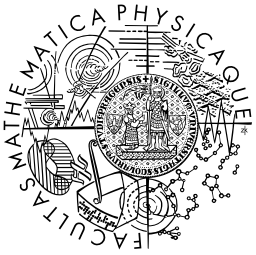


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

**BAKALÁŘSKÁ PRÁCE**

Dominik Hrdý

# PerfEval: Spojení unit testů s vyhodnocováním výkonu

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: prof. Ing. Petr Tůma, Dr.

Studijní program: Informatika

Studijní obor: Systémové programování

Praha 2024

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne .....  .....

Podpis autora

Poděkování.

Název práce: PerfEval: Spojení unit testů s vyhodnocováním výkonu

Autor: Dominik Hrdý

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: prof. Ing. Petr Tůma, Dr., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Abstrakt.

Klíčová slova: testování výkonnost

Title: PerfEval: Marrying unit testing with performance evaluation

Author: Dominik Hrdý

Department: Department of Distributed and Dependable Systems

Supervisor: prof. Ing. Petr Tůma, Dr., Department of Distributed and Dependable Systems

Abstract: Abstract.

Keywords: testing performance

# Obsah

# 1. Úvod

Po více než dvacet let pomáhají unit testy udržovat kvalitu kódu v průběhu vývoje softwaru. Za tuto dobu bylo vyvinuto mnoho knihoven pro implementaci a spouštění unit testů. Verzovací nástroje jako GitLab nebo GitHub umožňují v rámci vývoje software verzovat, spouštět unit testy a reagovat na jejich případná selhání.

Psaní výkonnostních testů, které by pomáhaly udržovat kvalitu kódu stejně jako unit testy, již tak běžné není. Práce s frameworky pro měření výkonu totiž není tak jednoduchá, jako používání knihoven pro psaní unit testů. Průběžné udržování výkonnosti by však mohlo pomáhat udržovat kvalitu kódu obdobným způsobem jako unit testy.

Některé frameworky pro vyhodnocování výkonu softwaru jsou podobné frameworkům pro psaní unit testů. Frameworky jako JMH nebo BenchmarkDotNet pomáhají implementovat výkonnostní testy. Vyhodnocování naměřených výsledků dnes obvykle zahrnuje i jejich ruční zhodnocení. Vyhodnocování výkonu totiž vyžaduje porovnat naměřený výkon s nějakou další referenční hodnotou. Měřící frameworky tyto referenční hodnoty nemají a ani je nikde získat nemohou, protože měří pouze jednu verzi softwaru. Nemohou tedy zmíněné celkové vyhodnocení dělat.

Vyhodnocování výkonnostních testů je složitější problém než vyhodnocování unit testů. Unit testy testují korektnost. Ta se prokáže tak, že na každý zadaný vstup program vrátí očekávaný výstup. Při vyhodnocování výkonu se změří sada dat o běhu programu. Tato data ale sama o sobě nemají požadovanou vypovídající hodnotu a je nutné je zkoumat v kontextu.

Výsledkem této práce je nástroj nazvaný PerfEval. PerfEval je konzolová aplikace napsaná v programovacím jazyce Java. PerfEval umí vyhodnocovat výsledky výkonnostních testů. Způsob a průběh vyhodnocování je řízený pomocí argumentů příkazové řádky.

PerfEval je schopen automatického hlášení výsledků výkonnostních testů. Umí porovnávat výsledky měření výkonu dvou verzí softwaru mezi sebou. PerfEval je také vhodný pro skriptování, protože o výsledcích informuje nejen výpisem na standardní výstup, ale také exit kódem. Nástroj podporuje zpracování výstupů frameworků BenchmarkDotNet a JMH ve formátu JSON, ale je rozšiřitelný i pro zpracování jiných frameworků nebo formátů.

# 2. Kontext

## 2.1 Testování softwaru při vývoji

Při vývoji softwaru je vhodné vyvíjený software neustále testovat. Software lze testovat více způsoby, ale cílem testů je vždy otestovat některý z důležitých kvalitativních atributů, jako je například korektnost nebo výkon. Testování výkonu není dnes při vývoji softwaru příliš obvyklé. (Stefan a kol., 2017)

Pro testování korektnosti se obvykle používají unit testy. Jedná se většinou o krátké testovací funkce, které kontrolují, jestli se testovaný kód chová požadovaným způsobem. Zjišťují tedy, jestli kód na zadaný vstup vrátí očekávaný výstup. Ke kódu, který není aktuálně vyvíjen, se obvykle nemění ani unit testy. Když se mění části kódu kolem již takto hotové části, tak unit testy stále průběžně sledují korektnost tohoto již hotového kódu. Zdali je kód korektní se pomocí unit testů zjistí velmi jednoduše. Kód je korektní, pokud vrátil očekávaný výstup a pokrytí kódu unit testy je dostatečné.

## 2.2 Průběžná integrace

Při vývoji softwaru se často používají nástroje pro automatizování některých činností. Obvykle se automatizují činnosti jako jsou správa verzí, spouštění testů a jejich vyhodnocování.

Pro správu verzí při vývoji softwaru se obvykle používá nástroj Git. Git je vhodný i pro práci ve velkých týmech. Umožňuje totiž členění projektu do větví. Každá větev je vhodná pro vývoj samostatné části aplikace. Následným spojováním větví pak dochází k propojení větších funkčních celků aplikace, které byly vyvinuty v jednotlivých větvích. Nástroj si formou pamatování si změn udržuje přehled o průběžných verzích a provedených změnách. Nástroj je možné ovládat jednoduchými příkazy z příkazové řádky. Je to tedy nástroj, který je možné ovládat ze skriptů.

Nástroj Git umí jednoduše zprostředkovat průběžnou integraci (continuous integration). Průběžná integrace umožňuje uživateli dělat automatizované kroky při nahrávání nových verzí do větve nebo při spojování větví. Při průběžné integraci tedy jde o spouštění skriptu při některé ze zmíněných událostí. V rámci průběžné integrace je možné testovat software pomocí unit testů i benchmarků pro měření výkonu. Dále je možné použít jakékoli jiné příkazy příkazové řádky. Do průběžné integrace je tedy možné jednoduše zapojit téměř jakoukoli konzolovou aplikaci. Při selhání skriptu, když skončí s nenulovým exit kódem, je možné změny neintegrovat. Tímto způsobem je možné zamezit integraci chybného kódu.

## 2.3 Měření výkonu

Testování výkonu obvykle probíhá tak, že se použije nějaký vhodný měřící framework. Obvyklé frameworky pro měření výkonu mají vlastní pravidla, jak se mají označit metody, které se mají měřit. Tyto frameworky umožňují měřit výkon

softwaru podle různých metrik. Mezi tyto metriky se řadí čas, propustnost a například spotřeba paměti. Výsledky měření umí frameworky obvykle zaznamenat jak do strojově čitelného formátu, tak do formátu čitelného pro člověka.

V dalších podkapitolách jsou uvedeny příklady měřících frameworků pro jazyky C# a Java. Z uvedených příkladů bude patrné, že tyto frameworky samy o sobě nemohou poskytnout informace o změně výkonu. Neuchovávají totiž informace o měření jiné verze softwaru, než té měřené. Jak již bylo v úvodu naznačeno, výsledky měření je nutné porovnat s výsledky měření jiné verze softwaru. Zmíněné nástroje toto neumí, ale pro samotné měření výkonu jsou velmi vhodné.

### 2.3.1 BenchmarkDotNet

BenchmarkDotNet je framework určený k měření výkonu programů na platformě .NET. V důsledku toho, že měří programy na platformě .NET, je schopen měřit výkon programů napsaných v programovacích jazycích C#, F# a Visual Basic. Podrobnosti o tomto měřícím frameworku je možné nalézt v dokumentaci (Akinshin, 2024).

Při měření výkonu pomocí BenchmarkDotNet se měřené hodnoty vypisují na standardní výstup včetně konečného shrnutí. Mimo standardní výstup se ještě výsledky měření ukládají do strojově zpracovatelných formátů, jako je například JSON nebo CSV.

Výstupem měření výkonu programu v jazyce C# pomocí frameworku BenchmarkDotNet jsou již statisticky zpracované hodnoty. Aby bylo možné sledovat jednotlivé naměřené hodnoty, je nutné zvolit již při psaní testů správný exportér, který tuto funkci podporuje.


```
},{
   "IterationMode":"Overhead",
   "IterationStage":"Warmup",
   "LaunchIndex":1,
   "IterationIndex":1,
   "Operations":8192,
   "Nanoseconds":32500
},{
   "IterationMode":"Overhead",
   "IterationStage":"Actual",
   "LaunchIndex":1,
   "IterationIndex":15,
   "Operations":8192,
   "Nanoseconds":31300
},{
   "IterationMode":"Workload",
   "IterationStage":"Warmup",
   "LaunchIndex":1,
   "IterationIndex":1,
   "Operations":8192,
   "Nanoseconds":613752700
},{
   "IterationMode":"Workload",
   "IterationStage":"Actual",
   "LaunchIndex":1,
   "IterationIndex":5,
   "Operations":8192,
   "Nanoseconds":641681400
},{
```


Obrázek 2.1: Struktura výsledků měření frameworku BenchmarkDotNET

BenchmarkDotNet je možné nakonfigurovat tak, aby se v souboru s výsledky nacházely podrobnosti o prostředí, jako je operační systém, verze platformy .NET, jméno, typ a parametry procesoru. Dále aby se v souboru s výsledky nacházely výsledky jednotlivých provedených měření. Výsledek měření u sebe má informaci

o jméně testovací metody, zpracované statistické údaje a naměřené hodnoty z různých módů a iterací měření. Konkrétní příklad toho, jak vypadají módy a iterace měření se nachází na obrázku 2.1. Názvy módů a iterací mají intuitivní názvy, takže je z nich poznat, kdy se ještě probíhá překlad, a kdy už se měří plně přeložený kód. C# je totiž jazyk kompilovaný metodou JIT, a proto kód není přeložen se všemi optimalizacemi hned na začátku, tudíž je nutné měřit více iterací, než se kód plně přeloží.

### 2.3.2 JMH (Java Microbenchmark Harness)

JMH je framework pro měření výkonu, který umožňuje pomocí anotací definovat výkonnostní testy pro programy v jazyce Java. Z průzkumu (Stefan a kol., 2017) vyplývá, že se jedná o nejpoužívanější framework pro měření výkonu pro projekty vyvíjené v jazyce Java.

JMH obdobně jako BenchmarkDotNet poskytuje výsledek měření jako tabulku na standardní výstup. Dále poskytuje výstup v podobě strojově zpracovatelných formátů, jako jsou například XML nebo JSON. O výstup v této podobě je nutné zažádat pomocí argumentů na příkazové řádce při spouštění měření. Další podrobnosti o frameworku JMH je možné nalézt v dokumentaci (Shipilëv, 2024).


```
{
    "jmhVersion" : "1.37",
    "benchmark" : "io.crate.types.StringTypeTest.booleanConversion",
    "mode" : "thrpt",
    "threads" : 1,
    "forks" : 5,
    "jvm" : "/usr/lib/jvm/java-21-openjdk-amd64/bin/java",
    "jvmArgs" : [
    ],
    "jdkVersion" : "21.0.1",
    "vmName" : "OpenJDK 64-Bit Server VM",
    "vmVersion" : "21.0.1+12-Ubuntu-222.04",
    "warmupIterations" : 5,
    "warmupTime" : "10 s",
    "warmupBatchSize" : 1,
    "measurementIterations" : 5,
    "measurementTime" : "10 s",
    "measurementBatchSize" : 1,
    "primaryMetric" : {
        "score" : 1.828893179607 3906E9,
        "scoreError" : 1.01653921742499 65E8,
        "scoreConfidence" : [
            1.727239257864891E9,
            1.9305471013498902E9
        ],
        "scorePercentiles" : {
            "0.0" : 1.43012788809343 58E9,
            "50.0" : 1.8645360710333076E9,
            "90.0" : 1.9365128374326475E9,
            "95.0" : 1.9403031190979056E9,
            "99.0" : 1.94167079732560 59E9,
            "99.9" : 1.94167079732560 59E9,
            "99.99" : 1.9416707973256059E9,
            "99.999" : 1.941670797325605 9E9,
            "99.9999" : 1.9416707973256059E9,
            "100.0" : 1.9416707973256059E9
        },
        "scoreUnit" : "ops/s",
        "rawData" : [
            [
                1.8554191503325295E9,
                1.8561380112017648E9,
                1.8560222540239005E9,
                1.8499319451317933E9,
                1.85752715931031 5E9
            ],
```


Obrázek 2.2: Struktura výsledků měření frameworku JMH

Ve výstupním souboru měření pomocí JMH lze nalézt informace o stroji na kterém probíhalo měření. Jedná se především o název stroje a verzi operačního systému. Dále zde lze nalézt verzi Javy, ve které probíhalo měření. V souboru je možné vidět také ostatní parametry měření, jako je zahřívací doba a počet zahřívacích iterací. Zahřívací iterace jsou zde uvedeny, protože Java je stejně jako C# jazyk kompilovaný metodou JIT, a tak je také nutné měřit více iterací, než se kód plně přeloží se všemi optimalizacemi.

