FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY

Bakalářská práce

Uživatelské rozhraní pro analyzátor paměti

Štěpán Faragula

PLZEŇ

2024

FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY

Bakalářská práce

Uživatelské rozhraní pro analyzátor paměti

Štěpán Faragula

Vedoucí práce
Ing. Richard Lipka, Ph.D.

PLZEŇ **2024**

Citace v seznamu literatury:
FARAGULA, Štěpán. *Uživatelské rozhraní pro analyzátor paměti*. Plzeň, 2024. Bakalářská práce. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky. Vedoucí práce Ing. Richard Lipka, Ph.D.

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2023/2024

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Štěpán FARAGULA**
Osobní číslo:	**A21B0119P**
Studijní program:	**B0613A140015 Informatika a výpočetní technika**
Specializace:	**Informatika**
Téma práce:	**Uživatelské rozhraní pro analyzátor paměti**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se s existujícím nástrojem pro analýzu alokací paměti v Java programech.
2. Seznamte se s metodami a postupy vizualizace různých metrik nad zdrojovými texty programu.
3. Navrhněte vizualizační metodu pro zobrazení alokací paměti a jejich duplicit.
4. Implementujte prototyp navrženého řešení.
5. Otestujte vytvořenou implementaci s ohledem na uživatelskou přívětivost a přehlednost.

Rozsah bakalářské práce: **doporuč. 30 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

Dodá vedoucí bakalářské práce.

Vedoucí bakalářské práce: **Ing. Richard Lipka, Ph.D.**
Katedra informatiky a výpočetní techniky

Datum zadání bakalářské práce: **2. října 2023**
Termín odevzdání bakalářské práce: **2. května 2024**

L.S.

Doc. Ing. Miloš Železný, Ph.D.
děkan

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 25. října 2023

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného akademického 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 Západočeská univerzita v Plzni má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Plzni dne 29. dubna 2024

. . . . . .
Štěpán Faragula

V textu jsou použity názvy produktů, technologií, služeb, aplikací, společností apod., které mohou být ochrannými známkami nebo registrovanými ochrannými známkami příslušných vlastníků.

Abstrakt

V této bakalářské práci je popsán návrh a implementace vizualizace metrik zdrojového kódu, které jsou poskytovány již existujícím analyzátorem paměti Java programů. Konkrétně se jedná o data detekovaných alokací a jejich duplicit nahromaděných za běhu pozorované aplikace. K vyřešení této úlohy vzniklo rozšíření textového editoru Visual Studio Code, které nabízí statický pohled na získaná dynamická data. Výsledkem je funkční prototyp navržené vizualizace, který je možné použít za účelem průběžné analýzy metrik zdrojového kódu.

Abstract

This bachelor's thesis describes the proposal and implementation of a visualization method capable of displaying source code metrics generated by an established memory analyzer for Java programs. More specifically, these metrics relate to identified allocations and duplications that were detected during the execution of the examined application. To achieve this, an extension was created for the Visual Studio Code text editor, which offers a static view of the collected dynamic data. The result is a functional prototype that implements proposed visualization technique, which is suitable for a continuous analysis of the mentioned source code metrics.

Klíčová slova

Java • analyzátor paměti • alokace paměti • vizualizace • plugin • textový editor • Visual Studio Code

Poděkování

Chtěl bych poděkovat vedoucímu práce Ing. Richardu Lipkovi, Ph.D. za metodické vedení a dobře míněné rady. Dále bych rád poděkoval své přítelkyni, která mě podporovala při tvorbě bakalářské práce a také při studiu. Děkuji své rodině, která mi byla oporou po celou dobu studia a poskytovala mi motivaci dále se vzdělávat.

Obsah

Úvod 1

Vývoj softwaru je dlouhý a náročný proces, který vyžaduje schopnost umět zacházet s dostupnými zdroji pro splnění všech požadavků specifikace. K jejich řešení nemusí být vždy zvolen nejefektivnější algoritmus, ať už z hlediska stability, časové odezvy nebo využití systémových prostředků. Do určité míry lze tyto nedokonalosti ignorovat, případně, pokud to specifikace dovoluje, obejít navýšením zdrojů. Jestliže zkušený programátor usoudí, že problematický úsek kódu je zapotřebí zefektivnit (např. kvůli zbytečnému plýtvání pamětí), může současný postup buď zcela nahradit za jiný, nebo ho ponechat a optimalizovat.

Účelem optimalizace je modifikovat zdrojový kód tak, aby poskytoval stejnou funkcionalitu a zároveň se snížily jeho nároky na využívané prostředky. V praxi je běžné, že při tomto procesu si vývojáři pomáhají analytickými nástroji, které umí rychle najít či nastínit místo, kde se nachází neuspokojivá část programu. Kořenovou příčinu problému, kterou by programátor hledal hodiny, lze tak snadno odhalit jedním spuštěním vhodně zvoleného a spolehlivého nástroje.

Různé parametry mohou být sbírány už za běhu sledované aplikace a po získání všech hodnot typicky následuje jejich shrnutí. Prezentace těchto dat se u různých analyzátorů liší, což je zejména dáno charakterem metrik, které představují. Může se jednat o pouhý textový soubor nebo vizualizace s interaktivním ovládáním. Hlavním záměrem analytických nástrojů je předat uživateli zpracované údaje tak, aby pro něj byly snadno pochopitelné a pomohli mu s řešením určitého problému.

Cílem této práce je navrhnout a implementovat vhodnou vizualizační metodu dat existujícího nástroje pro analýzu paměti Java programů. Analyzátor by měl poskytnout různé informace ohledně alokací a duplicit vytvořených za běhu pozorované aplikace. Popisovaný software vznikl jako praktická část bakalářské práce na Fakultě aplikovaných věd Západočeské univerzity v Plzni v roce 2023 [1].

Tento text lze rozdělit na tři hlavní části. V první je představen existující nástroj pro analýzu paměti, na který tato práce navazuje. Druhá část se věnuje teoretickému pojetí vizualizací spolu s jejich uplatněním v oblasti zobrazování metrik zdrojového kódu. Poslední část se věnuje návrhu a popisu realizace vlastní vizualizační metody, která zobrazuje informace o alokacích a duplicitách v Java programech.

Seznámení s existujícím nástrojem

2

Poznatky, které jsou popsány v celé této kapitole, vychází z textu bakalářské práce *Monitorování alokací paměti za běhu Java aplikací* [1] a zdrojového kódu její praktické části. Software je veřejně dostupný na adrese školního repozitáře platformy GitLab[1], která se načte po naskenování QR kódu na obrázku 2.1. Původní práci je možné dohledat pod příspěvkem *second revision*, který byl přidán 27. 4. 2023.

Nástroj, na který bude tato práce navazovat, slouží k profilování přidělené paměti v Java programech za jejich běhu. Analyzátor poskytuje data o velikosti detekovaných alokací v bytech, spolu s místem jejich vzniku, které je trasováno až na konkrétní řádek ve zdrojovém textu. Dále prohledává všechny instance identifikované tímto způsobem a pokouší se mezi nimi najít duplicitní hodnoty. To provádí rekurzivním porovnáváním všech atributů do hloubky. Zmíněné operace jsou vykonány na základě instrumentace, která umožňuje číst a upravovat bajtkód zkoumaného softwaru ještě před jeho vykonáním.

Analyzátor je napsán v programovacím jazyce Java a jeho následné sestavení zařizuje systém Apache Maven. Spouští se spolu s pozorovanou aplikací, kde pro úspěšné připojení analyzátoru na jiný Java program využívá speciální parametry `--add-opens` a `-javaagent:<agent.jar>`. Více informací je uvedeno v uživatelské příručce, která je součástí práce [1]. Výstupem tohoto nástroje je textový soubor `allocations.log`, do kterého jsou přidávány záznamy knihovnou Log4j 2.

Obrázek 2.1: QR kód směřující na repozitář analyzátoru

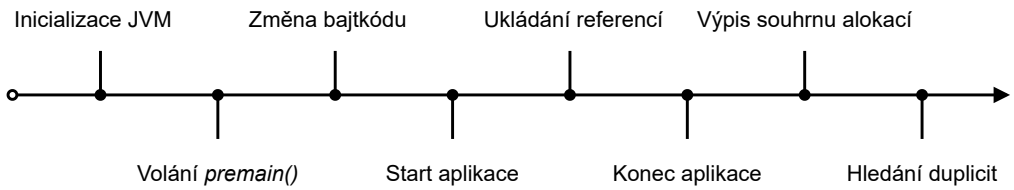
[1] https://gitlab.kiv.zcu.cz/lipka/java-memory-allocation-analyser

2.1 Podrobnosti k instrumentaci

Jak již bylo zmíněno, nástroj detekuje přidělenou paměť a vyhledává duplicity za běhu pozorované aplikace instrumentací. Aby se mohl takovýmto způsobem připojit na jiný proces, má definovanou speciální třídu, tzv. agenta. Tato třída pochází z balíku `java.lang.instrument` a disponuje metodou `premain()`, která je volána okamžitě po inicializaci Java Virtual Machine (JVM) [2]. Agent má tak prostor na upravení bajtkódu připojeného Java programu ještě před jeho vykonáním. Modifikace jsou prováděny tak, aby druhý program volal připravené profilovací metody analyzátoru po provedení určitých instrukcí a také po ukončení své činnosti. Jakmile agent dokončí veškeré úpravy, spustí se hlavní metoda pozorovaného softwaru.

Změnu bajtkódu řeší třída implementující rozhraní `ClassFileTransformer` metodou `transform()`. Analytický nástroj nutí připojenou aplikaci po každém vykonání jedné ze čtyř instrukcí `new`, `newarray`, `anewarray` nebo `multianewarray` volat metodu, která předá analyzátoru odkaz na právě vytvořenou instanci. S touto referencí je uchována i velikost přidělené paměti určená statickou metodou `getObjectSize()` spolu se stopou záznamu `StackTraceElement`, ze které je možné zjistit konkrétní místo vzniku instance ve zdrojovém textu. Všechny údaje se postupně ukládají do připravené kolekce `HashMap` inicializované ve třídě `AllocationDetector`. Ta slouží k uchování seznamu referencí pro budoucí porovnávání duplicit. Dále je záznam předán třídě `AllocationCounter` obsahující tři mapy, kde každá z nich je použita k agregaci vypočítané velikosti podle konkrétního řádku, metody nebo třídy. Seskupení alokací je prováděno okamžitě po předání reference na instanci tak, že jsou systematicky sčítány podle unikátního řetězcového klíče v mapě. Záznamy se tímto způsobem postupně hromadí na haldě, dokud pozorovaná aplikace nedokončí svou činnost.

Po ukončení běhu sledovaného programu jsou volány ještě dvě poslední metody, kde první z nich vypíše shrnutí údajů o zabrané paměti do souboru a druhá vyhledá a zapíše počet všech stejných instancí mezi uloženými referencemi. Hledání duplicitních hodnot probíhá rekurzivním porovnáváním všech atributů referovaných instancí, kde odkazy na kandidáty jsou uloženy v záznamu kolekce `HashMap`. Na obrázku 2.2 je vidět celý proces profilování prováděný analyzátorem.



