsou velice často zmiňovány a doporučovány v souvislosti s technikou Test-Driven Development a které jsou určeny pro vývojovou platformu *Microsoft .NET* a programovací jazyk *C#*. Každý z vybraných nástrojů také reprezentuje specifickou skupinu nástrojů, do které patří.

8.1 NUnit

NUnit [4] je jedním z prvních testovacích nástrojů určený pro vývojovou platformu *Microsoft .NET*. Původně vznikl jako klon nástroje JUnit určeného pro programovací jazyk Java. V současné době se jedná o de facto standard ve světě testovacích nástrojů. Je velice rozšířený a často odkazovaný nástroj. Většina knih a článků zabývajících se problematikou testování softwarových modulů na platformě *Microsoft .NET* používá NUnit jako svůj referenční nástroj. Také většina ostatních nástrojů si ho bere za svou inspiraci a alespoň v počáteční fázi svého vývoje se s ním snaží zachovávat zpětnou kompatibilitu. Nástroj NUnit je poskytován zdarma.

Výhody

- Testy vyvíjené pro nástroj NUnit je možné jednoduše přenést a spouštět i ve většině ostatních nástrojů.
- Velice robustní a spolehlivý nástroj. Obsahuje řadu optimalizací pro práci s testovacími moduly, které obsahují velké množství testů či generují velké množství výstupu. Například je v něm implementována vyrovnávací paměť pro výstup probíhajících testů, před zobrazením uživateli.

Nevýhody

- Nabízí podporu pouze pro tvorbu základních typů testů. Neobsahuje podporu pro vytváření paralelních, či parametrických testů.
- Poměrně dlouhé cykly, ve kterých jsou uvolňovány nové verze nástroje.
- Velice jednoduché uživatelské prostředí, které již nedosahuje úrovně některých novějších nástrojů.

8.2 Zanebug

Zanebug [6] je jedním z posledních přírůstků na poli testovacích nástrojů. Jedná se o velice dynamicky se rozvíjející nástroj, který je vyvíjen jako *open source* software. Samozřejmostí tohoto nástroje je zachování zpětné kompatibility s nástrojem NUnit. Kromě toho nabízí i celou řadu zajímavých funkcionalit, například analýzu výkonu testů, sledování stavu paměti, zobrazování výsledků ve formě grafů apod.

Výhody

- Umožňuje spouštět testy psané pro NUnit bez nutnosti rekompilace.
- Velice propracované grafické prostředí. Umožňuje zobrazovat výsledky testu ve formě grafů.

- V průběhu testu umožňuje sledovat a zaznamenávat řadu systémových ukazatelů, například velikost použité paměti, využití procesoru apod.
- Nabízí detailnější analýzu výkonu testovaného kódu. Umožňuje test spouštět opakovaně a měřit například průměrnou dobu trvání testu.

Nevýhody

- Nenabízí žádnou podporu pro paralelní testy.
- Nepodporuje parametrizovatelné testy.

8.3 TestDriven.NET

TestDriven.NET [7] patří mezi nástroje, které jsou plně integrované do prostředí Microsoft Visual Studio. Jedná se o poměrně nový nástroj, který nedefinuje vlastní rozhraní či mechanismy pro vývoj a definici testů, ale snaží se být kompatibilní s již existujícími nástroji. V současné době podporuje testovací rozhraní definované nástroji NUnit a MbUnit. Jedná se o komerční produkt, který je však pro osobní a studijní potřeby poskytován zdarma.

Výhody

- Plná kompatibilita s nástrojem NUnit.
- Jednoduché rozhraní plně integrované do vývojového prostředí.
- Testy jsou spouštěny v externím procesu. Tím nástroj zabraňuje vzájemnému ovlivnění vývojového prostředí a běžících testů.
- Umožňuje velice jednoduše krokovat spuštěné testy.
- Umožňuje spustit jakoukoli metodu jako test.
- Díky plné integraci do vývojového prostředí je velice užitečný pro testování jednotlivých testů během jejich vývoje. Odpadá zde nutnost spouštět pro jejich testování externí nástroje.

Nevýhody

- Neobsahuje samostatnou aplikaci, je vyvíjen pouze jako modul do vývojového prostředí Microsoft Visual Studio. Tato vlastnost znemožňuje jeho použití pro automatické testování či testování většího množství modulů. Před každým spuštěním testů je totiž nutné nejdříve otevřít příslušný projekt obsahující testovací modul ve vývojovém prostředí Visual Studio a až následně spustit požadované testy.
- Výpis výsledků testů je zobrazován pouze v textové podobě a je promíchán s výstupem generovaným probíhajícími testy. To činí nástroj nepoužitelným pro testování modulů, které obsahují velké množství testů.
- Nelze sledovat průběh právě probíhajících testů. Nástroj nezobrazuje výsledky ani výstup testů průběžně, ale až po skončení všech testů.
- Nepodporuje členění testů do logických kategorií.

9 Závěr

Výsledkem této bakalářské práce je testovací nástroj FastUnit, který může sloužit svým uživatelům při vývoji softwaru. Nástroje FastUnit nabízí řadu pokročilých funkcí, které usnadňují tvorbu složitějších testů. Jedná se zejména o podporu paralelních testů, testování výkonu modulů a podporu parametrizovatelných testů.

V rámci nástroje FastUnit byly vyvinuty dvě aplikace – GUI aplikace a konsolová aplikace. GUI aplikace nabízí interaktivní práci s testy v příjemném grafickém prostředí, přehledně zobrazuje dostupné testy, umožňuje selektivní spouštění testů a vizuálně indikuje výsledek testů. Konzolová aplikace nabízí možnost spouštění testů na automatické bázi a výsledek uložit ve formátu XML. To dělá aplikaci vhodnou pro integraci s jinými systémy používanými během procesu vývoje softwaru.

Práce si také kladla za úkol seznámit čtenáře s poměrně novou technikou vývoje softwaru zvanou Test-Driven Development. Technika Test-Driven Development byla prezentována, jak teoreticky, tak i za pomoci jednoduchého tutoriálu s využitím nástroje FastUnit.

Součástí této práce je přiložené CD, které obsahuje zdrojové i spustitelné soubory nástroje FastUnit, zdrojové soubory vzorových testů a text práce v elektronické podobě.

10 Literatura

[1] Beck K.: *Test Driven Development By Example*, Addison-Wesley Professional, 2002

[2] Ambler S.: *Database Techniques: Effective Strategies for the Agile Software Developer*, Wiley, 2003

[3] Fowler M., Beck K., Brant J., Opdyke W., Roberts D.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional, 1999

[4] NUnit (software)
http://www.nunit.com/

[5] MbUnit (software)
http://www.mertner.com/confluence/display/MbUnit/Home

[6] Zanebug (software)
http://www.adapdev.com/zanebug/

[7] TestDriven.NET (software)
http://www.testdriven.net/

[8] csUnit (software)
http://www.csunit.org/

[9] TestNG (software)
http://testng.org/doc/

[10] JUnit (software)
http://www.junit.org/index.htm

[11] XProgramming (stránky věnované Extremnímu programování)
http://www.xprogramming.com/

[12] Extreme Programming (stránky věnované Extremnímu programování)
http://www.extremeprogramming.org/