Jednotlivé naměřené hodnoty jsou ve výstupním souboru dostupné i ve výchozím nastavení JMH. Naměřené hodnoty nejsou bezrozměrná čísla, ale jsou doplněny i o fyzikální jednotku, kterou naměřená hodnota reprezentuje. Pro každý běh (fork), přičemž běhů může být v jednom spuštění JMH více, se objeví jedna sada měřených hodnot u položky `rawData`.

## 2.4 Scénář

Mějme programátora, který vytváří softwarové dílo. Mezi dobré programovací zásady patří udržovat kvalitu kódu hned od začátku vývoje. Programátor tedy podle dobrých zásad začne používat verzovací nástroj. Tento samotný nástroj pro udržování kvality kódu nestačí. Umožní ale programátorovi použít průběžnou integraci.

Programátor dále podle zvyku začne psát unit testy pro vytvářený kód. Unit testy mu pomohou udržovat korektnost kódu. Korektnost je jedním z aspektů kvality kódu. Aby mu unit testy pomohly udržovat kvalitu kódu, musí dobře pokrývat kód. Výhodou unit testovacích frameworků je jejich snadné zapojení do průběžné integrace. Programátor je tedy začne tímto způsobem spouštět s každou novou verzí.

Dalším z aspektů kvality kódu, které by chtěl udržovat je výkon. Programátor s využitím nějakého frameworku pro měření výkonu začne měřit výkon svého kódu. Měření výkonu se trochu podobá unit testům. U měřících frameworků jako je BenchmarkDotNET nebo JMH se vytvoří specializované měřící metody, které se podobají unit testovacím metodám z unit testovacích frameworků jako je například JUnit. Tyto měřící metody, ale změří pouze číslo a jednotku, která nevypovídá o změně výkonu oproti jiné verzi. Pokud chce programátor výkon udržovat potřebuje vyhodnotit právě tuto změnu.

Programátor tedy musí vzít výsledky měření aktuální verze a výsledky měření referenční verze vůči které chce změnu výkonu vyhodnocovat a sám výsledky porovnat. Porovnání výsledků měření nemusí být možné pouhým pohledem na sady číselných dat. Pravděpodobně si budou hodnoty hodnoty obou sad podobné. Obzvlášť pokud změna bude příliš málo výrazná. Proto by programátor musel nejprve číselná data nějakým způsobem sám zpracovat. Ze zpracovaných dat by pak pro každou měřenou metodu kódu musel vyhodnotit zdali došlo k výrazné změně výkonu, které by se měl dále věnovat.

Cílem této práce je vytvořit nástroj, který by programátorovi pomohl s vyhodnocováním změn výkonu. Nástroj by mělo být možné co nejjednodušeji zapojit do průběžné integrace. Jeho výstup by se měl co nejvíce podobat výstupu unit testovacích frameworků. Měl by tedy poskytovat výstup ve strojově čitelném formátu, ale také ve formátu čitelném pro člověka. Očekává se, že nástroj bude hlásit

výrazné změny výkonu mezi dvěma verzemi a že bude nezávislý na použitém měřícím frameworku.

# 3. Systém PerfEval

Když už jsou známy požadavky na aplikaci, tak je možné začít ji navrhovat. V této kapitole jsou podrobně popsány úvahy a rozhodnutí, které byly v průběhu vývoje provedeny. Jednotlivá rozhodnutí byla následně promítnuta do architektury a chování aplikace.

## 3.1 Analýza řešení

### 3.1.1 Měření výkonu

Před začátkem vývoje PerfEvalu bylo nutné zamyslet nad tím, jak můžou výsledky měření výkonu softwaru vypadat. V následujících odstavcích budou zmiňovány jednotlivé poznatky o výsledcích měření výkonu. Tyto poznatky vedly k tomu, jak se PerfEval chová a jakou má architekturu.

**Měřené veličiny.** Jak jsme v první kapitole viděli, tak měřící frameworky umožňují měřit mnoho různých fyzikálních veličin. Patří mezi ně například doba vykonávání metody, frekvence počtu operací za jednotku času a spotřeba paměti. Předpokládat se tedy dá jen to, že pokud vezmu dva výsledky měření výkonu ze dvou různých verzí, tak budou reprezentovány stejnou fyzikální veličinou a v lepším případě budou mít i stejnou fyzikální jednotku. Je tedy vhodné, aby výsledný systém byl schopen přijmout jakoukoli veličinu bez ohledu na jednotku.

**Identifikátory testů.** Protože testovací frameworky nepoužívají žádné identifikátory testů, tak je nejpřímější řešení k jejich rozpoznávání používat jména testovacích metod jako identifikátor. Tato jména poskytují ve výsledcích měření jak framework BenchmarkDotNet, tak framework JMH. Z dokumentace frameworku Criterion (Heisler, 2024) pro měření výkonu v jazyce Rust se název metody ve výsledcích nachází také. Z toho lze usoudit, že použití jména metody jako identifikátoru může být dostatečně obecné.

**Kompilace just-in-time.** V případě měření výkonu u jazyků, které jsou kompilované metodu JIT, je nutné být obezřetný. Je nutné všímat si, jaká data byla naměřena. Jazyky kompilované metodou JIT mohou při měření podléhat tzv. zahřívací fázi. Jedná se o fázi, kdy kód ještě není plně optimalizovaný, ale již se provádí a může být měřen. V závislosti na použitém měřícím frameworku je pak nutné naměřená data vhodně filtrovat. V případě, že by se data před a po optimalizaci nacházela v jedné sadě dat, mohla by být zkreslená.

### 3.1.2 Použití statistických metod pro analýzu dat

**Výsledek měření.** S testováním výkonu je problém. Když se spustí měření výkonu, tak výsledkem je pouhá datová sada. Tato datová sada nevypovídá nic o změně průběžného výkonu, která je zajímavá. Právě podle změny ve výkonu softwaru je možné zjistit, jestli je nutné kód optimalizovat, protože dochází k významným zhoršením.

**Šum.** Při měření výkonu dochází jako při jakémkoli jiném měření k šumu. Tento šum se projevuje tak, že pokud se měření opakuje při stejných podmínkách, tak se

naměřené hodnoty liší. Tomuto šumu se nelze vyvarovat. Měření proto opakujeme a naměřené hodnoty vyhodnocujeme pomocí statistických metod, které si s tímto šumem poradí.

**Testování hypotéz.** Pro vyhodnocování výkonu je využito metod testování hypotéz. Ve statistickém testování hypotéz se snažíme zamítnout nulovou hypotézu. V případě zamítnutí nulové hypotézy se předpokládá, že platí alternativní hypotéza. Popis testování hypotéz v této kapitole se řídí skripty Pravděpodobnost a statistika 1 (Šámal, 2023).

Při testování hypotéz rozeznáváme chyby I. a II. druhu. Chyba I. druhu znamená, že jsme nulovou hypotézu zamítli, i když platí. Chyba II. druhu znamená, že jsme ji nezamítli, ale ona neplatí.

V našem případě porovnávání výkonu bude nulová hypotéza tvrzení, že výkon dvou verzí softwaru je stejný. Jako alternativní hypotézu budeme uvažovat, že výkony dvou verzí softwaru jsou různé. Pomocí metody testování hypotéz bude zjišťováno jestli mají dvě spojité náhodné veličiny stejnou střední hodnotu. Hodnoty spojité náhodné veličiny jsou vždy hodnoty výkonu jedné testované metody programu jedné z verzí. Pro každou z verzí je tedy uvažovaná jedna náhodná veličina. Pokud hodnoty těchto veličin mají stejnou střední hodnotu, tak budeme tvrdit, že i výkon obou porovnávaných verzí je stejný.

Chyba I. druhu tedy v našem případě znamená, že jsme prohlásili, že výkony dvou verzí jsou různé, ačkoli jsou stejné. Pravděpodobnost chyby I. druhu je obvyklý parametr statistického testu. Pravděpodobnost chyby I. druhu bude dále značen jako parametr $\alpha$. Parametr $\alpha$ je součástí konfigurace systému PerfEval.

Následující dvě podkapitoly popisují vybrané statistické testy, které jsou využity v systému PerfEval. Dvouvýběrový Welchův t-test byl vybrán, protože t-test se podle (Šámal, 2023) používá pro porovnání středních hodnot dvou náhodných veličin. Welchův t-test byl vybrán, protože oproti Studentovu t-testu nevyžaduje stejné rozptyly porovnávaných náhodných veličin. Percentilový hierarchický bootstrap byl vybrán, protože je vhodný v situaci, kdy není možné předpokládat normální rozdělení dat. Poslední podkapitola popisuje jakým způsobem se řeší nevyvrácení nulové hypotézy.

### Welchův dvouvýběrový t-test

Welchův dvouvýběrový t-test se používá jako testová statistika při testování hypotéz. Tato statistika předpokládá, že náhodné veličiny jsou nezávislé a jejich rozdělení se blíží normálnímu rozdělení. Nezávislost náhodných veličin je dána vlastnostmi experimentu (Turčicová, 2021) a její zajištění je mimo doménu řešeného problému. PerfEval tedy v případě použití možnosti t-test možnou závislost zanedbává.

Normalitě náhodných veličin je možné se přiblížit díky centrální limitní větě (CLV). Se zajištěním normality nám pomůže samotná struktura naměřených výsledků. Výsledky měření obsahují běhy. Běhy obsahují jednotlivé naměřené hodnoty. Naměřené hodnoty představují vzorky náhodné veličiny. Pokud se budou v rámci t-testu namísto naměřených hodnot uvažovat průměry jednotlivých běhů, tak se podle CLV bude rozdělení těchto průměrů blížit normálnímu rozdělení.

Interval spolehlivosti pro Welchův t-test se spočítá podle dále uvedených vzorců. Postupně se počítají stupně volnosti, kritická hodnota t-rozdělení, chybo-

vost, dolní a horní mez intervalu. Vzorec pro výpočet stupňů volnosti je převzatý z Wikipedie Welchova t-testu (Wikipedia contributors, 2023). Ostatní vzorce jsou převzaty ze skript (Šámal, 2023).

$$\bar{X}_i = \frac{1}{n_i} \sum_{j=1}^{n_i} X_{ij}$$

$$s_i^2 = \frac{1}{n_i - 1} \sum_{j=1}^{n_i} (X_{ij} - \bar{X}_i)^2$$

$$df = \frac{(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2})^2}{\frac{s_1^4}{n_1^2(n_1-1)} + \frac{s_2^4}{n_2^2(n_2-1)}}$$

$$t = \psi_{df}^{-1}(1 - \frac{\alpha}{2})$$

$$marginOfError = t \cdot \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}$$

$$lowerBound = \bar{X}_1 - \bar{X}_2 - marginOfError$$

$$upperBound = \bar{X}_1 - \bar{X}_2 + marginOfError$$

Použité značení:

- $df$ - stupně volnosti
- $s_1^2$ - výběrový rozptyl první sady vzorků
- $s_2^2$ - výběrový rozptyl druhé sady vzorků
- $n_1$ - počet prvků první sady vzorků
- $n_2$ - počet prvků druhé sady vzorků
- $t$ - kritická hodnota t-rozdělení
- $\psi_{df}^{-1}$ - inverzní distribuční funkce t-rozdělení
- $\bar{X}_1$ - výběrový průměr první sady vzorků
- $\bar{X}_2$ - výběrový průměr druhé sady vzorku

Nakonec se již jen zkoumá, zdali tento interval obsahuje nulu. Pokud interval nulu neobsahuje, pak lze s pravděpodobností $1 - \alpha$ správně tvrdit, že nulová hypotéza neplatí.

**Algorithm 1:** MeanBootstrap1D

```
Input: measurements, replicationCount
Output: bootstrappedSamples
n = length(measurements);
samples = [];
for i = 0; i < replicationCount; i += 1 do
    resampledSamples = [];
    for j = 0; j < n; j = j + 1 do
        index = random() mod n;
        resampledSamples.add(measurements[index]);
    samples.add(mean(resampledSamples));
return samples;
```

**Algorithm 2:** MeanBootstrap2D

```
Input: runs1, runs2, replicationCount
Output: bootstrappedSamples
n = length(runs1);
m = length(runs2);
samples = [];
for i = 0; i < replicationCount; i += 1 do
    samples1 = [];
    for j = 0; j < n; j += 1 do
        index = random() mod n;
        samples1.add(MeanBootstrap1D(runs1[index], 1))[0];
    samples2 = [];
    for k = 0; k < m; k += 1 do
        index = random() mod m;
        samples2.add(MeanBootstrap1D(runs2[index], 1))[0];
    diff = mean(samples1)-mean(samples2);
    samples.add(diff);
return samples;
```

### Percentilový hierarchický bootstrap

Bootstrap je statistická metoda využívající tzv. resamplování. Stejně jako u dvouvýběrového t-testu se předpokládá nezávislost náhodných veličin. Podmínka nezávislosti bude zanedbána, protože ji není možné zaručit. Bootstrap se jako metoda používá v případě, kdy o náhodných veličinách není možné určit téměř žádné silné předpoklady. Díky této vlastnosti je bootstrap pro testování hypotéz o výkonu verzí softwaru použit.

Podle percentilového bootstrapu se interval spolehlivosti nalezne tak, že se hranice intervalu stanoví jako $\frac{\alpha}{2}$-tý a $\frac{1-\alpha}{2}$-tý percentil z resamplovaného souboru. Tyto dvě hodnoty budou představovat hranice intervalu.