Obrázek 2.2: Časová osa prováděných operací analyzátoru paměti

2.2 Popis výstupních dat

Výstupem analyzátoru je textový soubor, který je tvořen slovním popisem alokovaných instancí a jejich duplicit. Zpravidla se člení do dvou skupin – `TRACE` a `INFO`. Ukázka struktury tohoto dokumentu je vidět ve výpisu 2.1.

Výpis 2.1: Ukázkový výstup nástroje

```
2024-03-04 11:18:36,927 TRACE  [main] Object allocated at:
    Main.java:50
2024-03-04 11:18:36,943 TRACE  [main] Object size: 16
2024-03-04 11:18:36,943 TRACE  [main] Object type:
    TestingObject
2024-03-04 11:18:37,021 INFO  [main] Main.java:50 allocated
    16 bytes
2024-03-04 11:18:37,037 INFO  [main] Found 2 duplicates of
    object: main.TestingObject first allocated at:
    Main.java:39
```

První část se vypisuje postupně za běhu sledované aplikace, přičemž se záznamy přidávají okamžitě po detekci nové instance. Ve výpisu ji lze poznat podle značky `TRACE`. Každá položka je v tomto úseku tvořena trojicí řádků, které postupně poskytují informace o místě vzniku, velikosti a názvu třídy vytvořené instance.

Druhá část, označená výrazem `INFO`, se vypíše až po ukončení činnosti pozorovaného programu, tedy při provádění posledních dvou metod nástroje. Jsou zde vypsány agregované údaje přidělené paměti podle řádků, metod a tříd v tomto pořadí. Následující blok záznamů obsahuje informace o počtu každé z nalezených duplicit a místem vzniku první z nich vytvořené. V kontextu analyzátoru jsou kopie chápány jako množství identických prvků bez originálu, tudíž hodnota 2 ve výstupu představuje nalezení tří instancí se shodnými atributy.

Co se týče samotného obsahu záznamů, části ze kterých se skládají jsou v některých případech zcela nadbytečné či příliš detailní. Například součástí každého řádku je časová stopa označující, kdy přesně byla položka zapsána do dokumentu. V první polovině má tento údaj význam, jelikož také zobrazuje čas přidělení paměti. Dále už však nepřináší žádnou přidanou hodnotu k pozorované aplikaci, protože pouze ohlašuje čas, kdy analyzátor dokončil provádění rekurzivního hledání duplicit po ukončení činnosti programu. Někde pro změnu chybí ve výstupu užitečné informace jako třeba počet detekovaných alokací určitého typu nebo místo vzniku všech nalezených kopií, a ne pouze první, kterou se podařilo identifikovat.

I přes uvedené nesrovnalosti nabízí současný formát výstupu dostatek informací jak pro uživatele, který dokáže vyvodit určitý závěr o přidělování paměti, tak pro testera, který může provést manuální ověření očekávané funkcionality tohoto nástroje. Současná struktura je tudíž vhodná pro běžné používání i testování analyzátoru.

2.3 Zjištěné nedostatky

V rámci zkoumání nástroje byly nalezeny některé nedostatky, které by mohly mít negativní vliv na budoucí implementaci vizualizace poskytovaných údajů. Dále v kapitole 5 jsou popsány provedené úpravy tohoto analytického nástroje řešící některé problémy spojené s výstupním souborem, které jsou zde uvedeny.

2.3.1 Návaznost výstupu u další aplikace

Ačkoliv pro člověka může být stávající formát výstupu srozumitelný a přehledný, způsob zápisu dat v celých větách obsahuje z pohledu stroje mnoho redundance. Mimo jiné, formulace vět také zahrnuje zbytečnou variabilitu, jelikož některé informace je možno vyvodit z kontextu téhož záznamu. Existují např. dva různé zápisy s principiálně stejným formátem, které vzájemně rozlišují alokaci jedné instance a pole několika objektů. Takový případ je vidět ve výpisu 2.2, kde lze snadno mezi záznamy určit, o jaký ze dvou druhů alokace se jedná na základě hranatých závorek na konci názvu třídy. Pokud jde o rozšiřitelnost a údržbu existujícího formátu, věty jsou fixně zakomponovány do zdrojového kódu, tzv. *hardcoded*, a program nenabízí mnoho možností snadného rozšíření či úprav stávajících zpráv.

Výpis 2.2: Variabilita výstupu – první tři řádky označují pole, zbylé jednu instanci

```
1 2024-03-05 15:27:09,541 TRACE  [main] Object array allocated
     at: SudokuMode.java:3
2 2024-03-05 15:27:09,541 TRACE  [main] Array size: 32
3 2024-03-05 15:27:09,541 TRACE  [main] Array type:
     SudokuMode[]
4 2024-03-05 15:27:09,542 TRACE  [main] Object allocated at:
     DataModel.java:16
5 2024-03-05 15:27:09,542 TRACE  [main] Object size: 40
6 2024-03-05 15:27:09,542 TRACE  [main] Object type:
     SimpleObjectProperty
```

Výstupní soubor sice má a dodržuje jasně definovanou strukturu, ovšem budoucí extrakce dat by z důvodů redundance a variability textu vyžadovala nadměrnou práci s řetězci na vstupu další aplikace. Tím by mohla být příliš náročná na vytvoření. Analyzátor navíc pracuje za svého běhu s datovými strukturami, které jsou určeny vztahem ke svým atributům a tyto souvislosti nejsou v souboru přímo zaznamenány. To může představovat problémy zejména v případě, pokud by další aplikace chtěla opět pracovat s těmito strukturami. Spojitosti by si musela znovu sestavit zcela sama ze slovního výstupu analyzátoru, což by opět znamenalo práci s řetězci. Popisované problémy by mohly jít jednoduše vyřešit upravením stávající struktury výstupu tak, aby byly získané údaje ve specifickém formátu, ze kterého se později snadno vytvoří objekt se shodným modelem. Mohlo by se jednat o serializaci dat.

2.3.2 Výpis alokací

Jakmile se detekuje nová instance třídou `AllocationDetector`, do výstupního souboru se okamžitě zapíše záznam a následně je volána statická metoda `AllocationCounter.addCounts(StackTraceElement, long)`, které se předá trasa Java zásobníku a vypočítaná velikost přidělené paměti. V této metodě se z reference na detekovaný záznam vytvoří různé řetězcové identifikátory, které se uloží jako klíč do třech kolekcí `HashMap`. V každé této mapě je uchovávána agregovaná velikost přidělené paměti podle konkrétní řádky, metody a třídy, která danou instanci vytvořila. V tabulce 2.1 je vidět, v jakém konkrétním tvaru jsou vytvořené klíče zapsány. Všechny řetězce obsahují informace o místě, kde byla instance alokována, a podle kterého se provádí seskupování velikostí přidělené paměti. Právě z tohoto řetězce jsou v souhrnu vypisována výstupní data, která popisují přidělenou paměť dle řádků, metod a tříd. Údaje se zapisují až po doběhnutí pozorované aplikace do souhrnné části na konec souboru.

Tabulka 2.1: Reprezentace jednotlivých kolekcí ve třídě `AllocationCounter`

název	**klíč**	**hodnota**
lineCounter	<soubor>:<řádka>	celková přidělená paměť
methodCounter	<soubor>:<třída>:<metoda>	
classCounter	<soubor>:<třída>	

Namísto pouhého sčítání velikosti alokací by bylo vhodnější vědět, kolikrát byla instance s určitou velikostí přidělené pamětí vytvořena. Tento údaj lze získat i ze současného výstupu sečtením všech identických záznamů v první části `TRACE`, jedná se však o pracnou a repetitivní záležitost. Ve výpisu 2.3 je vidět ukázka výstupu, kde se dvakrát alokovala stejně velká paměť na stejném místě. Nastává otázka, jestli je vůbec potřeba mít ve výstupním souboru několik stejných záznamů, které ve skutečnosti pouze určují kvantitu jednoho údaje, nebo zda by nebylo vhodnější reprezentovat počet a velikost alokací zvlášť, přičemž by další aplikace vypočítala celkovou přidělenou paměť sama.

Výpis 2.3: Dva záznamy, ze kterých lze určit počet a velikost alokací

```
1 2024-04-19 20:45:04,674 TRACE  [main] Object size: 16
2 2024-04-19 20:45:04,674 TRACE  [main] Object type:
     TestingObject
3 2024-04-19 20:45:04,674 TRACE  [main] Object allocated at:
     Main.java:81
4 2024-04-19 20:45:04,674 TRACE  [main] Object size: 16
5 2024-04-19 20:45:04,674 TRACE  [main] Object type:
     TestingObject
6 2024-04-19 20:45:04,674 TRACE  [main] Object allocated at:
     Main.java:81
```

2.3.3 Výpis duplicit

Informace o duplicitách jsou neúplné, jelikož neposkytují údaje o všech místech vzniku alokací, přestože jejich porovnávání probíhá v rámci celého Java projektu. Stejné instance mohou být vytvořeny v různých souborech zdrojového textu projektu a výstup obsahuje pouze trasu k první z nich detekované v rámci skupiny. Data alokací se postupně ukládají do kolekce `HashMap` ve třídě `DuplicateFinder` podle řetězcového klíče, který je dán názvem třídy instance. Mapa zde ukládá seznam referencí na kandidáty kopií, kteří budou vzájemně porovnáváni. Stejné záznamy se postupně přepisují hodnotou `null` a k počtu duplicit se přičítá 1. Po dokončení porovnávání se vypíše místo alokace k první odkazovanou instanci ze seznamu, které se opět určí z klíče `HashMap`. Problém s chybějícími trasami by šel vyřešit tím, že by ostatní prvky nebyly okamžitě přepsány na `null`, ale zůstaly by po nějakou dobu uložené pro účel výpisu do souboru.

Z výpisu není jednoznačné, co se přesně myslí počtem duplicit. Bez hlubších znalostí o analyzátoru je tento údaj matoucí, protože podle stavby věty by se mohlo spíše jednat o počet kopií i s originálem. Změna významu počtu duplicit je čistě formální záležitost, ke stávajícímu číslu by se přičetlo 1.

Záznamy ve výstupu by mohly obsahovat konkrétní hodnoty atributů kopií, protože by se následně mohl uživatel vyvarovat vytvářením identických instancí.

2.3.4 Rekurzivní hledání duplicit

Hledání stejných objektů je řešeno rekurzivní metodou `eqWithCycles()`, která provádí porovnávání všech atributů referovaných instancí do hloubky. Vzhledem k tomu, že se jedná o rekurzi, může nastat situace, kdy ve vlákně programu přeteče zásobník, načež JVM oznámí vyhození výjimky `java.lang.StackOverflowError`. Jelikož se vyhodnocení duplicit provádí až po ukončení běhu pozorované aplikace, pro uživatele může být vyhození výjimky velmi frustrující, neboť mohl zkoumat dlouhodobě běžící aplikaci a znehodnotí se tak očekávaný výsledek analýzy.