Zkoumanou náhodnou veličinou je rozdíl dvou náhodných veličin. Tyto dvě náhodné veličiny jsou dány měřením výkonnosti dvou verzí softwaru. Nulová hy-

potéza, která je vyvracena, tvrdí, že obě veličiny mají stejnou střední hodnotu. Pokud tedy interval spolehlivosti neobsahuje nulu, tak můžeme nulovou hypotézu vyvrátit, protože s pravděpodobností $1-\alpha$ o ní test správně prohlásil, že neplatí.

Naměřené vzorky však nejsou prostý statistický soubor. Jedná se o hierarchický soubor dat. Každé jedno měření se skládá z jednoho, nebo více běhů. Každý běh se skládá z jednoho, nebo více naměřených údajů. Vytváření bootstrapového statistického souboru tedy vypadá trochu odlišně.

MeanBootstrap2D ukazuje, jak vypadá výběr nového statistického souboru. Vyberou se náhodné běhy z jednotlivých výsledků měření. Z těchto běhů se získá 1D bootstrap. Novým prvkem vytvářeného statistického souboru se stane rozdíl těchto bootstrapů.

#### Co dělat v případě nevyvrácení hypotézy?

V případě nevyvrácení nulové hypotézy nám statistické testy nedávají žádnou informaci. I přes tuto vlastnost statistických testů, je pro praktické užití nutné se rozhodnout, zdali změnu výkonu hlásit. Stále je nutné se vyvarovat chyby II. druhu. Proto v tomto případě nebudeme hlásit zhoršení výkonu, pokud bude získaný interval spolehlivosti dostatečně úzký. Pokud má tedy interval spolehlivosti dolní mez $D_{LOW}$ a horní mez $D_{HIGH}$, pak je jeho šířka $D_{HIGH} - D_{LOW}$. Odhadovaný průměr by byl $\frac{D_{HIGH}+D_{LOW}}{2}$. Relativní šířka intervalu je tedy poměr šířky a průměru, tedy $\frac{2\cdot(D_{HIGH}-D_{LOW})}{D_{HIGH}+D_{LOW}}$.

V případě většího množství vzorků je možné zužovat interval spolehlivosti. Vztah mezi šířkou a počtem vzorků odpovídá $O(\frac{1}{\sqrt{n}})$, kde $n$ je počet vzorků. Z daného množství vzorků je tedy možné odhadnout, kolik vzorků je ještě zapotřebí změřit.

V případě, že je interval dostatečně úzký, nebudeme zhoršení výkonu hlásit a řekneme, že výkon obou verzí je stejný. V případě, že interval není dostatečně úzký, prohlásíme, že vzorků není dost.

PerfEval tedy v konečném důsledku rozlišuje tři základní výsledky porovnání výkonu verzí. Test nahlásí zhoršení výkonu právě tehdy, když bude nulová hypotéza vyvrácena a zároveň bude průměrný výkon novější verze horší než průměrný výkon starší verze. Test nenahlásí žádnou změnu výkonu, pokud nulová hypotéza nebyla vyvrácena a interval spolehlivosti bude dostatečně úzký. V případě, že nulová hypotéza nebyla vyvrácena a interval spolehlivosti je příliš široký, test nahlásí, že vzorků není dostatečné množství.

### 3.1.3 Použité statistické metody

Pro porovnání dvou výsledků měření se používají statistické metody. Statistické metody se používají k zjištění, jestli výsledky měření považované za náhodné veličiny mají stejnou střední hodnotu a jestli je vzorků dostatečné množství. V předchozí kapitole byly popsány dvě statistické metody, které jsou v systému PerfEval implementovány. Jedná se o metody **bootstrap** a **t-test**.

**Bootstrap.** Výhodou bootstrapu je, že není vyžadovaný předpoklad normálního rozdělení dat. Bootstrap je implementován pomocí algoritmů z předchozí podkapitoly. Je tedy patrné, že výpočet je pomalý, protože dochází k velkému množství náhodných výběrů z dvoudimenzionální sady dat. Pokud má uživatel dostatečné

množství času měl by této metodě dát přednost právě kvůli tomu, že není vyžadován předpoklad normálního rozdělení dat. Jak již bylo řečeno, tak zajištění normality dat je totiž mimo doménu řešeného problému.

**T-test.** T-test je použitý ve variantě Welchova t-testu. Tento test je výhodný tím, že je rychlejší než bootstrap. Rychlejší je proto, že se jedná o pouhé dosazení hodnot do vzorců. Nevýhodou je, že je vyžadován předpoklad normálního rozdělení dat. Tento předpoklad však není zaručen, a tudíž může dojít k zkreslení výsledků. T-test je tedy vhodný pro případy, kdy je množství naměřených vzorků velké a uživatel má málo času na vyhodnocení výsledků.

### 3.1.4 Spouštění testů systémem PerfEval

V počátku vývoje bylo nutné se rozhodnout, jakým způsobem bude systém přijímat a zpracovávat výsledky testů. V úvahu přicházela varianta **Měření výkonu uživatelem**, že uživatel provede měření výkonu sám. Druhá varianta byla **Měření výkonu PerfEvalem** tak, že uživatel systému vysvětlí, jakým způsobem se testování výkonu spouští. Pokud by byla zvolena tato varianta, tak by bylo nutné nalézt dostatečně univerzální způsob spouštění testů.

**Měření výkonu PerfEvalem.** Aplikace a benchmarky pro měření výkonu mohou být jak konzolové, tak grafické aplikace. Pokud by PerfEval měl měření provádět sám, tak by téměř určitě nebyl schopen pracovat s grafickými aplikacemi, ale byl by schopen spouštět programy s parametry na příkazové řádce.

Dále by bylo nutné vysvětlit, jak vypadá výstup spouštěných testů. Když pomineme formát, tak je nutné zjistit, kam program, který provádí měření, výsledky ukládá. Benchmarkovací systém BenchmarkDotNet například vypisuje výsledky měření v podobě tabulky na standardní výstup a zároveň ukládá strojově čitelné výsledky do speciálního k tomu určeného adresáře.

Pokud by PerfEval využíval této varianty, tak by uživatel při inicializaci systému musel zadat, jak spustit testy a kam se uloží výsledek. Tímto způsobem by došlo k tomu, že PerfEval by začal určovat, jak mají vypadat programy, jejichž výstupy přijímá. **Měření výkonu uživatelem.** V této variantě tedy uživatel spouští testy sám a PerfEval pouze porovnává výsledky. Při inicializaci uživatel oproti předchozí variantě pouze vybere vhodný parser. Zvolený parser umí zpracovat příslušný formát dat z benchmarkovacího frameworku, který uživatel používá. Toto řešení bylo vybráno a implementováno v systému PerfEval.

Vybrané řešení tedy od uživatele požaduje krok navíc. Nicméně dává uživateli mnohem větší prostor pro to, jak spouští testy a kam ukládá výsledky. O tom, že existují výsledky měření dané verze, a kde se nachází, uživatel pouze informuje PerfEval. Systém tak ani do budoucna neklade žádné nároky na to, jak má uživatel testy spouštět, ani kde se mají výsledky ukládat.

### 3.1.5 Rozpoznání formátu výsledků měření

Systém, který porovnává výsledky měření výkonu, by měl mít informace o tom v jakém formátu jsou data uložena a který benchmarkovací framework je vytvořil. Podle použitého frameworku a formátu je totiž možné výsledky měření zpracovat pomocí programu a transformovat data o měření tak, aby jim systém rozuměl.

Problém je tedy v tom, jak se systém dozví o tomto formátu a o použitém frameworku.

Nejpříjemnější řešení pro uživatele by bylo, že by systém sám přišel na to, který framework a formát je použitý. Uživatel by totiž nemusel vědět pomocí jakého frameworku a v jakém formátu data ukládá. Pro systém by však mohl být problém různé frameworky rozlišit.

Při rozlišování by se totiž musel podívat do dat uložených v souboru a na základě obsahu určit o jaký formát a framework se jedná. Správné určení frameworku a formátu by bylo zásadní pro správnou reprezentaci dat. Samotné rozlišování frameworků a formátu by bylo obtížné, protože soubory s výsledky měření obsahují podobná data a položky, ale hierarchie struktur ve kterých jsou uloženy jsou různé. Při následném rozlišování většího množství formátů a frameworků by tedy systém automatického rozpoznávání začal být příliš komplikovaný, aby si zachoval přesnost.

Docházelo by také k dvojímu čtení souboru z paměti. První čtení souboru by sloužilo k rozpoznání formátu, aby systém zjistil, jak má data ze souboru zpracovávat. Při druhém čtení souboru by již transformoval data tak, aby jim rozuměla vyhodnocovací část systému.

PerfEval tedy řeší tento problém tak, že uživatel při inicializaci zadá jméno jednoho z dostupných parserů. Předpokládá se tedy, že pokud uživatel používá výkonnostní testy, tak framework a výstupní formát je shodný s těmi které zvolený parser rozpoznává. Pokud má PerfEval parser pro tento framework a formát dat parser, pak je schopný vyhodnocovat výsledky měření. V opačném případě si tento parser může uživatel doimplementovat.

Tento přístup umožňuje snazší implementaci nových parserů. Není totiž nutné k těmto parserům implementovat také sadu pravidel, kdy má být použitý. Uživatele tento přístup omezuje v tom, že musí znát framework a formát ve kterém se výsledky měření nachází. Protože psaní výkonnostních testů není jednoduché, tak lze předpokládat, že uživatel je dostatečně zkušený, aby tuto znalost měl.

### 3.1.6 Kdy zpracovávat naměřená data?

Systém musí v některém bodě výpočtu zpracovat data z měření. Existuje několik možností, kdy je možné toto zpracování do formátu, kterému bude rozumět, udělat. Je možné buď zpracovat data ihned po tom, co se systém dozví o jejich existenci, nebo až těsně před vyhodnocováním.

Pokud by se data zpracovávala hned po tom, co se o nich systém dozví, tak to nutně neznamená, že již má proběhnout vyhodnocování. Je tedy nutné zvolit nějaký formát do kterého zpracovaná data uložit. Při vyhodnocení by se pak data z tohoto formátu musela opět zpracovat. Tento způsob zpracování dat by měl význam pouze v případě, že by první předzpracování vedlo k velkému zrychlení druhého zpracování. Zároveň by toto řešení vedlo k dvojímu ukládání dat, protože by někde byl uložený původní výsledek měření v původním formátu, a také soubor se předzpracovanými výsledky měření, který by obsahoval data se stejným významem.

Výše zmíněné nevýhody vedly k tomu, že systém zpracovává data z původního formátu těsně před vyhodnocováním. Podle zvoleného parseru se tedy data naparsují těsně před vyhodnocením ze souborů, které vygeneroval přímo framework

pro měření výkonu.

### 3.1.7 Jak přistupovat k naměřeným datům?

Systém, který porovnává výsledky měření, by měl mít nějaké informace o tom, k jaké verzi se měření vztahuje, nebo kde se soubor s výsledky nachází. Proto bylo při vývoji nutné se zamyslet nad tím, jakými způsoby lze tyto informace získat a spravovat. Systém PerfEval potřebuje o výsledcích měření vědět, kde jsou uloženy a ke které verzi softwaru bylo měření provedeno.

**Složka s výsledky měření.** První zvažovaná varianta byla, že by existovala složka, do které by uživatel vkládal výsledky měření. Tento adresář by byl určený systémem PerfEval, nebo by byla zadaná cesta k němu při inicializaci systému. Po každém měření by tedy uživatel výsledek pouze vložil do správné složky a o nic víc by se nemusel starat.

Toto řešení má několik nevýhod. Omezuje uživatele v tom, jak musí výsledky testů ukládat. Dále by se špatně určovalo, ke které verzi bylo měření provedeno, protože toto se ve výsledcích měření běžnými frameworky neudává. Pravděpodobně by proto vznikla hierarchie v tomto adresáři a uživatel by například pomocí pojmenovávání složek určoval, ke které verzi se měření vztahuje.

Celkově tedy tento způsob poskytuje jednoduché zjištění, kde se výsledek testu nachází. Je to pevně dané. Není ale možné jednoduše zjistit verzi, protože je nutné projít adresářovou strukturu, což může být při velkém množství dat pomalé.

**Cache a složka s výsledky měření.** Druhé uvažované řešení bylo vylepšení prvního řešení o cache a sledování času modifikace adresářů. Verze se totiž dobře určují při vkládání, ale špatně dohledávají. Pravděpodobně by se používání některých verzí k porovnání opakovalo častěji než u jiných. Zároveň pokud by nejnovější verze nebyla v cache, tak by se dohledávala postupným průchodem na základě nejmladšího data změny adresáře v adresáři s výsledky měření. Cache by byla tvořena posledními několika desítkami naposledy použitých záznamů s cestou k výsledkům měření a verzí.

Toto řešení má ale podobné nevýhody jako první. V případě, že se verze porovnává poprvé, tak záznamy o ní jistě nejsou v cache a musí se prohledat adresářová struktura. Pokud je poslední úprava cache starší než nejnovější úprava nějakého podadresáře v adresáři s výsledky měření, tak je nutné tuto část adresářové struktury projít. V procházené části adresářové struktury je vždy nutné prozkoumat zdali se jedná o soubor verze, která je v cache, a tak by se do ní měl přidat záznam o tomto souboru. Popsaná práce s cache měla za následek poslední uvažované řešení, kde již nezáleží na tom, kde jsou uložené výsledky měření. Jediné, co budeme předpokládat, je, že uživatel soubory s výsledky měření nepřesouvá.

**Databáze s metadaty o výsledcích měření.** Poslední uvažované řešení je strukturované ukládání metadat o výsledcích měření. Využije se databáze o jedné tabulce, kde je uvedena cesta k výsledku měření a verze ke které byl výsledek změřen. Uživatel do systému nahlásí, že má nový výsledek měření, kde je uložen, a k jaké verzi softwaru je měření provedeno. Systém si tato data pouze poznamená do databáze. Při vyhodnocení pak pomocí databázových dotazů nalezen snadno všechny soubory s výsledky měření k potřebné verzi.

Toto řešení poskytuje jak rychlé vkládání nových výsledků, tak rychlé vyhledávání výsledků měření k dané verzi. Díky těmto vlastnostem bylo vybráno

a implementováno v systému PerfEval.

### 3.1.8 Formát výstupu

Při volbě formátu v jakém se budou výsledky výkonnostních testů prezentovat je nutné se zamyslet nad tím, kdo je bude číst a zpracovávat. Výsledky vyhodnocení bude zpracovávat zejména člověk a průběžná integrace v rámci verzovacího nástroje Git.

Pro člověka je čitelný formát zpracovaných dat tabulka s údaji o tom, který test prošel a který ne. Jeden z výstupních formátu systému PerfEval je takováto tabulka na příkazové řádce. Druhý formát s daty čitelnými pro člověka je tabulka na HTML stránce, kterou PerfEval umí vygenerovat. Výstup v HTML formátu je výhodný, protože je možné soubor s výsledky otevřít v internetovém prohlížeči, který je dnes dostupný téměř na každém zařízení.

Strojem čitelný formát je XML soubor. XML má formát shodný s JUNIT XML formátem, který je běžně používaný v průběžné integraci při hlášení výsledků unit testů. Druhým zvoleným formátem je JSON, který je dnes běžně používaný pro serializaci dat v různých programovacích jazycích. Umožňuje tak snadnou integraci s dalšími programy.

## 3.2 Architektura systému PerfEval

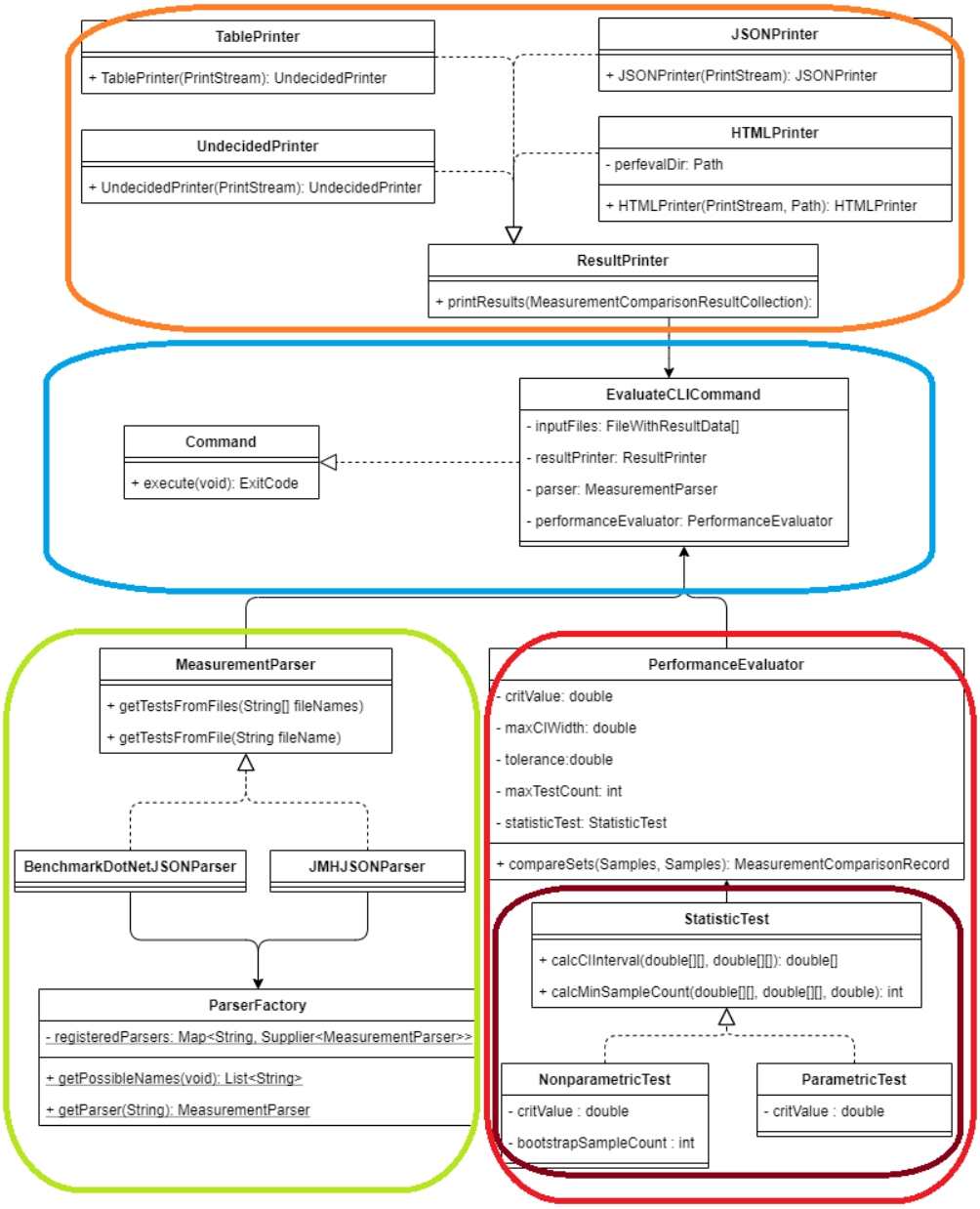
Tato kapitola se věnuje popisu konkrétní architektury systému PerfEval. Popisuje jakým způsobem jsou jednotlivé části systému navrženy a jak spolu vzájemně interagují.

Samotné vyhodnocování jakožto porovnání dvou výsledků měření jedné sady testů výkonu je proces, který lze rozdělit do několika kroků. Zpracování dat uložených v souborech s výsledky měření až po vypsání výsledků vyhodnocení probíhá následujícím způsobem:

1. Načtení a naparsování výsledků měření ze souborů.
2. Pomocí statistického testu porovnat dvě sady naměřených dat.
3. Výsledek porovnání vypsat uživateli.

Obrázek 3.1 znázorňuje třídy, které se podílejí na vyhodnocování výsledků měření. Třída `EvaluateCLICommand` je hlavní třídou, která zastřešuje celý proces vyhodnocování. Tato třída využívá implementace rozhraní `ResultParser`, `StatisticalTest` a `ResultPrinter`. Implementace těchto rozhraní jsou předány při konstrukci třídy `EvaluateCLICommand`. Celé vyhodnocování je tedy postaveno na rozhraních a jejich implementacích, které jsou zvýrazněny na obrázku 3.1.

Implementace rozhraní `ResultParser` slouží k naparsování výsledků měření z jednoho formátu a frameworku. Starají se tedy o první krok vyhodnocování. Implementace rozhraní `StatisticalTest` slouží k porovnání dvou instancí třídy `Samples`. Tuto implementaci využívá třída `PerformanceEvaluator` spolu s nastavením z konfiguračního souboru pro vyhodnocení výsledků měření. Implementace rozhraní `ResultPrinter` slouží k vypsání výsledků vyhodnocení.




Obrázek 3.1: Objektový návrh části PerfEvalu pro porovnávání výsledků měření

Po vypsání výsledků porovnání se jen podle voleb z konfiguračního souboru nastaví exit kód programu a ukončí se vyhodnocování. Konfigurační soubor určuje jestli se mimo zhoršení výkonu hlásí také zlepšení výkonu a nemožnost doměření dostatečného množství vzorků.

Samotná architektura systému je tedy postavena na rozhraních, které obklopují třídu `EvaluateCLICommand`. Pomocí nových implementací těchto rozhraní je možné systém rozšiřovat o nové možnosti zpracování výsledků měření. Architektura dále umožňuje snadné rozšíření o nové formáty vstupu i výstupu. Systém byl navržen tak, aby byl dostatečně flexibilní v případě, že by bylo potřeba přidat nebo změnit nějaký implementační detail.

# 4. Uživatelská dokumentace systému PerfEval

## 4.1 Rychlý start

V této kapitole bude v několika stručných krocích vysvětleno, jak nainstalovat aplikaci PerfEval a porovnat pomocí ní výkonu dvou verzí softwaru. Rychlý start je určen pro uživatele Linuxových distribucí. Na platformě Windows je vhodné využít WSL (Windows Subsystem for Linux) nebo jiný nástroj, který umožní spouštění Linuxových příkazů.

1. V adresáři se zdrojovým kódem spusťte příkaz `./gradlew jar`. Tento příkaz zkompiluje a sestaví aplikaci PerfEval.
2. Přejděte do adresáře svého projekt, kde chcete PErfeval používat.
3. Zadejte příkaz `./perfeval.sh init --benchmark-parser parser-name`. V případě, že nevíte jaký parser použít, nezadávejte tento argument a systém vám nabídne dostupné parsery. Parser vyberte podle použitého testovacího frameworku a výstupního formátu. Po vybrání parseru zadejte jeho název jako argument parametru `--benchmark-parser`.
4. Přidejte výsledky měření referenční verze pomocí příkazu `./perfeval.sh index-new-result --path path-to-results --version 1.0`.
5. Přidejte výsledky měření nové verze pomocí příkazu `./perfeval.sh index-new-result --path path-to-results --version 2.0`.
6. Porovnejte výsledky měření pomocí příkazu `./perfeval.sh evaluate`.
7. Na standardní výstup bude vypsána tabulka s výsledky porovnání verzí 1.0 a 2.0. Pokud došlo k zhoršení výkonu, bude program ukončen s nenulovým exit kód.

## 4.2 Instalace

Instalace systému PerfEval probíhá pomocí nástroje gradle. V adresáři se zdrojovým kódem se nachází spustitelný soubor `gradlew`, který je určen pro kompilaci a sestavení PerfEvalu v systémech Linux. Pro kompilaci a sestavení v systémech Windows je určený spustitelný soubor `gradlew.bat`.

- Instalace v systému Linux: `./gradlew jar`
- Instalace v systému Windows: `gradlew.bat jar`

Po kompilaci a sestavení pomocí zmíněných souborů vznikne soubor typu JAR v podadresáři `build/libs`. Soubor je možné spustit pomocí příkazu `java -jar PerfEval.jar`, nebo pomocí připravených skriptů `perfeval.sh` pro systémy Linux a `perfeval.bat` pro systémy Windows. Tyto skripty se nachází taktéž v adresáři se zdrojovým kódem.

- Spuštění v systému Linux: `./perfeval.sh`
- Spuštění v systému Windows: `perfeval.bat`

## 4.3 Dostupné příkazy

Tato část práce se zabývá jednotlivými příkazy systému PerfEval a jejich parametry. V jednotlivých podkapitolách je vysvětleno, k čemu se daný příkaz používá. V podkapitolách se nachází také informace o volitelných a povinných parametrech jednotlivých příkazů.

### 4.3.1 Příkaz init

Příkaz init slouží k inicializaci systému v rámci aktuálního pracovního adresáře. Systém PerfEval po spuštění hledá v pracovním adresáři složku s názvem .performance. Pokud složka není nalezena a nebyl zadán příkaz init, končí s chybou, že systém není inicializovaný.

#### Povinné argumenty:

`benchmark-parser` Nastavení parseru, který se bude pro tento projekt používat. Jméno parseru je zadávané jako parametr tohoto příznaku. Parser se volí podle použitého testovacího frameworku a výstupního formátu. Jedná se o jediný parametr konfiguračního souboru, který lze zadat při inicializaci. Důvodem je, že je to jediný konfigurační parametr, jehož hodnotu není možné nastavit nějakým výchozím způsobem. Určuje totiž podobu a formát vstupních dat, které PerfEval očekává.

#### Volitelné argumenty:

`force` Příznak, který vynutí inicializaci i v případě, že je systém v adresáři již inicializovaný.

### 4.3.2 Příkaz index-new-result

Příkaz index-new-result slouží k přidání výsledků měření výkonu do databáze. Databáze je pro každý projekt zvlášť a je tedy možné na jednom zařízení systémem PerfEval spravovat více projektů. Při přidávání informací o souboru s výsledky je nutné zadat cestu k tomuto souboru. Verze, ke které byly výsledky změřeny, může být zadaná také, nebo ji systém zkusí určit podle git repozitáře.

#### Povinné argumenty:

`path` Parametr tohoto příznaku udává cestu k souboru s výsledky měření.

### Volitelné argumenty:

`version` Parametr tohoto příznaku udává textovou reprezentaci verze SW, která se měřila.

`tag` Parametr tohoto příznaku udává tag verze měření.