Problém lze do určité míry obejít spuštěním programu s parametrem `-Xss`, který navýší velikost zásobníku všech vláken za cenu jeho vyšší paměťové náročnosti [3]. Trvalým řešením by mohlo být odstranění rekurzivního volání, což by ale vzhledem k současnému principu profilování mohlo být velmi obtížné, jelikož nelze dopředu přesně určit, jakým modelem budou objekty v druhém programu popsány, případně jakou konkrétní hodnotu budou všechny jejich atributy mít.

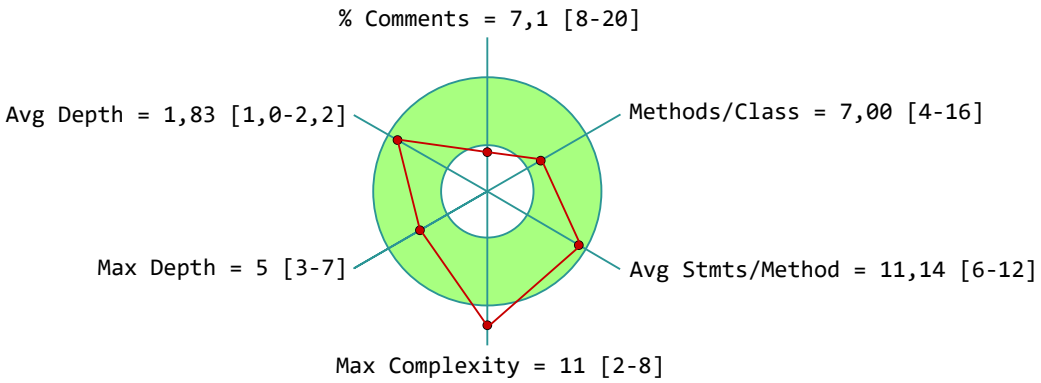
2.3.5 Detekce statických atributů

Nástroj nedokáže poskytnout informace o přidělené paměti statických atributů tříd. Je možné, že analyzátor neumí tyto atributy vůbec detekovat.

Metody vizualizace 3

Vizualizace představuje způsob reprezentace dat grafickým prostředkem a jedná se o alternativní zobrazení několika záznamů k jejich textové podobě. Příkladem může být použití ikon za účelem navigace v uživatelském rozhraní nebo graf zobrazující vývoj kurzu dvou měn za určité období. Záznamy jsou zpracovány do jednoho celku, který umí poskytnout abstraktní pohled na řešenou problematiku. Abstrakce a zjednodušení mnoha informací jsou dvě oblasti, ve kterých vizualizace pomáhají lidem pochopit význam určitých dat [4].

Z hlediska zobrazení metrik zdrojového kódu mohou být vizualizace užitečné pro testování a upravování programu během jeho vývoje. Podle účinnosti zvolené grafiky a míry její interaktivity mohou mít vizualizace větší a rychlejší přínos informací, než čtení jednotlivých záznamů v textové formě. Na obrázku 3.1 je vidět příklad vizualizace, která poskytuje různé informace o parametrech jednoho ze souborů analyzátoru paměti popsaného v kapitole 2. Jedná se o radarový graf, který zobrazuje hodnotu pěti různých metrik najednou, spolu s jejich ideálním rozsahem. Nástroj, kterým je tato vizualizace generována, je podrobněji rozebrán v sekci 3.4.1.



Obrázek 3.1: Vizualizace různých metrik zdrojového kódu radarovým grafem

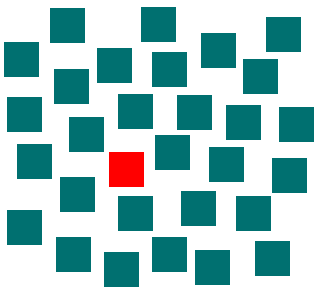
3.1 Princip lidského vnímání grafiky

Nasazení grafiky oproti textu přináší efektivnější a rychlejší zpracování informace, protože proces jejího pochopení probíhá současně s jejím vnímáním [4]. Text, na druhou stranu, musí být nejprve přečten a až následně zpracován mozkem. Mimo jiné, kresby nejsou závislé na konkrétním jazyku, ale pouze na kontextu, ve kterém jsou užity [4]. Může se tak jednat o univerzální způsob, kterým je možno prolomit jazykovou bariéru mezi lidmi.

Lidé vnímají podněty z okolního světa svými smysly – v případě vizualizace zrakem. Všechny získané vjemy jsou okamžitě zpracovávány mozkem, který na ně zcela přirozeně, na základě předchozích zkušeností, dokáže generovat odpovídající reakci [4]. Tato odezva je dána u každého člověka zcela individuálně, proto není samozřejmostí, že zhotovená tvorba bude vždy pochopena stejným způsobem jak autorem, tak pozorovatelem [4, 5]. Z toho důvodu může být vhodný návrh konceptu grafiky, spolu s jejím zařazením do správného kontextu, stěžejní bod procesu implementace jakéhokoliv vizuálního prostředí či zobrazení dat. K tomu lze aplikovat různé znalosti z oblasti lidského vnímání.

3.1.1 Barevný tón

Lidé jsou schopni velmi rychle a přesně identifikovat pozici prvku, který svým barevným tónem vyčnívá vůči ostatním [4]. Tímto způsobem mohou tvůrci vizualizací přitáhnout pozornost sledujících k určité informaci, která je schována mezi mnoha jinými. Názorná ukázka popisované situace je vidět na obrázku 3.2, kde červený čtverec výrazně vyniká oproti zelenomodrým. Konkrétně červená barva je ve většině případů volena za účelem upoutání pozornosti [5]. Užitím kontrastních barev je možné zkrátit reakční dobu člověka, čímž se zrychlí proces předání informace.

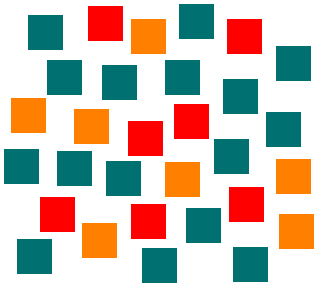


Obrázek 3.2: Upoutání pozornosti prvku jeho kontrastním barevným tónem

Volbou kontrastních barev není zajištěna hierarchie nadřazenosti prvků, protože samotné barvy tuto vlastnost neumí vyjádřit [4]. Pokud by obrázek 3.2 byl tvořen z jedné poloviny červenými čtverci a z druhé zelenomodrými, nebylo by možné

určit, která ze skupin představuje důležitější informaci. Takový stav je možno pozorovat na obrázku 3.3, na kterém je ještě vysvětlen jiný poznatek z oblasti vnímání barev. Další způsoby, kterými je možno od sebe odlišit několik druhů prvků, je např. kombinací jejich různých tvarů, velikostí a jasu [4].

Stejně jako s kontrastem, i prvky které mají podobné barvy budou vnímány podobně [4]. Na obrázku 3.3 je vidět vzájemné odlišení prvků patřící do třech různých tříd, přičemž červené a oranžové prvky jsou vnímány jako sobě podobné a zároveň odlišné od zelenomodrých.



Obrázek 3.3: Odlišení několika skupin prvků barevným tónem

Příkladem upoutání pozornosti odlišným barevným tónem může být pozorování světelné signalizace na přechodu pro chodce. Ta je vidět na obrázku 3.4. Piktogram zde představuje názornou ukázku, co se má vykonat a barva se chová jako zvýraznění. Jelikož jde o dvě protichůdné a rovnocenné činnosti, barvy jsou voleny tak, aby byly vzájemně kontrastní a snadno rozlišitelné.

Pokud by značení neobsahovalo žádné barvy, jako je znázorněno na obrázku 3.5, signalizace by mohla teoreticky stále fungovat, nebyla by ale tak efektivní, jako klasická verze. Konkrétně u semaforu má navíc zvolená paleta barev přidaný význam určitým kulturním pozadím [5]. Z toho důvodu mohou lidé v rámci jedné kultury některé barvy interpretovat podobným způsobem např. červenou negativně a zelenou pozitivně. Tohoto poznatku lze také využít při vytváření vizualizací.

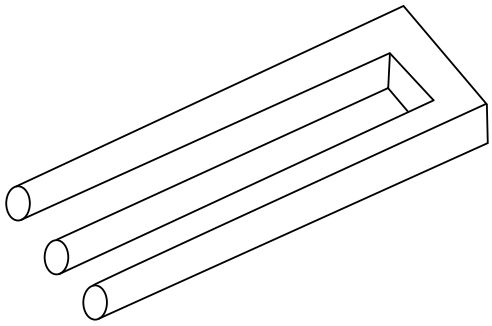
Obrázek 3.4: Klasická signalizace

Obrázek 3.5: Bezbarvá signalizace

3.1.2 Optické klamy

Interpretace pozorovaného snímku nemusí vždy odpovídat zobrazované realitě, v tomto smyslu není lidské vnímání dokonalé [6]. Existuje řada optických klamů potvrzující tuto skutečnost, jedním z nich jsou tzv. nerozluštitelné obrazce [6, 7]. Jde o kresby, které se na první pohled mohou zdát srozumitelné a reálné, avšak při pozorném zkoumání jejich konstrukce nedává smysl [7].

Názorná ukázka takové iluze je vidět na obrázku 3.6, kde je vyobrazeno zdánlivě smysluplné schéma trojzubé vidličky. Pro zajímavost, článek [7] podrobněji pojednává o principu, na kterém se tyto optické klamy zakládají.



Obrázek 3.6: Trojzubá vidlička, která nelze zkonstruovat [8]

3.1.3 Tvarová psychologie

Cílem optických klamů je poukázat na vzniklé nesrovnalosti u rychlého a jednoduchého zpracování vjemů z okolního světa [6]. Při rozpoznávání obrazu se mozek snaží hledat podstatu toho, co vidí, kde pro jeho pochopení automaticky doplňuje chybějící dílky či odstraňuje rušivé prvky podle vlastního očekávání [6]. Tímto způsobem se podvědomě zanedbávají specifické detaily, které brání v porozumění vyššího, třebaže i nesprávného, významu věci.

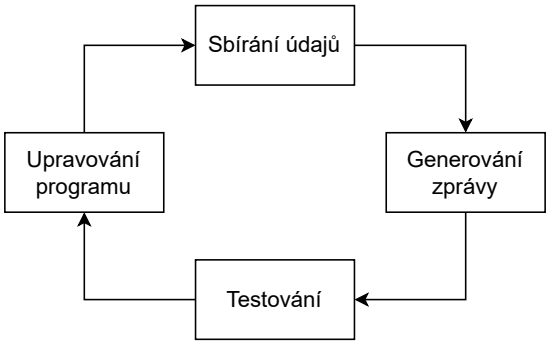
Příklad optického klamu, který je srozumitelný pro osoby se specifickou znalostí, je vidět na obrázku 3.7. Pokud by snímek začal být deformovaný v takové míře, že by nebylo možné zařadit jeho kontext mezi nabyté zkušenosti, stane se v očích pozorovatele nepochopeným [6]. S přihlédnutím na tyto podklady lze do určité míry zjednodušit vizualizaci tím způsobem, že nebude působit přeplněně a zároveň bude obsahovat všechny potřebné informace pro pochopení jejího významu.

Obrázek 3.7: Znalost písmen abecedy umožňuje přečíst *Grafika*

3.2 Význam metrik zdrojového kódu

Během různých fází vývoje lze nad jeho zdrojový kódem softwaru provádět statickou či dynamickou analýzu, ze které je možné získat ukazatele o jeho charakteristikách [9]. Těmto parametrům se říká metriky zdrojového kódu a mohou být dále použity k vytvoření jiných, nových údajů, např. jejich agregací. Všechny tyto záznamy se mohou dále použít k vytvoření zprávy, která umožňuje všem zainteresovaným získat přehled o současném stavu softwaru. Právě součástí této zprávy mohou být vizualizace, které mají za cíl usnadnit pochopení významu všech nasbíraných a jinak odvozených informací.

Po vytvoření zprávy je nutné zvážit, co mělo být cílem analýzy a jak by se mělo naložit s jejím výsledkem. Pokud bylo jejím účelem pouze získat zajímavé statistické údaje, proces je ukončen. Zpracované informace je ale také možno použít jako zpětnou vazbu v rámci vývoje softwaru. Podle výsledků analýzy lze totiž dohledat možné neefektivity algoritmu spolu se zanesenými defekty vyvolávající poruchu. Průběžná analýza tak může být součástí vývojového cyklu softwaru. Na obrázku 3.8 je vidět proces vývoje, který je založený na průběžné analýze a testování zdrojového kódu. Vývoj řízený metrikami má své výhody, zejména usnadňuje tvorbu aplikace, šetří programátorovi čas a následně firmě peníze [9].



Obrázek 3.8: Životní cyklus analýzy a testování programu

Pojem analýza může být často zaměňován s testováním, avšak jedná se o dva rozdílné termíny, které spolu úzce souvisí [10]. Cílem analýzy je dostatečně popsat zdrojový kód kvantitativními či kvalitativními parametry tak, aby byl co nejlépe pochopen jeho význam či funkcionalita okolím. Analýzou lze např. spočítat velikost přidělené paměti alokovaných instancí. Tyto poznatky se následně mohou použít pro zohlednění v budoucím vývoji softwaru. Testování, na druhou stranu, ověřuje samotnou funkčnost aplikace podle specifikace produktu. Jeho výsledkem je, že program buďto pracuje správně, nebo ne.

Vývojáři mohou být nuceni firemní politikou uplatňovat konkrétní standard psaného kódu. Při tvorbě programu si může firma stanovit klíčové metriky a s nimi i optimální hodnoty, které bude od finálního produktu očekávat. Na základě analýzy a její dobře čitelné zprávy mohou všichni zainteresovaní zjistit, že některé parametry mají nevyhovující hodnoty a tím pádem produkt nesplňuje všechna kritéria, která jsou na něj kladena. Z hlediska vývoje softwaru hraje tato informace klíčovou roli, protože testeři přesně ví, jakým komponentám je potřeba se více věnovat a zároveň kdy mohou přestat s jejich testováním [9]. Krom toho, dodržováním stanovených hodnot metrik se mohou jak programátoři, tak firma chránit tím, že byla zajištěna požadovaná úroveň kvality softwaru.

3.3 Použitelnost vizualizací

Grafická reprezentace dat nemusí být vždy vhodně navržená. Její efektivita je založena na několika informativních a emocionálních kritérií [11]. Při zobrazení metrik zdrojového kódu za účelem jeho testování by měla být použitelnost vizualizace hlavní prioritou, protože jejím cílem je poskytnout dostatek informací k vyřešení konkrétního problému. Článek [11] ohodnocuje několik vizualizací podle zmíněných kritérií a zdůvodňuje v čem jsou vybrané techniky efektivní či nikoliv.

Použitelnost vizualizací není zajištěna pouze jejich grafickou reprezentací dat, ale také mírou interaktivity a přizpůsobení, kterou uživatelům nabízejí [4]. V případě profilovacích nástrojů je žádoucí, aby vývojářům byla poskytnuta možnost podrobně prozkoumávat všechny příčiny podezřelých hodnot. Pokud se např. jedná o agregovanou velikost alokací podle tříd, uživatel by měl být schopen dohledat velikosti alokací všech metod této třídy.

Estetické provedení má také vliv na použitelnost vizualizace [4]. Atraktivní a přehledná prezentace dat může pozitivně ovlivnit ochotu uživatele ji používat a nabádat ho k její prozkoumávání [4]. Protože je těžké zvolit univerzální vzhled, který by vyhovoval všem, bylo by vhodné do navrhované vizualizace implementovat možnost měnit alespoň část její podoby. Její snadnou čitelnost by zase mohlo zařídit zobrazení pouze těch nejnutnějších grafických prvků, jelikož si zbytek umí mozek domyslet sám na základě tvarové psychologie.

3.4 Vizualizace metrik existujících nástrojů

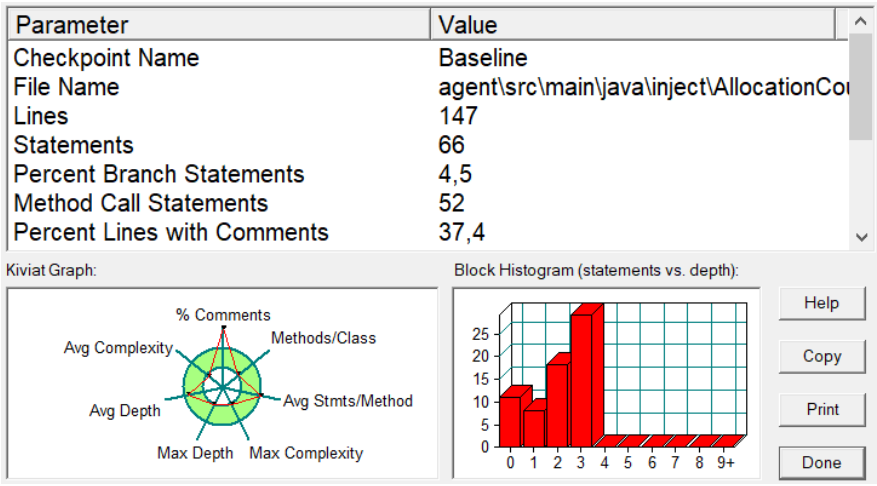
Analyzátorů, které zobrazují metriky formou interaktivní vizualizací, existuje celá řada. Jedná se o programy, které mohou být samostatné okenní aplikace, či rozšíření funkcionalit textového editoru nebo vývojového prostředí *Integrated Development Environment* (IDE). V této sekci jsou představeny dva statické analyzátory, profiler Java programů a plugin, který ukazuje pokrytí zdrojového kódu za běhu.

3.4.1 SourceMonitor

Autor: Jim Wanner, Campwood Software LLC
Web: `https://www.derpaul.net/SourceMonitor/`
Licence: proprietární
Poslední verze: 3.5.16.62 vydána 2. 3. 2022
Podporované jazyky: Java, C/C++, C#, HTML, aj.

SourceMonitor je nenáročný a velice přizpůsobitelný statický analyzátor, který poskytuje přehled základních metrik zdrojového kódu vybraného programu. Součástí programu je uživatelská příručka, ze které jsou některé poznatky uvedené v tomto textu. Implementace je provedena formou samostatné okenní aplikace, která při spuštění vyžaduje cestu ke zvolenému projektu. Mezi poskytované metriky patří počet řádků, pokrytí kódu komentáři a složitost metod, která je dle výchozího nastavení reprezentována větvením programu.

Nástroj má spíše interaktivní charakter než vizualizační. Dokáže se navigovat v poskytnutém zdrojovém textu svou vlastní implementací textového editoru a umí tak najít např. nejvíce zanořený blok v souboru. Na obrázku 3.9 je vidět ukázka výstupu tohoto nástroje, kde hodnoty metrik jsou vypsány v tabulce a pod ní jsou umístěny dva grafy – radarový a sloupcový.



Obrázek 3.9: Náhled metrik zobrazených nástrojem SourceMonitor

Radarový graf (zde zvaný kiviat) obsahuje několik vzájemně nezávislých metrik. Jeho cílem je zobrazit odlehlé hodnoty, které se nachází mimo přijatelné rozmezí reprezentované zelenou oblastí. Zobrazované metriky, včetně jejich intervalu, je možno měnit přes nastavení programu. Tím je zajištěna použitelnost vizualizace i v případě, kdy by byly zvoleny nevhodné výchozí hodnoty. Při důkladném nastavení je tak výstup grafu pro uživatele stále přínosný.

Sloupcový graf zobrazuje počet řádek všech příkazů (svislá osa) podle jejich hloubky zanoření v souboru (vodorovná osa). Iluze prostorového zobrazení je zde na škodu, protože zbytečně ruší čtení dat. To samé lze říct i o vzniklých průsečících os, jelikož k odečtení hodnoty by stačily pouze vodorovné přímky.

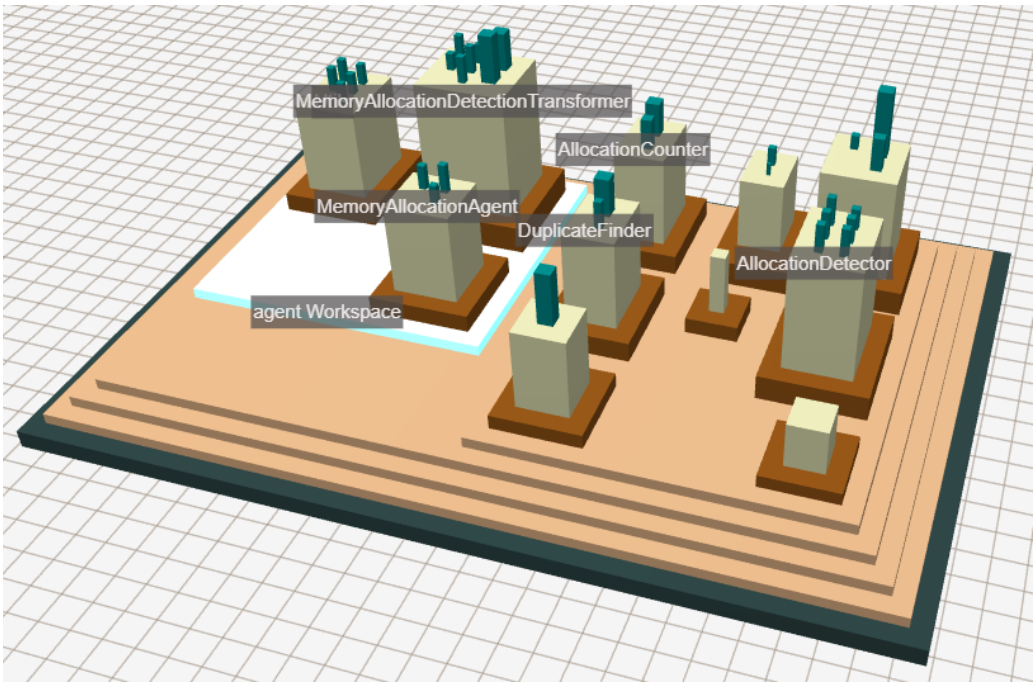
3.4.2 Grappl ve Visual Studio Code

Autor: AUT Ventures
Web: `https://grappl.io/`
Licence: proprietární
Poslední verze: 0.2.0 vydána 9. 3. 2024
Podporované jazyky: jakýkoliv, který je podporován ve Visual Studio Code [12]

Grappl je rozšíření pro textový editor Visual Studio Code, který po spuštění provádí statickou analýzu nad právě otevřeným projektem. Jeho výstupem je detailní zobrazení hierarchie zdrojového kódu v podobě interaktivního 3D města. Tímto způsobem je možno pozorovat např. které metody patří do jakých tříd a jejich složitost určenou počtem řádků. Nástroj se nejspíše stále nachází ve stádiu vývoje, tomu napovídá číslo verze `0.2.0`. Uživatelská příručka [12], ani oficiální web zatím neobsahuje mnoho informací ohledně poskytnutých metrik a jejich konkrétní vizualizaci, umí však zobrazit počet řádků souboru a počet komentářů.