### 4.3.3 Příkaz index-all-results

Příkaz index-all-results slouží k přidání více výsledků měření výkonu do databáze. Při přidávání informací o souborech s výsledky je nutné zadat cestu k adresáři s těmito výsledky. Budou přidány všechny (i zanořené) soubory v tomto adresáři. Verze, ke které byly výsledky změřeny, může být zadaná také, nebo ji systém zkusí odhadnout podle git repozitáře.

### Povinné argumenty:

`path` Parametr tohoto příznaku udává cestu k adresáři s výsledky měření.

### Volitelné argumenty:

`version` Parametr tohoto příznaku udává textovou reprezentaci verze SW, která se měřila.

`tag` Parametr tohoto příznaku udává tag verze měření.

### 4.3.4 Příkaz evaluate

Příkaz evaluate porovnává dvě poslední zaznamenané verze, které byly změřeny. Verze k porovnání je možné specifikovat také manuálně pomocí příznaků. Výstupem je tabulka nebo JSON s výsledky porovnání. V případě, že alespoň u jednoho porovnání došlo ke zhoršení výkonu, bude selhání signalizováno exit kódem 1.

### Volitelné argumenty:

`new-version` Parametr udává textovou reprezentaci verze, která se má při porovnání považovat za novější.

`new-tag` Pouze soubory s tímto tagem budou použity k porovnání jako novější.

`old-version` Parametr udává textovou reprezentaci verze, která se má při porovnání považovat za starší.

`old-tag` Pouze soubory s tímto tagem budou použity k porovnání jako starší.

`t-test` T-test bude při statistickém porovnání použitý místo bootstrapu.

`json-output` Formát výstupu bude JSON.

`html-output` Výstup bude ve formátu HTML stránky.

`html-template` Při použití příznaku html-output je možné ještě jako argument tohoto příznaku dodat adresu nové HTML šablony pro vypsání výsledků.

`junit-xml-output` Výstup bude ve formátu JUNIT XML. Tento formát je běžně přijímaný nástrojem Git v rámci průběžné integrace.

`filter` Parametr tohoto příznaku umožňuje filtrovat výsledky podle názvu testovací metody. Možnosti jsou `test-id`, `size-of-change`, `test-result`.

### 4.3.5 Příkaz list-undecided

Příkaz list-undecided porovnává dvě poslední zaznamenané verze, které byly změřeny. Verze k porovnání je možné specifikovat také manuálně pomocí příznaků. Výstupem jsou dva sloupce oddělené znakem tabulátoru. V prvním sloupci je název testovací metody. Ve druhém sloupci je počet měření, které jsou potřeba, aby bylo možné s dostatečnou pravděpodobností říct, že je výkon stejný.

#### Volitelné argumenty:

`new-version` Parametr udává textovou reprezentaci verze, která se má při porovnání považovat za novější.

`new-tag` Pouze soubory s tímto tagem budou použity k porovnání jako novější.

`old-version` Parametr udává textovou reprezentaci verze, která se má při porovnání považovat za starší.

`old-tag` Pouze soubory s tímto tagem budou použity k porovnání jako starší.

`t-test` T-test bude při statistickém porovnání použitý místo bootstrapu.

`filter` Parametr tohoto příznaku umožňuje filtrovat výsledky podle názvu testovací metody. Možnosti jsou `test-id`, `size-of-change`, `test-result`.

### 4.3.6 Příkaz list-results

Příkaz list-results vypíše informace o souborech uložených v databázi. Příkaz nemá žádné argumenty. Výstupní formát je tabulka s informacemi o souborech. Poskytuje jednoduchý přehled o souborech v databázi PerfEvalu.

## 4.4 Konfigurační soubor

Po spuštění systému PerfEval s příkazem `init` dojde k vytvoření složky .performance v pracovním adresáři. Ve vytvořené složce bude konfigurační soubor config.ini. Tento soubor obsahuje nastavení spojené s používáním systému PerfEval při vyhodnocování výkonu jednoho projektu. Změnou hodnot v konfiguračním souboru je možné omezeně změnit chování systému PerfEval.

## Hodnoty v konfiguračním souboru

`falseAlarmProbability` Určuje pravděpodobnost chyby I. druhu při testování hypotézy, že výkony verzí jsou stejné.

`accuracy` Určuje maximální relativní šířku intervalu spolehlivosti

`minTestCount` Určuje minimální počet testů (běhů), který bude dodán.

`maxTestCount` Určuje maximální počet testů (běhů), který je uživatel schopný změřit.

`tolerance` Určuje maximální pokles výkonu (relativně vůči starší verzi), který nezpůsobí selhání.

`git` Určuje, jestli projekt podléhá správě verzí pomocí nástroje git. Nabývá hodnot TRUE a FALSE.

`parserName` Jméno parseru, který bude použit při zpracovávání souborů s výsledky měření.

`highDemandOfRuns` Určuje, jestli má PerfEval hlásit, že je zapotřebí vyšší počet běhů, než udává maxTestCount. Nabývá hodnot TRUE a FALSE.

`improvedPerformance` Určuje, jestli má PerfEval hlásit, že výkon novější verze je lepší.

# 5. Programátorská dokumentace systému PerfEval

## 5.1 Architektura systému

Architekturu systému PerfEval je možné rozdělit do dvou částí. První část systému tvoří parser příkazové řádky a tzv. setup třídy. Druhou částí jsou tzv. command třídy, které provádí skutečně požadovanou činnost. V této části dokumentace nemusí být některé podrobnosti o implementaci, které je možné najít v automaticky generované dokumentaci JavaDoc. Jedná se například o argumenty zmiňovaných metod.

Vnitřní struktura běhu je velice jednoduchá. Metoda `main` má tři jednoduché úkoly. Zavolat metodu `getCommand` na třídě `Parser`. Tato metoda vrátí objekt typu `Command`, na kterém metoda `main` zavolá metodu `execute`. Metoda `execute` při návratu vrací enumerátor typu `ExitCode`. Na objektu `ExitCode` pak na konci metoda `main` zavolá metodu `exit`, která ukončuje program s požadovaným exit kódem. Metody `getCommand` a `execute` mohou skončit s výjimkami `ParserException` a `PerfEvalCommandFailedException`. Pro tyto případy tyto výjimky obsahují položku `exitCode`, na které metoda `main` opět zavolá metodu `exit`.

### 5.1.1 Použité návrhové vzory

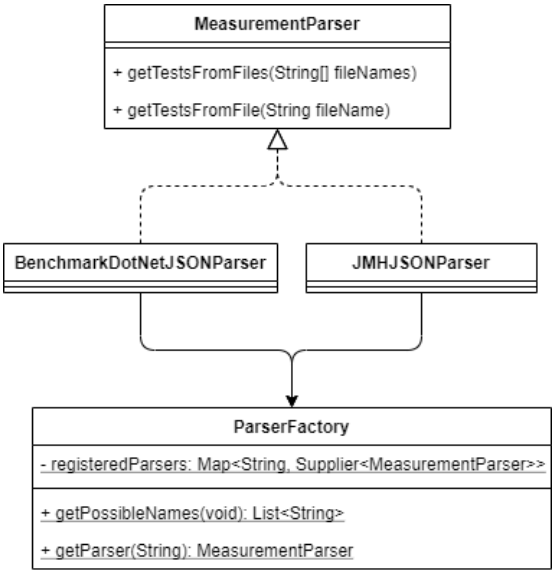
Součástí architektury systému PerfEval je několik běžných návrhových vzorů. Návrhové vzory z této kapitoly jsou převzaty z knihy Design Patterns: Elements of Reusable Object-Oriented Software Gamma a kol..

Prvním z použitých návrhových vzorů je factory. Factory je použitý při práci s parsery vstupních souborů. Factory třídou podle tohoto návrhového vzoru je zde třída `ParserFactory`. Tato třída konstruuje parsery podle parametru metody `getParser`.

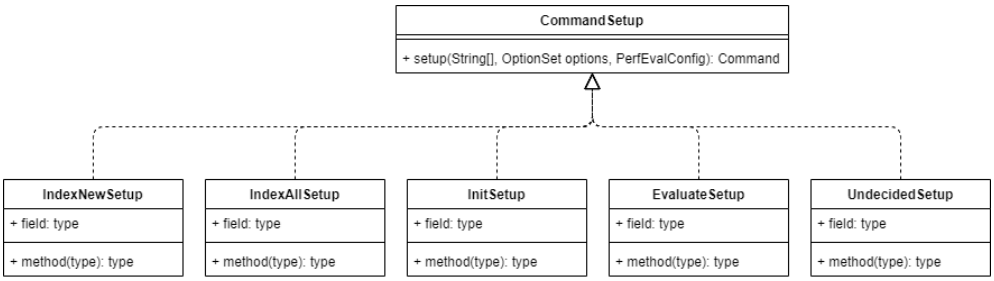
Druhým použitým návrhovým vzorem je builder. Jednotlivé builder třídy jsou implementacemi rozhraní `CommandSetup`. Tyto builder třídy v rámci systému PerfEval v rámci metody setup konstruují objekt typu `Command`.

Dalším použitým návrhovým vzorem je strategy. Tento návrhový vzor je využitý při volbě statistického testu, který se bude používat pro vyhodnocení, a pro volbu toho, jakou podobu výstupu má program vyrobit. Na obrázku s příkladem tříd jsou vidět jednotlivé strategie pro volbu formy výstupu.

Posledním použitým návrhovým vzorem je návrhový vzor command. Protože je PerfEval konzolová aplikace ovládaná pomocí příkazů, tak je příhodné jej použít. Na obrázku jsou vidět dostupné implementace rozhraní `Command`. Pro každý z příkazů systému PerfEvalu se zkonstruuje nějaká jeho implementace a na ní se zavolá její metoda `execute`.




Obrázek 5.1: Struktura části PerfEvalu - návrhový vzor factory



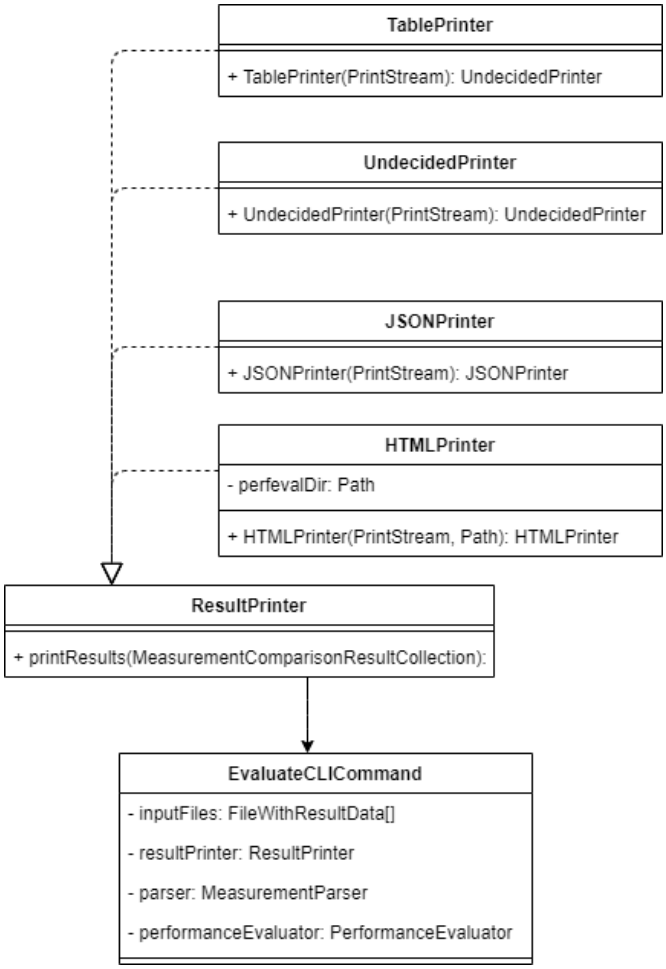

Obrázek 5.2: Struktura části PerfEvalu - návrhový vzor builder

### 5.1.2 Parser a setup třídy

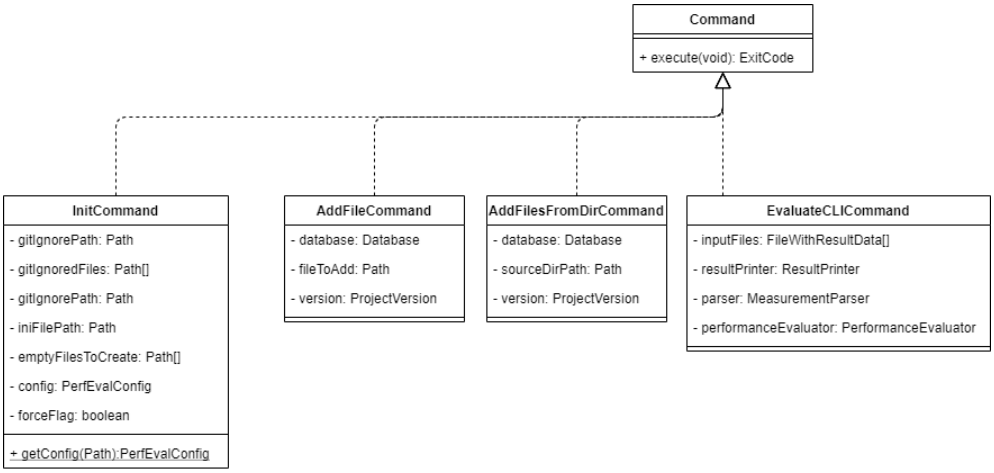
Třídu `Parser` tvoří slovník `commandPerSetup` a metoda `getCommand`. Ve slovníku jsou k jednotlivým řetězcovým klíčům přiřazeny objekty typu `Supplier`, které mají za úkol vracet instance typu `CommandSetup`. Klíče jsou skutečné řetězce zadávané uživatelem. `Parser` pak na základě příkazu zadaného na příkazové řádce zkonstruuje skrze `Supplier` získaný ze slovníku správnou instanci třídy `CommandSetup`. Na této instanci se zavolá metoda `setup`, která vrací objekt typu `Command`, podle definice rozhraní `CommandSetup`. Metoda `getCommand` je volaná z metody `main`, která tvoří hlavní metodu programu.