Na obrázku 3.10 na další straně je vidět výstup tohoto nástroje, kde je vyobrazen abstraktní pohled na hierarchii zdrojového kódu analyzátoru z kapitoly 2. Vizualizace umožňuje pohyb kamerou v prostoru a po přiblížení zobrazuje popisky všech budov, podle kterých se v ní lze orientovat. Po kliknutí na blok se naviguje ve zdrojovém kódu a dokáže tak najít konkrétní prvek, který daný kvádr představuje. V nastavení tohoto rozšíření je možnost měnit výšku budov podle počtu řádků a komentářů konkrétního prvku. Ostatní dimenze již nejsou jednoznačně popsány, představují však vypočítanou složitost komponenty [12]. Barva bloku rozlišuje, o jaký typ záznamu se jedná, např. béžová představuje adresář, hnědá soubor a bílá třídu [12]. Kontrastní světlemodrá barva je použita pro zvýraznění právě vybraného bloku.

Vizualizace města vyniká svým vzhledem a potenciálem, protože krom hierarchického pohledu dokáže také mapovat alespoň 4 různé údaje do jednoho záznamu (3 dimenze + barva). Teoreticky by tak šlo i vizualizovat alokovanou paměť nástroje z kapitoly 2, ten ovšem neposkytuje adekvátní počet metrik, aby byly možnosti vizualizace využity naplno.



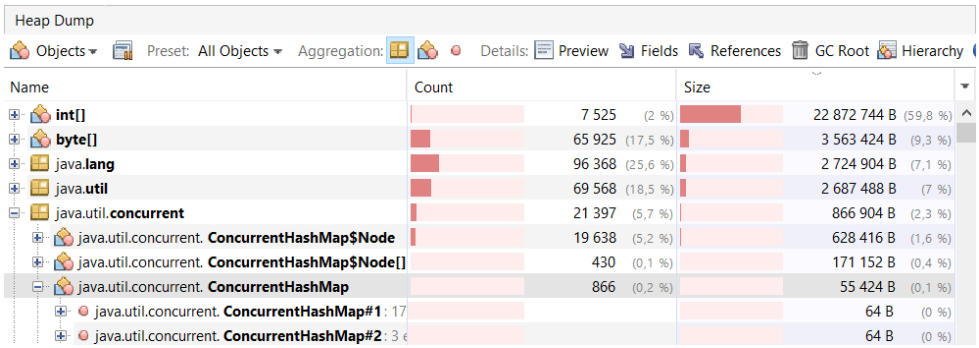
Obrázek 3.10: Vizualizace metrik analyzátoru paměti z kapitoly 2 nástrojem Grappl

3.4.3 VisualVM

Autor: Oracle
Web: `https://visualvm.github.io/`
Licence: GPLv2 + CE
Poslední verze: 2.1.8 vydána 19. 3. 2024
Podporované jazyky: Java

VisualVM je rozsáhlý profilovací nástroj zobrazující informace o lokálních či vzdáleně připojených Java aplikacích. Profilování se může provádět za běhu programu nebo ze záznamu haldy, tzv. heap dump. Tímto způsobem lze získat podrobné informace o programu nahromaděné za určité období nebo z jednoho konkrétního okamžiku. Mezi zobrazovaná data patří zátěž procesoru, stav vláken a zabíraná paměť programu v čase [13]. Uživatel může přepínat mezi několika pohledy.

U zobrazení využité paměti ze záznamu haldy jsou všechna data vypsána v interaktivní stromové struktuře. Její součástí je histogram, který zobrazuje relativní četnost prvku a její velikosti mezi všemi záznamy. Seskupení údajů je možno volit podle balíků, tříd nebo polí objektů. Nástroj umožňuje propojení se zdrojovým kódem projektu, lze tak snadno dohledat různé příčiny podezřelých záznamů. Na obrázku 3.11 na straně 22 je vidět způsob reprezentace dat o využité paměti.



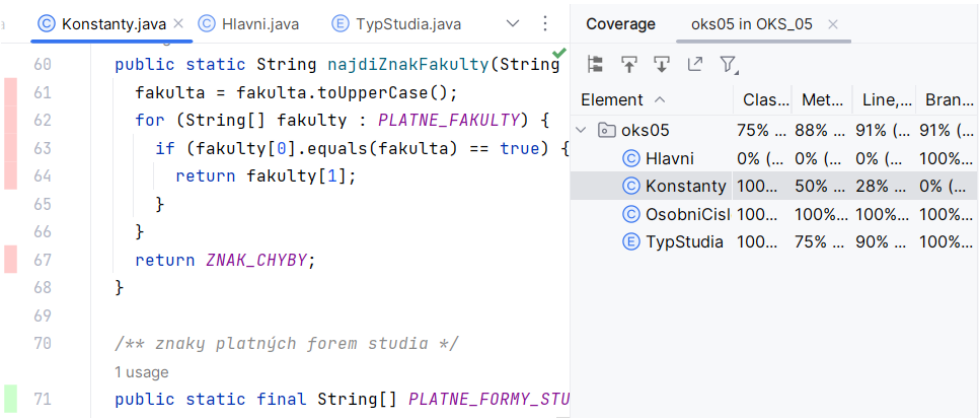
Obrázek 3.11: Náhled zobrazení využité paměti nástrojem VisualVM

3.4.4 Pokrytí kódu v IntelliJ IDEA

Autor: JetBrains
Web: `https://www.jetbrains.com/idea/`
Licence: proprietární / Apache 2.0
Poslední verze: 2024.1 vydána 4. 4. 2024
Podporované jazyky: Java

Vývojové prostředí IntelliJ IDEA má v sobě zabudovaný systém, který umí zobrazit pokrytí řádek kódu za běhu aplikace. Jde o minimalistickou vizualizaci, která nalevo u každého řádku s příkazem zobrazí pruh. Jeho obarvení označuje tři různé stavy, a to jestli byl příkaz na této řádce vykonán celý, částečně nebo vůbec. Uživatel se tak může potřebné informace dozvědět pouhým prohlížením zdrojového kódu.

Součástí vizualizace je i tabulka zobrazující procento pokrytí řádek, kterou se lze přepnout mezi jednotlivými soubory. Ukázka této vizualizace je vidět na obrázku 3.12. Typicky se používá u strukturálního testování, které řeší pokrytí kódu testy.



Obrázek 3.12: Zobrazení pokrytí kódu ve vývojovém prostředí IntelliJ IDEA

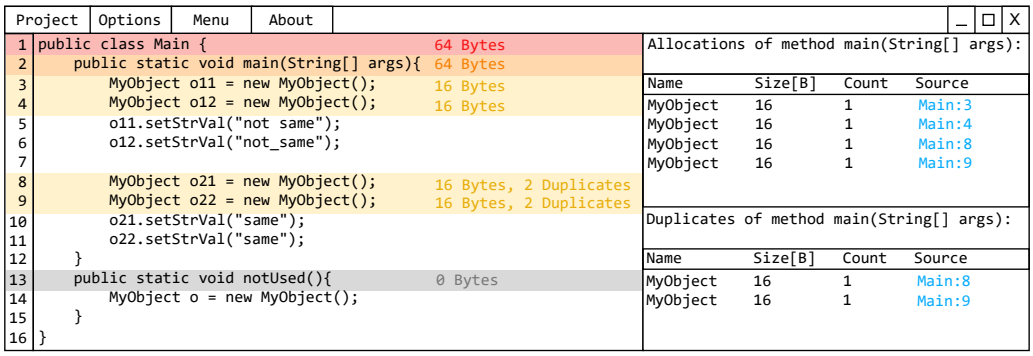
Návrh vizualizace dat analyzátoru paměti — 4

Aplikováním různých znalostí z kapitoly 3 bude navržena vizualizace dat analyzátoru paměti, na který má tato práce navazovat. U návrhu bude kladen důraz zejména na její snadnou použitelnost, přehlednost a uživatelskou přívětivost související s testováním a vyvíjením softwaru za jeho průběžné analýzy. Uživateli bude poskytnut statický pohled na dynamická data přímo ve zdrojovém textu vyvíjené aplikace. Vizualizace bude zobrazovat souhrn údajů posledního běhu pozorovaného programu.

4.1 Navrhovaná metoda vizualizace

Zobrazení dat bude založeno na podobném principu jako funguje rozšíření popsané v sekci 3.4.4, které odlišuje významné řádky zdrojového kódu barevným pruhem.

Obrázek 4.1 ukazuje návrh budoucí vizualizace dat. Nalevo je vidět zdrojový kód vyvíjeného softwaru s podbarvenými řádky alokací. Součástí tohoto zvýraznění je dekorativní text oznamující, kolik paměti v bytech bylo na řádku celkem přiděleno a kolik duplicit se na něm našlo. Napravo jsou umístěny dvě tabulky ukazující podrobné informace ke zvolenému záznamu. Horní tabulka bude obsahovat detaily o alokacích, dolní o duplicitách. Obě budou disponovat odkazy na místo původu alokace ve zdrojovém textu softwaru.



Obrázek 4.1: Návrh vizualizace detekovaných alokací a duplicit

4.1.1 Podbarvení řádků

V sekci 3.1.1 bylo popsáno užití různého barevného tónu několika různých skupin prvků k jejich vizuálnímu odlišení. Analogicky tak půjde zvýraznit řádky obsahující příkaz alokující paměť spolu s těmi na kterých je definována metoda či třída. Uživatel ihned uvidí konkrétní místo alokující paměť při pouhém procházení zdrojového textu. Tímto způsobem získá okamžitě přehled o všech alokacích a bude si moci vybrat, které z nich bude chtít podrobněji zkoumat v tabulkách. Aby na sebe řádky upoutaly větší pozornost, budou dle výchozího nastavení podbarveny odstíny červené barvy, jako je doporučeno v článku [5].