Třída `Parser` předává do metody `setup` objekty `OptionSet`, které reprezentují argumenty příkazové řádky, a `PerfEvalConfig`. Třída `PerfEvalConfig` reprezentuje globální konfiguraci systému PerfEval podle konfiguračního souboru. `Parser` tedy skrze statické metody této třídy zařídí přečtení konfiguračního souboru do objektu `PerfEvalConfig`. Objekt `OptionSet` reprezentuje parametry a značky zadané uživatelem na příkazovou řádku. Jedná se o objekt z knihovny joptsimple. Rozpoznávány jsou všechny značky blíže specifikované ve třídě `SetupUtilities`.

`SetupUtilities` je třída obsahující statické položky a řetězcové konstanty. Tyto položky a konstanty slouží pro zpracování argumentů příkazové řádky. Setup třídy a `Parser` využívají jednotlivé metody této třídy. Protože některé setup třídy využívají stejných metod ze třídy `SetupUtilities`, tak jsou tyto metody




Obrázek 5.3: Struktura části PerfEvalu - návrhový vzor strategy




Obrázek 5.4: Struktura části PerfEvalu - návrhový vzor command

umístěny právě ve třídě `SetupUtilities`.

`InitSetup` je třída, která má za úkol připravit instanci `InitCommand`. Protože se jedná o setup třídu, tak připravuje data pro práci instance typu `Command`, který konstruuje. Poté, co v rámci metody `setup` připraví potřebná data zkonstruuje `InitCommand` a vrátí jej. Tímto je `InitCommand` připraven k práci.

`IndexNewSetup` je třída pro přípravu instance `AddFileCommand`. Tato setup třída dodává při konstrukci instance `AddFileCommand` cestu k přidávanému souboru, implementaci rozhraní `Database` a instanci objektu `ProjectVersion`. Instance objektu `ProjectVersion` popisuje verzi softwaru pro kterou byl nově přidávaný výsledek změřen. Zkonstruovanou instanci třídy `AddFileCommand` vrací metoda `setup`.

`IndexAllSetup` je třída pro přípravu instance `AddFilesFromDirCommand`. Té je při konstrukci dodána implementace rozhraní `Database` a cesta ke složce, ze které se budou přidávat soubory, a instance objektu `ProjectVersion`. Instance objektu `ProjectVersion` popisuje verzi softwaru pro kterou byly nově přidávané výsledky změřeny. Zkonstruovanou instanci třídy `AddFilesFromDirCommand` vrací metoda `setup`.

`EvaluateSetup` je třída která připravuje `EvaluateCLICommand`. Tato setup třída musí v databázi najít soubory s výsledky měření, které bude konstruovaný `EvaluateCLICommand` porovnávat. Dále vyrábí správné implementace rozhraní `StatisticTest` a `ResultPrinter`, podle argumentů z příkazové řádky. `EvaluateSetup` musí vyrobit také instanci objektu `PerformanceEvaluator`. Tuto instanci vyrobí z nastavení v objektu `PerfEvalConfig` a ze zmíněné implementace `StatisticTest`. `EvaluateCLICommand` je zkonstruovaný z vyrobených instancí `PerformanceEvaluator`u, `ResultPrinter`u, `MeasurementParser`u a vstupních souborů s výsledky. `MeasurementParser` je součástí objektu `PerfEvalConfig`.

`UndecidedSetup` konstruuje `EvaluateCLICommand` způsobem z předchozího odstavce. Jediný rozdíl je, že se použije pevně stanovený `ResultPrinter` typu `UndecidedPrinter`. Tato implementace `CommandSetup` vznikla proto, aby se vypsání informace o nerozhodných testech vypisovala pomocí odlišného příkazu.

Třída `ListResultsSetup` připravuje instanci `ListResultsCommand`. Jediná činnost metody setup spočívá ve vyrobení instance rozhraní `Database`. Tuto instanci předá konstruktoru třídy `ListResultsCommand`. Vyrobenou instanci třídy `ListResultsCommand` vrací.

### 5.1.3 Command třídy

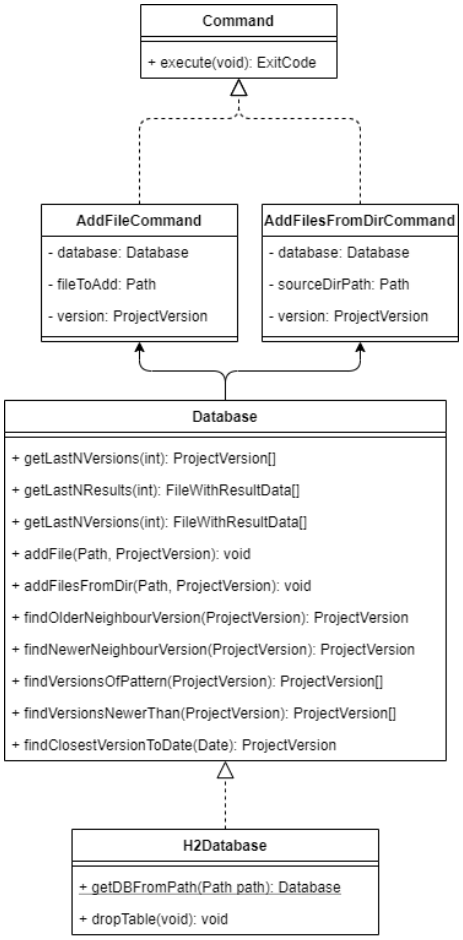
V této podkapitole je popsaný význam a práce jednotlivých implementací rozhraní Command. Implementace třídy Command musejí mít implementovanou jedinou metodu `execute`.

#### AddFileCommands

Uživatel musí každý výsledek měření výkonu zaznamenat do systému PerfEval. Pokud výsledek měření nebude zaznamenán, tak s ním PerfEval nebude vůbec pracovat. Přidávat je možné soubory samostatně, nebo z adresáře. Při přidávání souborů z adresáře budou přidány všechny soubory, a to i soubory zanořené ve vnitřních adresářích zadaného adresáře.

Na obrázku je vidět, že za oběma příkazy pro přidávání výsledků měření stojí jedna implementace rozhraní Database. Jediná existující implementace tohoto rozhraní využívá technologie H2 databáze. Jedná se o technologii, která umožňuje vést si databázi lokálně v rámci souboru a pokládat na ní klasické SQL dotazy.

Implementace rozhraní Database pomocí technologie H2 je třída H2Database. V rámci databáze pro jeden projekt, který PerfEval spravuje, jsou lokální soubory




Obrázek 5.5: Objektový návrh části PerfEvalu pro přidávání výsledků měření

H2 databáze uloženy v adresáři .performance. Databáze pro správu dat o výsledcích měření má jen jednu tabulku. V této tabulce jsou informace o cestě k souboru a informace o verzi.

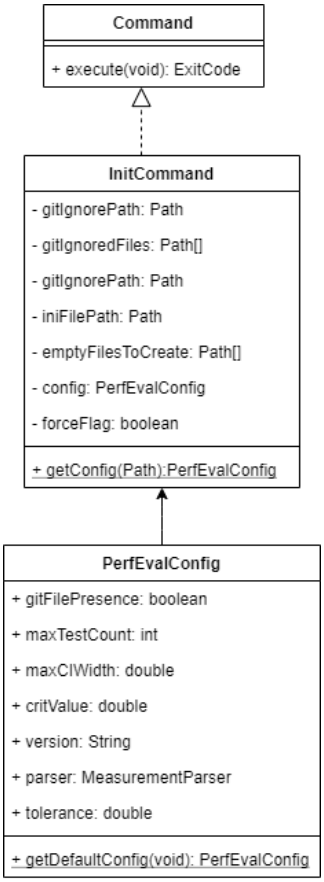
### InitCommand

Před použitím systému PerfEval k vyhodnocování je nutné jej inicializovat. Účelem inicializace je vytvoření konfiguračního souboru a databáze pro ukládání informací o výsledcích měření. Tyto soubory jsou potřebné pro fungování systému PerfEval v rámci vybraného projektu.

Kroky inicializace jsou následující:

1. Zjištění zdali adresář podléhá správě verzí systému Git
2. Vytvoření adresáře .performance
3. Vytvoření konfiguračního souboru config.ini s výchozími hodnotami
4. Vytvoření databáze pro ukládání výsledků měření

Při každém spouštění dalšího příkazu systému PerfEval je pak načtena konfigurace z konfiguračního souboru. Při vyhodnocování výkonu se načítají výsledky




Obrázek 5.6: Objektový návrh části PerfEvalu pro inicializaci

měření společně s metadaty o verzi, ke které se měření vztahuje. Tato metadata jsou uložena ve zmíněné databázi.

Pomocí těchto kroků je systém PerfEval inicializován pro další použití. Na obrázku 5.5 je vidět objektový návrh části systému PerfEval, která se stará o inicializaci systému. Hlavní třídou je třída `InitCommand`, která provádí výše zmíněné kroky inicializace v rámci metody `execute`. Protože třída inicializace se provádí pomocí příkazu `init`, tak je třída `InitCommand` implementací rozhraní `Command`. Třída `PerfEvalConfig` reprezentuje obsah konfiguračního souboru.

### EvaluateCLICommand

Třída `EvaluateCLICommand` má za úkol porovnat výsledky měření výkonu dvou různých verzí. Soubory s výsledky dvou verzí a parser k nim jsou dodané při konstrukci. Při konstrukci je dodaná implementace rozhraní `StatisticTest`. Zmíněná instance slouží ke statistickému porovnání výsledků měření. Při konstrukci je dále dodaná implementace rozhraní `ResultPrinter`. Implementace tohoto rozhraní rozhoduje o tom, jakým způsobem budou prezentovány výsledky porovnání. Třída v rámci volání metody `execute` nechá naparsovat soubory s výsledky měření do objektů typu `Samples`. Seznamy těchto objektů nechá porovnat pomocí statistického testu. Výsledkem je objekt MeasurementComparisonResultCollection. Tento objekt je nakonec pomocí dodané implementace rozhraní `ResultPrinter` prezentováno uživateli.

Třída `ListResultsCommand` a její metoda `execute` slouží k prostému vypsání

obsahu databáze s výsledky měření. Databáze obsahuje informace o výsledcích měření jako je cesta k souboru a popis verze. Tyto údaje vypíše metoda print na třídě FileInfoPrinter do přehledné tabulky na standardní výstup.

### 5.1.4 Implementace rozhraní MeasurementParser

Rozhraní `MeasurementParser` je určeno ke zpracování souborů s výsledky měření výkonu. Metoda `getParser` třídy `MeasurementFactory` vrací správnou instanci `MeasurementParser`u na základě dodaného jména (řetězce). Jméno parseru je uloženo v konfiguračním souboru.

Implementace rozhraní mohou při zpracovávání souborů vyhazovat runtime výjimku `MeasurementParserException`. Výjimka je odchytávaná mimo implementaci `MeasurementParser` v metodě `execute` třídy `EvaluateCLICommand`. Výjimku je tedy bezpečné vyhazovat.

Aktuálně dostupné implementace rozhraní `MeasurementParser` jsou pouze dvě. `JMHJSONParser`, který umí zpracovávat výstup frameworku JMH ve formátu JSON, a `BenchmarkDotNetJSONParser`, který umí zpracovávat výstup frameworku BenchmarkDotNet ve formátu JSON.

### 5.1.5 Implementace rozhraní StatisticTest

Rozhraní `StatisticTest` definuje, jaké metody mají mít implementace statistických testů pro porovnání dvourozměrných polí typu double. Jedná se o metodu, která vrací interval spolehlivosti. Interval spolehlivosti je intervalový odhad střední hodnoty pro rozdíl dvou náhodných veličin. Dále musí umět vrátit odhad minimálního počtu vzorků, které jsou zapotřebí pro zajištění dostatečně úzkého intervalu.

Implementace tohoto rozhraní jsou dvě. První je třída `ParametricTest`, která používá Welchův t-test, který je podrobněji popsán v kapitole 2.3.1. Druhou implementací je `NonparametricTest`, který využívá hierarchického bootstrapu. Tento bootstrap je podrobněji popsaný v kapitole 2.3.2.

### 5.1.6 Implementace rozhraní Database

Rozhraní `Database` definuje funkce, které jsou požadovány od systému, který má ukládat metadata o výsledcích měření. Jeho jediná implementace využívá technologie H2 embedded databáze. Technologie umožňuje vyhledávat položky v lokální databázi pomocí standardních SQL příkazů.

## 5.2 Rozšiřitelnost a její omezení

Systém PerfEval byl od počátku projektován tak, aby byl rozšiřitelný. V různých částech návrhu se vyskytují místa, kde je možné významným způsobem doplnit a změnit chování celé aplikace.