Zvýraznění bude doplněno o text, který bude ukazovat celkovou velikost alokací vytvořených na řádku spolu s jejich počtem nalezených duplicit. Přidaný text bude mít dekorativní charakter, tzn. že nepůjde upravovat a nebude uložen jakou součást zdrojového kódu, bude pouze součástí vizualizace. Uživateli bude umožněno změnit barvu zvýraznění řádku i textu nezávisle na sobě, podle osobní preference.

Vzhledem k tomu, že charakter poskytnutých dat je proměnný a nemusí vždy obsahovat alokace stejných metod a tříd, bude nutné vyřešit problém jejich absence. V případě, kdy budou tyto záznamy chybět, budou řádky definující metodu či třídu podbarveny neutrální šedou barvou. Tímto způsobem zůstane zachován jednotný formát výstupu a uživatel tak nebude vizualizací zmatený, protože bude informován i o třídách a metodách, jejichž vnořené příkazy žádnou paměť nepřidělily. Chybějící data konkrétních příkazů zde nemá smysl řešit, vizualizace podbarví pouze ty řádky, na kterých se vytvořil nějaký objekt. V případě, že se na celém řádku nenajdou žádné duplicity, vypíše se pouze velikost alokované paměti bez textu duplikátů.

4.1.2 Tabulky s podrobnostmi

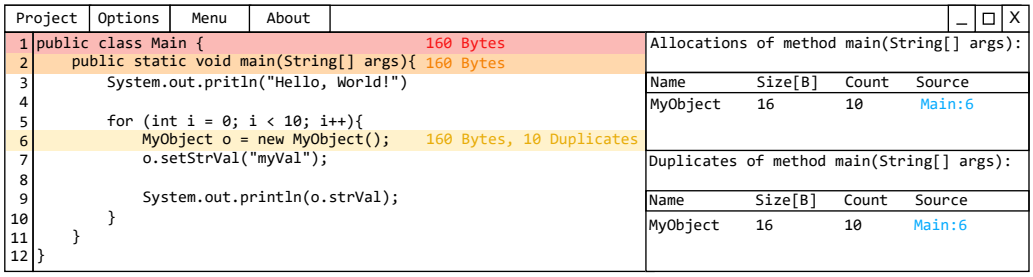
Součástí vizualizace budou dvě tabulky obsahující detailní informace o všech alokacích a duplicitách konkrétního agregovaného prvku, tedy třídy, metody nebo řádku. Tabulka se zobrazí pouze tehdy, pokud se uživatel sám rozhodne detailněji zkoumat všechny příčiny obsahující podezřelou hodnotu podbarveného řádku.

Namísto součtu alokované velikosti celého řádku budou obě tabulky obsahovat velikost jednoho prvku v bytech a počtu těchto vytvořených instancí, či jejich nalezených duplicit. Uživatel tak bude umět rozpoznat rozdíl mezi přidělením paměti jednomu velkému prvku a mnoha drobnými. Dále budou všechny záznamy obsahovat informace o názvu jejich třídy a interaktivní odkaz na původ, kde vznikly v rámci celého Java projektu. Tyto reference budou označeny modrou barvou. Po jejich kliknutí se přepne okno zdrojového kódu na vybraný soubor a kurzor se přemístí na konkrétní podbarvený řádek. Jakmile se vybere tento nový řádek, tabulky o něm zobrazí informace.

V případě, kdy agregované záznamy o přidělené paměti metod a tříd budou chybět, nebude tabulka zobrazovat žádné informace o těchto položkách. Řádky jejich definic budou zvýrazněny šedou barvou ve zdrojovém kódu, proto není nutné detailněji informovat uživatele o chybějícím záznamu.

4.1.3 Případ užití

Samotná vizualizace nebude mít definované žádné kritérium, kdy bude mít konkrétní alokace nevhodnou hodnotu. Toto rozhodnutí bude provádět uživatel zcela individuálně a na základě vlastního očekávání. Příkladem podezřelé konstrukce ve zdrojovém kódu je vidět na obrázku 4.2, kde se v cyklu `for` opakovaně vytváří stejné instance, namísto aby se znovu použily. Uživatel nejprve zahlédne barevné podbarvení, klikne na zobrazení detailů záznamu a v tabulkách napravo uvidí podrobné informace o alokovaných instancí a jejich duplicit. Dozví se, že na řádku 6 ve třídě `Main` bylo vytvořeno příliš mnoho stejných instancí. Pokud to nebylo záměrem, může tento poznatek dále použít k úpravě svého kódu.



Obrázek 4.2: Zdrojový kód s podezřelou konstrukcí

4.2 Hodnoty vstupních údajů

Vizualizace bude přijímat pouze dva druhy záznamů, a to data alokací a duplicit detekovaných na určitém řádků. Tyto údaje budou dále použity k agregaci celkové alokované velikosti metod a tříd, přičemž řádky jejich definic si najde vizualizace sama podle specifických Java symbolů.

Některé hodnoty budou atomické a software si z nich bude sám odvozovat nové metriky. Půjde např. o velikost přidělené paměti jedné instance a počet jejího výskytu za běhu, kde se jejich vynásobením a agregací s jinými alokacemi vypočítá celková přidělená paměť na řádku.

4.3 Možnosti implementace vizualizace

Navrženou vizualizační metodu bude nutné nějakým způsobem implementovat a propojit s analyzátorem paměti z kapitoly 2. Jak ukázaly příklady existujících nástrojů v sekci 3.4, k tomu mohou sloužit dva základní pohledy. Buďto se bude jednat o samostatnou grafickou aplikaci, nebo půjde o rozšíření již existujícího IDE.

4.3.1 Samostatná aplikace

Vizualizaci je možno realizovat tradičním způsobem, jako samostatnou aplikaci s grafickým uživatelským rozhraním. Podobně jsou i navrženy nástroje SourceMonitor a VisualVM popsané v sekcích 3.4.1 a 3.4.3. Mohlo by jít o jednu aplikaci, která by v sobě měla zakomponovanou logiku profilování paměti, vytváření metrik a i jejich následnou vizualizaci. Aby tento software implementoval navrženou metodu, musel by také obsahovat vlastní textový editor, který by uměl podbarvit významné řádky. Také by se musel vymyslet způsob, jak zobrazit tabulky alokací a duplicit.

Výhodou tohoto přístupu je v ucelenosti jedné aplikace, která by vykonávala všechny operace sama. Nevýhodou je, že by pro navrženou vizualizační metodu musela být vytvořena vlastní implementace textového editoru, která by uměla rozpoznat konstrukce jazyku Java a zvládala by navigaci ve zdrojovém kódu celého projektu. Vytvoření takového editoru by bylo příliš pracné, nenabízelo by veškeré funkce již existujících produktů a ani by nesouviselo se záměrem této práce. Krom toho, software by byl plně závislý na konkrétní implementaci analyzátoru, a ne pouze na záznamech, které poskytuje. Tento přístup by byl velmi pracný a ze zmíněných důvodů nebude vizualizace realizována jako součást analyzátoru.

4.3.2 Rozšíření vývojového prostředí

Jak bylo zjištěno v sekci 3.4, vizualizaci je také možno implementovat jako součást již existujícího textového editoru či vývojového prostředí. Vytvořené rozšíření by mohlo umět spustit analyzátor nad přeloženým kódem právě otevřeného softwaru, čímž by se vytvořil výstupní soubor, který by plugin následně zpracoval a zobrazil navrhovanou vizualizaci.

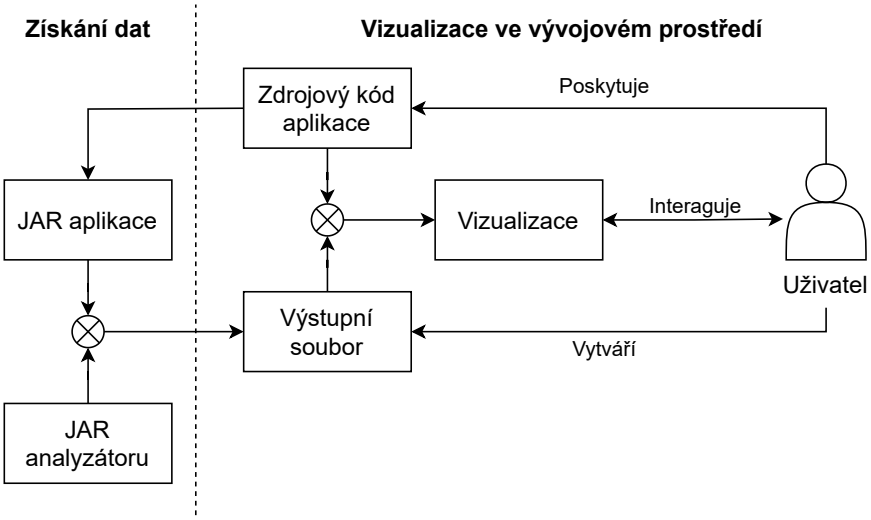
Velikou výhodou tohoto přístupu je snadná navigace mezi konstrukcemi Javy ve zdrojovém kódu, protože bude využit již plně funkční textový editor. Uživatel bude moct využívat různé příkazy, jako např. nalezení místa implementace či definice metod, které ještě více zlepší použitelnost navrhované vizualizace metrik. Dále může být i do určité míry zajištěna uživatelská přívětivost, protože se bude jednat o doplňující funkce již programátorovi známému prostředí. S možnostmi upravit vzhled zvýraznění by také mohly být vyřešeny problémy s různorodostí barevných schémat editoru, jelikož někdo rád programuje ve tmavém či světlém motivu.

Nevýhodou je úplná závislost na zvolené technologii. To zahrnuje jak samotnou podporu vývoje komunitních rozšíření, tak i fakt, že ne všichni programátoři používají stejný software pro vývoj Java aplikací. Vizualizace by se špatně distribuovala a její potenciál by mohli naplno využít pouze ti, kteří by znali vybranou technologii a uměli s ní pracovat. Zmíněná úskalí jsou oproti prvnímu přístupu minimální a dají se vyřešit rychlým průzkumem nabízených možností. Vizualizace bude navržena jako rozšíření nějakého vývojového prostředí pro Javu.

4.4 Propojení vizualizace s analyzátorem

Vizualizace bude realizována nezávisle na konkrétní implementaci analyzátoru, přičemž bude přijímat výstupní soubor se záznamy alokací spolu se zdrojovým kódem pozorovaného programu. Tím bude celý proces zobrazení dat rozdělen do dvou fází, které budou vykonávané rozdílnými aplikacemi, a to získání záznamů následované jejich vizualizací ve vývojovém prostředí.

Na obrázku 4.3 je vidět architektura navrhovaného systému, které znázorňuje obě zmíněné etapy. Ve fázi získání dat se provede kompilace zdrojového kódu a jeho profilování analyzátorem, čímž vznikne výstupní soubor. Podle možností zvolené technologie bude tuto akci vykonávat samotné rozšíření, nebo ji provede uživatel zcela nezávisle sám. Následně se vezme výstupní soubor a zobrazí se vizualizace nad jeho zdrojovým kódem ve vývojovém prostředí. Uživatel bude v tomto případě zodpovědný za poskytnutí shodných verzí zdrojového kódu a výstupního souboru.

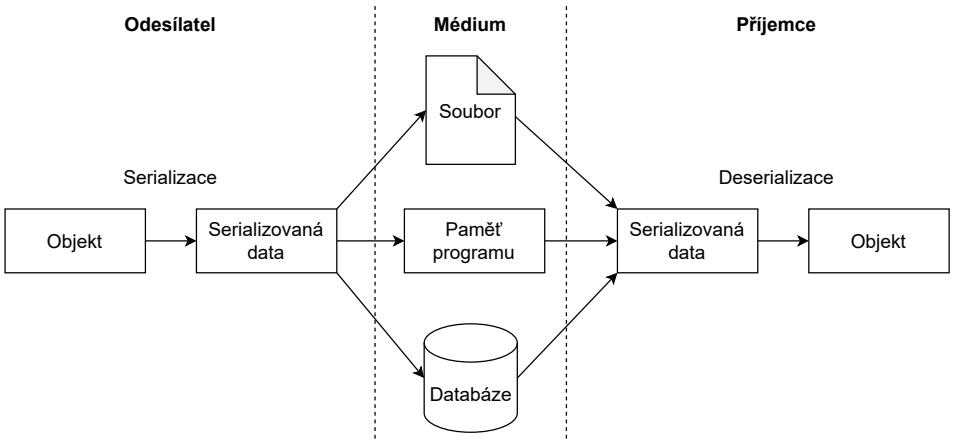


Obrázek 4.3: Postup získání dat a jejich následná vizualizace

4.5 Serializace výstupu analyzátoru

Pokud má vzniknout samostatná softwarová vizualizace, která bude navazovat na výstupní soubor analyzátoru paměti, bude nutné upravit existující formát poskytovaných údajů. Hlavní motivací je, že soubor bude pro další program lehce zpracovatelný a také na straně analyzátoru jednodušeji rozšiřitelný a udržitelný. Toho by šlo docílit vytvořením zcela nové struktury výstupu, která bude zároveň podporovat serializaci dat. Tím by se i mohl vyřešit problém s předáním všech atributů a jejich vazeb na vytvořenou instanci, který byl zmíněn v sekci 2.3.1.

Pro účely jednoduchého přenosu vytvořených datových struktur mezi několika uzly je možno využít proces serializace dat. Serializovaná data lze podobně jako text zapsat na přenosové médium, které si následně protější strana přečte a zpracuje sama [14, 15]. Zpětný chod serializace se nazývá deserializace. Mezi známé a používané formáty souborů patří např. JSON, XML a MessagePack [15]. Tímto způsobem se mohou snadno zpřístupnit vytvořené datové struktury několika různorodým systémům, které umí data deserializovat. Na obrázku 4.4 je vidět zjednodušený proces předání existujícího objektu přenosovým médiem mezi dvěma uzly.

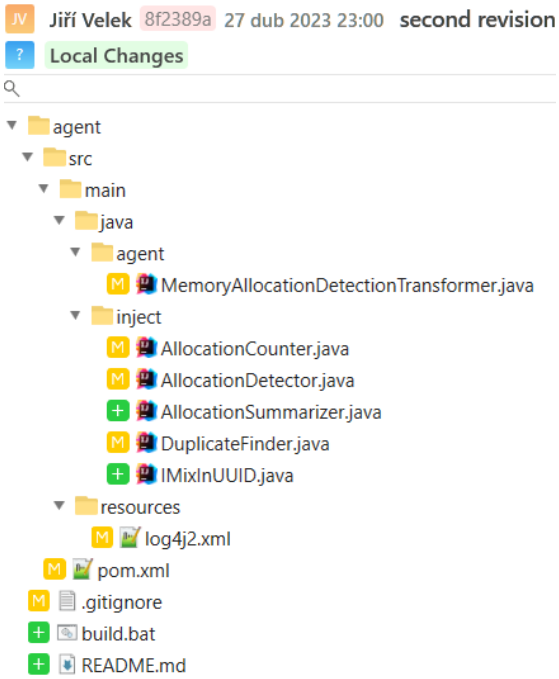


Obrázek 4.4: Předání serializovaných dat mezi dvěma uzly [14]

Úpravy existujícího nástroje

5

Původní nástroj z kapitoly 2 musel být upraven tak, aby jeho struktura výstupních dat obsahovala všechny údaje požadované vizualizací. Tyto modifikace zahrnovaly vytvoření zcela nového formátu souboru, který nyní obsahuje několik přidaných informací užitečných pro navazující aplikaci. Na obrázku 5.1 je vidět souhrn všech provedených změn podle jednotlivých souborů analyzátoru. Oranžovým čtvercem jsou zvýrazněny úpravy textu, zeleným nové soubory. Modifikace se týkaly především balíku `inject`, ve kterém se ukládají a zpracovávají alokace za běhu programu [1]. Samotná logika instrumentace v balíku `agent` nebyla zásadně upravena. Všechny změny jsou uvedeny v repozitáři GitLab[1].



Obrázek 5.1: Souhrn všech provedených úprav analyzátoru paměti

[1] https://gitlab.kiv.zcu.cz/lipka/java-memory-allocation-analyser

5.1 Údaje alokací

Z napsaného textu v sekci 2.3.2 vyplývá, že získaná data za běhu aplikace jsou nadbytečná a vytvořené kolekce `HashMap` ve třídě `AllocationCounter` slouží pouze k uchovávání tras a agregaci výstupních záznamů. Zmíněné kolekce, které seskupují celkovou zabíranou paměť, bylo nutné upravit podle potřeby vizualizace, která vyžaduje údaj o počtu alokovaných instancí.

Jednotlivé výpisy o detekovaných alokací za běhu ve třídě `AllocationDetector` byly odstraněny. Tři kolekce ve třídě `AllocationCounter` byly nahrazeny za jednu, která obsahuje řetězcový klíč s podrobnějšími informacemi. Tato mapa nyní uchovává jako hodnotu počet detekovaných alokací. V tabulce 5.1 je vidět nová reprezentace kolekce `HashMap`, ze které se tvoří souhrn po ukončení běhu programu. Název a velikost se týká instance, která byla alokována. Ostatní údaje označují trasu obsahující příkaz co přidělil této instanci paměť. Vizualizace si následně sama vypočítá celkovou hodnotu záznamu vynásobením počtu výskytu a velikosti jedné alokace.

Tabulka 5.1: Nová reprezentace záznamu ve třídě `AllocationCounter`

název	**klíč**	**hodnota**
lineCounter	<název>;<třída>;<metoda>;<řádek>;<velikost>	počet alokací

5.2 Údaje duplicit

Podobně jako u alokací musel být i výpis duplicitních hodnot předělán. V sekci 2.3.3 bylo popsáno, že duplicity neobsahují všechny trasy vytvořených objektů. Z hlediska interaktivity navržené vizualizace se jedná o zásadní nedostatek analyzátoru.

Provedené úpravy se týkaly především metody `countDuplicates()` definovanou ve třídě `DuplicateFinder`. Metoda byla přejmenována na výstižnější `getDuplicateTraces()` a nyní vrací `HashMap` obsahující řetězcový identifikátor a seznam tras duplicit. V té se pokaždé vytvoří nová mapa, do které se podobně jako u alokací ukládá řetězcový klíč a podle něj se uchovává počet nalezených duplicit. Řetězec zde představuje místo původu vytvořené instance. Pokud jsou nalezeny dvě duplicitní hodnoty, je k počítadlu konkrétní trasy přičtena 1. Následně je instance v seznamu přepsána na `null`, jako tomu bylo doposud. Prakticky se jedná o změnu, která nezasahuje do původního konceptu porovnávání duplicit a zároveň dokáže uchovat informace navíc. Metoda vrací zmíněnou kolekci, ze které se později získá počet všech kopií vzniklých na daném místě. Počty jednotlivých tras se sčítají, aby daly dohromady celkový počet detekovaných duplicit v rámci běhu celého Java programu.

Dále byla provedena změna reprezentace počtu duplicit, která nyní vyjadřuje počet všech nalezených stejných instancí včetně originálu.

5.3 Formát výstupu

Analyzátor byl upraven tak, aby jeho výstupní soubor mohl být pro vizualizaci snadno deserializovatelný. K tomu byl použit formát JSON. Ten byl zvolen zejména kvůli jeho nativní podpoře v programovacím jazyce JavaScript, která zajišťuje jednoduché mapování datových struktur mezi souborem a programem [15]. Tato podpora velmi úzce souvisí s volbou další technologie, která bude popsána v sekci 6.1.

Výstup je nově z části tvořen knihovnou Jackson verze 2.15.0, která pro analyzátor představuje novou závislost. Knihovna je použita k vytvoření serializovatelných objektů. Původní výstup `allocations.log` generovaný Log4j 2 zůstal zachován, ale jeho obsah byl upraven. Nástroj nyní poskytuje dva výstupní soubory, a to `data.json` určený pro vizualizaci rozšířením spolu s `allocations.log`, který obsahuje čitelné informace pro člověka.

Struktura nového výstupu `data.json` obsahuje pouze agregovaná data alokací podle řádků a informace o duplicitách, včetně všech jejich tras. Ukázka formátu souboru je vidět ve výpisu 5.1, kde je také popsán význam jednotlivých parametrů a datové typy jejich hodnot.

Výpis 5.1: Struktura výstupního souboru `data.json`

```
[
  {
    "LINE": [                 # pole alokací
      "class": string,        # třída s balíkem
      "method": string,       # metoda bez parametrů
      "line": number,         # řádek indexovaný od 1
      "size": number,         # velikost alokace [B]
      "count": number,        # počet alokací
      "name": string          # název instance
    ]
  }, {
    "DUPLICATE": [            # pole duplicit
      "name": string,         # název instance
      "size": number,         # velikost duplicity [B]
      "duplicates": number,   # počet duplicit celkem
      "traces": [             # pole tras k duplicitám
        "class": string,      # třída s balíkem
        "method": string,     # metoda bez parametrů
        "line": number,       # řádek indexovaný od 1
        "count": number       # počet duplicit této trasy
      ]
    ]
  }
]
```

5.4 Další změny

Sloučení výpisů do jedné třídy

Byla vytvořena třída `AllocationSummarizer` se statickou metodou `summarizeAllocations()`, která je volána na konci běhu analyzátoru. Ta slučuje dvě rozdílné metody pro výpis agregovaných dat a nalezených duplicit v balíku `agent`. Jedná se o přidanou abstrakci, která by měla v budoucnu zajistit snazší udržitelnost a rozšiřitelnost výstupu analyzátoru paměti.