Nejdůležitější ze zmiňovaných rozšíření je rozšíření o datový formát. Tato možnost dělá z PerfEvalu poměrně univerzální nástroj. Činí ho totiž méně závislým na použitém měřícím systému a jeho výstupním formátu.

### 5.2.1 Rozšíření o datový formát

Vezměme opět scénář našeho programátora z první kapitoly. Programátor si napsal výkonnostní testy. Testy napsal pomocí nástroje, který nepodporuje PerfEval. Nicméně programátor ví, že systém PerfEval dělá přesně to, co potřebuje. Jediný problém je tedy ve vysvětlení svého datového formátu systému.

PerfEval je od počátku zamýšlen pro rozšíření v tomto místě. Programátor tedy musí prozkoumat, jak správně zkonstruovat třídu `Samples`. Třída `Samples` obsahuje všechny hodnoty naměřené pro jednu testovanou metodu a instanci třídy `Metric`. Instance třídy `Metric` reprezentuje fyzikální jednotku a příznak jestli vyšší hodnota znamená vyšší výkon.

Programátor musí implementovat rozhraní `MeasurementParser`. V tomto rozhraní je důležitá metoda `getTestsFromFiles`. Tato metoda na vstupu přijme všechny soubory s výsledky měření jedné verze. Výstupem je list objektů typů `Samples`. Pro každou metodu (test výkonu), který se v souborech nachází, se v listu vyskytuje pouze jedna instance typu `Samples`.

Poslední krok tohoto rozšíření po implementování `MeasurementParser`u je jeho registrace. Registrace probíhá tak, že se přidá jeden řádek do statického konstruktoru třídy `ParserFactory`. Na řádku bude přidání položky do objektu s názvem `registeredParsers`. Tento objekt je typu `HashMap` a přiřazuje k sobě `Supplier`, který vrací `MeasurementParser`, a název typu `String`. Přidávaná položka je tedy `String` odpovídající názvu parseru a reference na metodu, která umí parser zkonstruovat.

Použití nového parseru je pak jednoduché. Při inicializaci systému PerfEval příkazem init se jako argumentu benchmark-parser použije jméno nového parseru. Dokonce jej začne hlásit v nabídce i chybová hláška v případě, že žádný parser není při inicializaci zadán.

### 5.2.2 Rozšíření o komparátor

Pokud uživatel chce PerfEvalu změnit pořadí výpisu testů na výstupu, může použít komparátor. Pokud mu žádný z připravených nevyhovuje, tak může implementovat nový komparátor. Tento komparátor je objektem typu Comparator, který porovnává instance MeasurementComparisonRecord. Pomocí tohoto komparátoru, pak dojde k setřídění vypisovaných prvků.

Dále je nutné přepsat metodu resolvePrinterForEvaluateCommand takovým způsobem, aby reagovala i na nový druh komparátoru podle příkazové řádky. Tuto metodu je možné nalézt ve třídě `SetupUtilities`. Posledním krokem je přidání nové značky do parseru příkazové řádky v metodě createParser. Komparátor se pak může předávat objektům typu `ResultPrinter` při konstrukci. O jejich dalším použití si tedy tyto objekty rozhodují samy.

### 5.2.3 Rozšíření o statistický test

Může se stát, že uživatel systému PerfEval má o svých datech nějaké předpoklady, které by chtěl při vyhodnocování zohlednit. Může si tedy naprogramovat vlastní implementaci rozhraní `StatisticTest`, kde tyto předpoklady zohlední.

Po naprogramování vlastní implementace rozhraní `StatisticTest`, pak jen stačí ve třídě `SetupUtilities` změnit chování metody resolve`StatisticTest`

tak, aby rozpoznávala novou implementaci podle příkazové řádky. Posledním krokem je přidání nové značky do parseru příkazové řádky v metodě createParser.

### 5.2.4 Rozšíření o možnost výpisu

Pro vypisování výsledků porovnání slouží rozhraní `ResultPrinter`. Pokud by si uživatel chtěl implementovat vlastní způsob vypisování, tak stačí implementovat jedinou jeho metodu PrintResults.

Přidání nového `ResultPrinteru` je podobné jako rozšíření o statistický test. Je nutné změnit metodu `resolvePrinterForEvaluateCommand` tak, aby vracela i novou implementaci `ResultPrinteru`. Zmíněná metoda se nachází ve třídě `SetupUtilities`. Pokud by bylo zapotřebí nového argumentu na příkazové řádce, tak je nutné do metody `createParser` jeho použití také správně implementovat.

### 5.2.5 Rozšíření o novou HTML šablonu

Při použití přepínače `html-output` bude výsledek porovnání uložen v podobě HTML souboru. Tento soubor je možné otevřít v běžném internetovém prohlížeči. Přepínač je možné doplnit o argument `html-template`, kde bude uvedena cesta k nové šabloně. Dodaná šablona bude ve formátu, který podporuje šablonový procesor Thymeleaf. Šabloně bude dodaná instance objektu `MeasurementComparisonResultCollection` jehož struktura je zdokumentovaná v JavaDoc dokumentaci systému PerfEval.

### 5.2.6 Změna použitého databázového systému

V důsledku rozhodování se o tom, jak se budou informace o výsledcích měření ukládat, vzniklo rozhraní `Database`. Rozhraní má mnoho metod. Pokud by se uživatel rozhodl změnit způsob ukládání dat o měřeních, tak by musel implementovat celé toto rozhraní. Po implementaci rozhraní pak už jen stačí změnit metodu `constructDatabase` ve třídě `SetupUtilities`, která vrací instanci objektu typu `Database`.

### 5.2.7 Rozšíření o příkaz

Každý příkaz PerfEval se skládá ze dvou tříd. Jedná se o třídu implementující rozhraní `CommandSetup` a o třídu implementující rozhraní `Command`. Pokud by uživatel chtěl doplnit nějaký nový příkaz, který by v kontextu systému dával smysl, tak je to možné. Dobrý příklad pro reprezentování nového příkazu bude vyhodnocení výkonu s grafickým výstupem.

Na rozdíl od stávajícího vyhodnocování by grafické vyhodnocování potřebovalo údaje z více než dvou posledních verzí. Proto by příkaz evaluate-graphical metoda `getCommand` rozpoznala jako příkaz a vytvořila instanci nové setup třídy `EvaluateGraphicalSetup`. Na této třídě, implementaci `CommandSetup`, by pak zavolala metodu `setup`. Metoda setup na třídě `EvaluateGraphicalSetup` by měla za úkol z konfigurace PerfEvalu a příkazové řádky vyrobit instanci nové třídy `EvaluateGraphicalCommand`. Tuto instanci, která by implementovala rozhraní `Command`, by pak metoda `getCommand` na třídě `Parser` vrátila.

Zbytek programu by se již nezměnil a metoda `main` ve třídě `Main` by vykonala dodaný příkaz. Spustila by standardním způsobem metodu `execute` na instanci objektu typu Command.

Zaregistrování nového příkazu by probíhalo přidáním nové položky do statického konstruktoru třídy `Parser`. Položka mapy `commandPerSetup` má obdobnou strukturu jako v případě rozšíření o nový `MeasurementParser`. Dodal by se řádek s názvem příkazu typu `String` a z reference na bezparametrický konstruktor implementace rozhraní `CommandSetup`. Název příkazu odpovídá příkazu, kterým se bude volat z příkazové řádky.

### 5.2.8 Omezená rozšiřitelnost ve vyhodnocování

Jeden z nejhorších požadavků na rozšíření systému by bylo rozšíření v oblasti vyhodnocování. Jedná se o změnu implementace třídy `PerformanceEvaluator`. Tato třída utváří celkovou vyhodnocovací logiku. Její rozšiřitelnost je omezená na implementace rozhraní `StatisticTest`.

Omezenou změnu chování vyhodnocování výkonnostních testů provést lze. Omezené změny ve vyhodnocování se provádí změnami hodnot uvnitř config.ini souboru.

# 6. Vyhodnocení práce

Výsledkem této práce je implementace systému pro porovnávání výkonu verzí softwaru pojmenovaná PerfEval. V této kapitole bude krok za krokem ukázáno, jak se systémem pracovat.

## 6.1 Používání systému PerfEval

V následujících dvou ukázkách bude vysvětleno, jak používat PerfEval. První ukázka bude cílit na nastínění, co nejjednoduššího použití. Druhá část ukáže aplikaci PerfEvalu na skutečném projektu.

### 6.1.1 Jednoduché použití PerfEvalu

Následující kód popisuje obvyklou posloupnost příkazů práce se systémem PerfEval. Na začátku je nutné jej inicializovat. PerfEval bude inicializován pro použití JMHJSONParseru. Následně se přidají výsledky měření několika (alespoň dvou) různých verzí. Nakonec program vypíše, jestli výkonnostní testy prošly nebo ne.

```
#!/bin/bash
perfeval init --benchmark-parser JMHJSONParser
perfeval index-all-results --path tests/old_test --
   version old_version
perfeval index-all-results --path tests/new_test --
   version new_version
perfeval evaluate && echo "Performance␣test␣passed"
    | exit 0
echo "Performance␣test␣failed" | exit 1
```

### 6.1.2 Návrh skriptu pro doměření výsledků

Následující kód nastiňuje možnost využití příkazu list-undecided. Výstupem tohoto příkazu jsou dva tabulátorem oddělené sloupce. V prvním sloupci jsou názvy metod, pro něž systém eviduje málo naměřených běhů. Ve druhém sloupci je uveden tento počet. Příkaz je určený pro skriptování, proto není dodaná žádná další hlavička.

V případě, že výstupem není žádný výpis, tak je hodnot u všech testovacích metod naměřeno dostatek. Druhou alternativou je, že v důsledku nastavení v konfiguračním souboru systém vyhodnotil, že není možné požadovaný počet testů doměřit. Následné vyhodnocení pak bude vyžadovat kontrolu uživatelem, protože systém PerfEval bude vyhodnocení považovat za nevyhovující.

Skript projde všechny řádky výpisu. Pokud je výpis prázdný, tak skončí. V následujícím kódu je celá situace velmi zjednodušena. Nalezne se maximální počet testů, který je zapotřebí změřit. Pro tento maximální počet se změří výkony obou

verzí znovu. Výsledky těchto měření se zaznamenají do systému PerfEval. Po doběhnutí všech měření skript skončí. Vyhodnotí mezi sebou výsledky těchto verzí a skončí. Parametry $1 a $2 jsou stará a nová verze k porovnání. Předpokládá se, že příkaz measure provede měření verze zadané jako první argument a výsledek uloží do souboru specifikovaného jako druhý argument.

```
#!/bin/bash

index=1
while true; do
    output=$(perfeval list-undecided --old-version
       "$1" --new-version "$2")
    if [[ -n "$output" ]]; then
        max=$(echo "$output" | cut -f2 -d$'\t' |
           sort -n -k | head 1)
        for ((i=1; i<=max; i++)); do
            result_file="old_version_$index"
            measure "$1" "$result_file"
            perfeval index-new-result --path "
               $result_file" --version "$1"

            result_file="new_version_$index"
            measure "$2" "$result_file"
            perfeval index-new-result --path "
               $result_file" --version "$2"
            ((index++))
        done
    else
        perfeval evaluate --old-version "$1" --new-
           version "$2"
        exit $?
    fi
done
```

### 6.1.3 Použití PerfEvalu v průběžné integraci

Systém PerfEval je možné po vzoru unit testů používat při průběžné integraci. V příkladu níže je uvedeno, jak může vypadat soubor `.gitlab-ci.yml` pro GitLab CI. Uvedený vzor ilustruje, jak dodat výsledky měření do systému PerfEval a jak následně spustit vyhodnocení výsledků měření dvou verzí.

Skripty `measure_old_version.sh` a `measure_new_version.sh` v uvedeném příkladu simulují měření výkonu, tak že přečtou výsledky z předem připravených souborů. Tyto připravené soubory jsou výsledky měření, které byly získány spuštěním výkonnostních testů projektu Crate Lutz a kol. (2024).

Výsledek porovnání je zde uložen do souboru `junit-report.xml`. Tento soubor je ve formátu JUNIT XML. Tento formát je běžně přijímaný nástrojem Git v rámci průběžné integrace.

```
image: openjdk:21-jdk

stages:
    - test

variables:
    JUNIT_REPORT_PATH: junit-report.xml

build:
    stage: test
    script:
        - bash measure_old_version.sh >
          old_measurement.json
        - bash measure_new_version.sh >
          new_measurement.json
        - bash perfeval.sh init --benchmark-parser
          JMHJSONParser
        - bash perfeval.sh index-new-result --
          version 1.0 --path ./old_measurement.
          json
        - bash perfeval.sh index-new-result --
          version 2.0 --path ./new_measurement.
          json
        - bash perfeval.sh evaluate --t-test --
          junit-xml-output > $JUNIT_REPORT_PATH

    artifacts:
        when: always
        paths:
        - $JUNIT_REPORT_PATH
        reports:
        junit: $JUNIT_REPORT_PATH
        expire_in: 1 week
```

## 6.2 Nasazení systému v praxi

V této kapitole bude ukázáno, jak systém PerfEval funguje při svém nasazení. Pro ukázku byl vybrán projekt Crate Lutz a kol. (2024). Jedná se o databázový projekt volně dostupný na platformě GitHub.

Projekt Crate byl vybrán, protože je volně dostupný, a protože má implementované výkonnostní testy. Tyto testy je možné spouštět samostatně přímo z adresáře projektu a to i pro starší verze.

Systém PerfEval tedy bude použit pro porovnání vybraných commitů. Cílem tohoto spouštění je zjistit, jestli systém PerfEval detekuje zhoršení a případně zlepšení výkonu.

### 6.2.1 Výběr commitů

Commity pro prezentování práce systému byly vybírány z období od 21. dubna 2020 do 11. května 2023. Byly vybrány ty commity, jejichž commit message obsahuje slovo „performance“ a jejich sousední commity (pro porovnání). Commity byly vybírány z hlavní větve projektu. Všechny výkonnostní testy byly prováděné na strojích stejného druhu a konfigurace.

Z takto vybraných commitů byly dále vybrané jen ty, u kterých bylo možné projekt bez problémů sestavit. Zároveň bylo také nutné, aby bylo možné sestavit a spustit výkonnostní testy. Nakonec tedy bylo vybráno a naměřeno celkem 16 commitů z uvedeného období. Pro každý z těchto commitů, který reprezentuje v doméně systému PerfEval verzi, bylo měření provedeno celkem osmkrát.

### 6.2.2 Výsledky porovnávání přímých sousedů

Nástroj PerfEval byl spouštěn pro porovnání výsledků měření z předchozí podkapitoly. Výsledky byly zaznamenány do databáze PerfEvalu. Ten byl předtím inicializován pro použití JMHJSONParseru. Výsledky byly zaznamenány do databáze a následně vyhodnocovány. Vyhodnocování probíhalo s použitím statistické metody bootstrap a filtrem `-test-id`, který setřídil výsledky podle názvů metod.

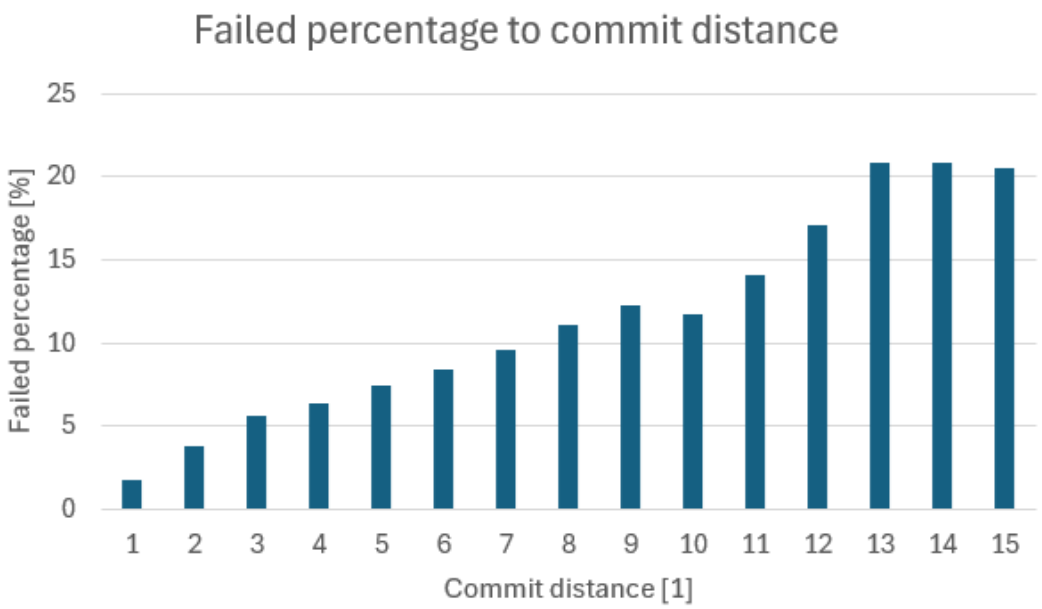
Při vyhodnocování přímých sousedů nedošlo k zaznamenání žádného zhoršení výkonu. Přímí sousedé byly vybráni tak, že se nalezl commit s commit message, která obsahovala slovo „performance“ a jejich přímý soused byl commit, který bezprostředně následoval, nebo předcházel tomuto. Pokud by tedy PerfEval byl součástí průběžné integrace při integrování takto vybraných commitů, tak by nebylo zaznamenáno žádné zhoršení výkonu, a tedy by bylo možné pokračovat v integraci.

Ani v jednom z těchto případů nebylo zaznamenáno ani zlepšení výkonu na které commit message poukazovaly. Je tedy nutné vyřešit otázku jestli PerfEval je schopen vůbec nějaké změny výkonu detekovat. Pro tento účel byly porovnány také bezprostředně nesousedící commity. Změny výkonu se totiž mohou kumulovat skrze více commitů. Pokles výkonu se tedy nemusí projevit u commitu, který změnu způsobil, ale až u commitu, který tuto změnu dostatečně zesiluje. Pokud by PerfEval detekoval změny na větší vzdálenost mezi commity, tak by to znamenalo, že PerfEval je schopen detekovat změny výkonu. Dále to znamená, že commit message uvedené u vyhledaných commitů nereferují na změnu výkonu provedenou v daném commitu.

### 6.2.3 Výsledky porovnávání nesousedících commitů

Protože PerfEval nezaznamenal žádné zhoršení výkonu u přímých sousedů, tak byly porovnány také přímo nesousedící commity. Došlo tedy k porovnání každého ze 16 commitů (vybraných v kapitole 6.2.1) s každým z těchto 16 commitů, který byl novější. Nejstarší commit byl tedy porovnán s 15 novějšími a nejmladší commit nebyl porovnán s žádným jiným novějším.

PerfEval porovnává pomocí příkazu `evaluate` právě dvě verze (novou a starou). Jako nová verze byla volena mladší ze dvou porovnávaných verzí. Jako stará byla volena starší ze dvou porovnávaných verzí. Celkem tedy bylo na 16 commitech provedeno 120 takových porovnání.




Obrázek 6.1: Graf počtu testů, které selhaly, vůči vzdálenosti commitů v čase

Při těchto porovnáních PerfEval ve svém výchozím nastavení již byl schopen detekovat zhoršení výkonu. Ačkoli všechny vybírané commity měly referovat ke zlepšením výkonu, tak PerfEval ukázal, že u některých výkonnostních testů došlo ke zhoršení výkonu. Ukazuje se, že čím jsou commity vzdálenější od sebe ve vývoji softwaru, tím se zvyšuje počet testů, které selhávají. Toto pozorování je vidět na obrázku grafu 6.1. Procentuální neúspěšnost byla spočtena jako podíl označených zhoršení výkonu vůči celkovému počtu společných testů, které mají obě verze.

K tomuto postupnému růstu poměru selhávajících testů dochází, protože se zhoršení výkonu může kumulovat skrze více commitů. Pokud se tedy měřená metoda zhoršuje v průběhu času, tak se to projeví až u commitu, který tuto metodu dostatečně zesiluje. Po dostatečném zesílení tohoto zhoršení se metoda zhorší natolik, že ji porovnání se vzdálenou předchozí verzí detekuje, ačkoli porovnání s přímým sousedem by ji nedetekoval.

Ukazuje se tedy, že pomocí PerfEvalu je sice možné detekovat zhoršení výkonu v rámci průběžné integrace. Není ale vhodné jej používat jako blokující faktor při integraci. PerfEval by měl být spíše používán jako nástroj pro detekci zhoršení výkonu v průběhu vývoje. V případě, že PerfEval detekuje zhoršení výkonu, tak je uživatel schopen jednoduše zjistit, kde ke zhoršení došlo. Pokud by PerfEval byl součástí průběžné integrace jako blokující faktor, tak by se mohlo stát, že by integrace byla zablokovaná kvůli zhoršení výkonu, který ve skutečnosti z větší části způsobil jiný commit.

V grafu na obrázku 6.1 je vidět, že se zde nachází i několik neúspěšných testů u commitů se vzdáleností 1. Tato porovnání byla provedena tak, že se z vybraných 16 commitů vždy vybral commit a jeho soused v tomto výběru a ne pouze bezprostřední soused. Proto i když bylo výše uvedeno, že PerfEval nezaznamenal žádné zhoršení, tak na grafu je vidět, že některé testy selhaly. Tato selhání byla způsobena tím, že se výkonnostní testy se ve skutečnosti porovnávaly

i s commity, které byly vzdáleny o víc než jeden commit. To je způsobeno tím, že výběr commitů byl proveden tak, že byly vybrány commity, které obsahovaly slovo „performance“ v commit message a jejich sousední commity. Celý tento výběr však nejsou bezprostředně navazující commity.

# 7. Závěr

Tato práce se snaží navázat na využívání unit testů při vývoji softwaru. Nástroj PerfEval, který byl vytvořen, má za úkol poskytnout vývojářům možnost psát výkonnostní testy obdobně jako unit testy.

V první kapitole bylo vysvětleno, co je průběžná integrace. Dále byl popsán scénář programátora, který ji využívá pro udržování korektnosti svého kódu, a jak by ji chtěl využít i pro udržování výkonu. Programátor měl napsané výkonnostní testy pomocí benchmarkovacího frameworku a potřeboval je vyhodnocovat. Neměl ale žádný nástroj, který by uměl výsledky měření porovnat a vyhodnotit změnu výkonu.

Druhá kapitola se věnuje analýze problematiky kolem vyhodnocování výkonu softwaru. Je zde popsáno jak vypadají výstupy benchmarkovacích frameworků JMH a BenchmarkDotNET. Dále jsou popsány statistické metody, které se následně v nástroji využívají k porovnání výsledků měření.

Ve třetí kapitole je vysvětleno několik důležitých rozhodnutí, které byly provedeny před a v průběhu vývoje nástroje PerfEval. Zároveň byla nastíněna architektura nástroje a jakým způsobem se používá.

Uživatelská dokumentace ukazuje jaké příkazy má uživatel k dispozici včetně jejich argumentů. V této kapitole se také uživatel dozví jak nástroj instalovat a používat včetně toho, jak jej konfigurovat.

Programátorská dokumentace obsahuje detailní popis struktury kódu. Je zde popsáno jaké byly využité návrhové vzory a jak spolu jednotlivé třídy vzájemně interagují.

Výsledkem práce je nástroj PerfEval jehož způsob použití včetně nasazení v rámci existujícího projektu je demonstrováno v poslední kapitole. Nástroj byl nasazen v rámci projektu Crate a byl použit k analýze výkonnostních testů verzí, které byly označené, jako verze se změnou výkonu.

Ukázalo se, že vyvinutý nástroj PerfEval je schopen detekovat změny výkonu. Při použití v rámci průběžné integrace je vhodné dodávat verzi, která bude považovaná za referenční. Tato verze by měla mít výkon, který chce uživatel udržet. V případě, že se výkon změní, tak PerfEval uživatele upozorní. Zároveň tento nástroj splňuje požadavek, že se má připodobnit k používání unit testů.

Tento nástroj tedy řeší původní programátorův problém chybějícího nástroje pro vyhodnocování výkonu. Dodáním řádku do svého měřícího skriptu programátor předá informace o výsledcích měření nástroji PerfEval. Následně programátor jedním příkazem vyhodnotí změnu mezi požadovanými verzemi.

Do budoucna by bylo možné nástroj rozšířit například o vyhodnocování výkonu na bázi skóre. Optimalizace výkonu totiž obvykle bývá taková, že zlepšením výkonu jednoho testu se zhorší výkon jiného. Proto by se jednotlivým testům mohlo přiřazovat skóre, které by bylo závislé na změně výkonu a na tom, jak moc je test důležitý.

# Seznam použité literatury


Akinshin, A. (2024). dotnet/BenchmarkDotNet. URL https://github.com/dotnet/BenchmarkDotNet. original-date: 2013-08-18T06:17:48Z.

Gamma, E., Helm, R., Johnson, R. a Vlissides, J. Design Patterns : Elements of Reusable Object-Oriented Software.

Heisler, B. (2024). criterion - Rust. URL https://docs.rs/criterion/latest/criterion/index.html.

Lutz, C., Dorn, B. a Batlogg, J. (2024). crate. URL https://github.com/crate/crate. original-date: 2013-04-10T09:17:16Z.

Shipilëv, A. (2024). openjdk/jmh. URL https://github.com/openjdk/jmh. original-date: 2019-05-15T06:47:19Z.

Stefan, P., Horký, V., Bulej, L. a Tůma, P. (2017). Unit testing performance in java projects: Are we there yet? In *Proc. 8th ACM/SPEC Intl. Conf. on Performance Engineering (ICPE)*, pages 401–412. ISBN 978-1-4503-4404-3. doi: 10.1145/3030207.3030226.

Turčicová, M. (2021). Dvouvýběrové testy. URL https://www.karlin.mff.cuni.cz/~turcic/Dvouvyberove_testy.pdf.

Wikipedia contributors (2023). Welch's t-test — Wikipedia, the free encyclopedia. URL https://en.wikipedia.org/w/index.php?title=Welch%27s_t-test&oldid=1184251732. [Online; accessed 28-March-2024].

Šámal, R. (2023). NMAI059 Pravděpodobnost a statistika 1. URL https://iuuk.mff.cuni.cz/~samal/vyuka/2223/PSt1/skripta.pdf.

# Seznam obrázků

# A. Přílohy

## A.1 První příloha