Zajištění vyšší přehlednosti

Některé části zdrojového kódu byly pročištěny. To zahrnuje přejmenování metod a objektů na výstižnější názvy spolu s drobnými úpravami komentářů. Pro účely snazší použitelnosti nástroje byl do projektu repozitáře GitLab přidán soubor `README`. Ten představuje uživatelskou příručku psanou v anglickém jazyce. Jeho obsahem je postup sestavení, spuštění a také popis nově vytvořené struktury výstupních dat. Informace uvedené v příručce vychází z textu předchozí práce [1].

Experimentální funkce

V rámci experimentování byly provedeny úpravy, které by dokázaly předat konkrétní hodnoty atributů nalezených duplicit. K tomu byla využita knihovna Jackson a serializace objektů ve třídě `DuplicateFinder`. Ukázalo se však, že tento přístup není stabilní a nedokáže vždy poskytnout všechny atributy neznámé instance. Problém totiž nastal při cyklickém porovnávání atributů. Pro tento účel bylo vytvořeno rozhraní `IMixInUUID`, které označuje instanci unikátním identifikátorem a zamezí se tak nekonečné smyčce porovnávání. Vzhledem k tomu, že by bylo potřeba provádět rozsáhlejší testování analyzátoru, byly tyto změny v kódu zakomentovány.

5.5 Kontrola provedených úprav

Aby do analyzátoru nebyly zanesené nové defekty, musel být každý zásah do zdrojového kódu pečlivě zkoumán. Přestože nebyl zásadně změněn princip, na kterém se nástroj zakládá, jeho výstup byl zcela předělán a bylo nutné ověřit, zda poskytuje stejné informace. Tato kontrola byla prováděna průběžně, a to manuálním porovnáváním původního výstupu s upraveným spolu s přihlédnutím na zdrojový kód pozorovaného projektu. K tomu se využily především testovací programy v adresáři `tests`. Všechny tyto projekty byly následně převzaty a nyní mohou sloužit k ukázce implementované vizualizace alokací a duplicit. Současně je také možné porovnat originální výstup analyzátoru s výstupem pluginu.

Implementace navržené vizualizace

6

V následujících sekcích je popsán postup implementace prototypu navržené vizualizace alokací a duplicit z kapitoly 4. Při vytváření praktické části této bakalářské práce byl kladen důraz zejména na její přehlednost a uživatelskou přívětivost. Zdrojový kód byl napsán s ohledem na běžné programovací principy, jako jsou DRY a SOLID. Plugin byl testován sadou jednotkových testů a také uživatelskými testy, na kterých se podílelo celkem 5 dobrovolníků. Z hlediska rozšiřitelnosti a návaznosti u další práce obsahuje zdrojový text mnoho dokumentačních komentářů, které by měly pomoci budoucím vývojářům v porozumění jeho významu.

Podobně jako analyzátor z kapitoly 2, je i zdrojový kód tohoto rozšíření veřejně dostupný na adrese školního repozitáře platformy GitLab[1], na kterou směruje QR kód na obrázku 6.1. Pro účely snazšího průběhu instalace a použití byl vytvořen soubor `README` představující uživatelskou příručku v anglickém jazyce. Jeho obsah je vidět v repozitáři a zároveň je součástí přílohy A. Posledním příspěvkem v době psaní této práce je *tester* přidaný 26. 4. 2024. Podle štítku vydání jde o verzi 0.9.6.

Obrázek 6.1: QR kód směrující na repozitář rozšíření

6.1 Kritéria volby vývojového prostředí

Jedny z populárních vývojových prostředí pro Javu mohou být IntelliJ IDEA, Eclipse a Apache NetBeans [16]. Vývoj Java programů je také možné realizovat v textovém editoru Visual Studio Code (VS Code), který se mimo jiné v hodnocení [16] umístil na 4. místě. K tomu je zapotřebí doinstalovat podporu jazyka Java.

[1] https://gitlab.kiv.zcu.cz/lipka/visualisation-of-allocations

Všechny zmíněné programy umožňují tvorbu komunitních rozšíření, avšak liší se v podpoře programovacích jazyků a možnostmi integrace funkcionalit API. Při rozhodování byly zvažovány dva hlavní faktory: jednoduchost načtení JSON souboru a podpora pluginu umět vyhledat Java definice ve zdrojovém textu. Krom toho byly zohledněny také osobní preference a znalosti daných prostředí. Výsledkem je, že rozšíření bylo implementováno v textovém editoru VS Code.

6.1.1 Načtení JSON souboru

Rozšíření pro VS Code jsou psána v programovacím jazyce JavaScript, spolu s jeho nadstavbou TypeScript. Tato skutečnost měla veliký význam z pohledu návaznosti vizualizace na JSON soubor, kvůli nativní podpoře načtení vytvořených objektů.

6.1.2 Získání Java definic ze zdrojového textu

Bylo zjištěno, že do textového editoru VS Code je možné dodefinovat jakýkoliv programovací jazyk implementací *Language Server Protocol* (LSP) [17]. LSP zajišťuje různé funkcionality požadované po plnohodnotném IDE přímo v textovém editoru. Tyto funkce mohou zahrnovat např. automatické doplňování znaků nebo navigace mezi definicemi tříd určitého jazyka. Na obrázku 6.2 je vidět způsob, kterým LSP usnadňuje komunikaci mezi několika různými programovacími jazyky a textovými editory. Pro jazyk Java již existuje několik pluginů, které zajišťují definice jeho symbolů. Z toho důvodu není potřeba implementovat vlastní LSP.

Obrázek 6.2: Význam LSP mezi programovacím jazykem a textovým editorem [17]

Po samotné definici symbolů bylo nutné určit, jakým způsobem je možné předat získané informace vytvořenému pluginu. Bylo zjištěno, že VS Code API disponuje funkcí `vscode.commands.executeCommand`, která při volání s parametrem

`vscode.executeDocumentSymbolProvider` vrací seznam všech nalezených symbolů v rámci vybraného souboru [18]. Tento přístup je univerzální pro kterékoliv externí rozšíření definující symboly programovacího jazyka, protože se nevolají konkrétní funkce pluginu implementující LSP. Rozšíření je tak teoreticky možné nahradit za jiné, pokud budou obě definovat stejné symboly.

6.2 Využité technologie

Rozšíření bylo napsáno v programovacím jazyce TypeScript pro textový editor Visual Studio Code verze 1.86.0 a vyšší. Součástí zdrojového kódu je soubor `package.json`, který se používá k sestavení projektu systémem Node.js. Následné získání instalačního souboru pro VS Code provádí software vsce. K testování byla použita nativní podpora Node.js a pokrytí kódu testy bylo generováno balíkem c8. Kvalita zdrojového kódu byla zajištěna jeho průběžnou statickou analýzou nástrojem ESLint. Po celou dobu vývoje byl používán verzovací systém Git a v pozdějších fází byl zdrojový kód zveřejněn na školním repozitáři GitLab.

K získání Java symbolů bylo použito rozšíření *Language Support for Java(TM) by Red Hat*[2], které zajišťuje podporu programovacího jazyka Java vlastní implementací LSP serveru. Také se jedná o závislost, kterou si musí uživatel sám doinstalovat do svého textového editoru.

6.3 Struktura zdrojového kódu

Zdrojový kód je rozdělen do několika adresářů představující logické celky softwaru, kde každý z nich zastupuje určitou funkcionalitu vizualizace. Na obrázku 6.3 je zobrazen obsah složky `src` spolu s popisem jednotlivých částí.

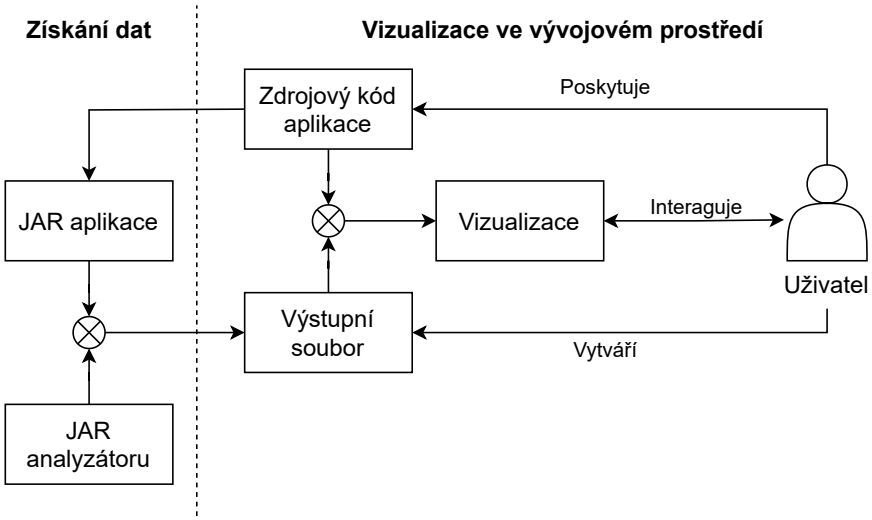
```
src
├── highlight ..... správa podbarvení řádků
├── load ..... načtení dat a symbolů
├── model ..... struktury pro mapování
│   ├── json ..... vstupní data
│   └── lsp ..... Java symboly
├── test ..... jednotkové testy
├── webview ..... správa tabulek detailů
├── Constants.ts ..... definice konstant
├── extension.ts ..... vstupní bod programu
└── ExtensionManager.ts .. obsluha požadavků
```

Obrázek 6.3: Struktura zdrojového kódu rozšíření

[2] https://marketplace.visualstudio.com/items?itemName=redhat.java

6.4 Popis implementace

Vytvořené rozšíření se řídí dvoukrokovým procesem, který již byl popsán v sekci 4.4. Pro připomenutí je také zobrazen na obrázku 6.4. Na vstupu vizualizace je zdrojový kód, který představuje momentálně otevřený Java projekt ve VS Code a výstupní JSON soubor analyzátoru paměti. Proces získání dat samotný plugin neřeší, protože VS Code umožňuje tvorbu vlastní konfigurace běhu, ve které je možné provést kompilaci a následnou analýzu kódu. Očekává se, že si uživatel obstará JSON sám, a to buď přes konfiguraci VS Code, nebo libovolným jiným způsobem.



Obrázek 6.4: Struktura implementované vizualizace

Samotný proces vizualizace lze rozdělit na několik klíčových etap, které jsou ve stručnosti popsány v dalších sekcích. Všechny zmíněné kroky řídí třída `ExtensionManager`, která propojuje požadavky uživatele s ostatními částmi programu.

6.4.1 Načtení vstupních dat a Java symbolů projektu

První fází vizualizace je získání dat z JSON souboru a Java symbolů z momentálně otevřeného projektu. Obě operace provádí třída `Loader` umístěná v adresáři `load`.

Vzhledem k tomu, že TypeScript nativně podporuje načítání dat z JSON souboru, stačí pouze kontrolovat jeho očekávaný formát záznamů. V případě, kdy má vstupní soubor neznámou strukturu, vizualizace skončí s chybovou hláškou. Součástí kontroly jsou také nevalidní hodnoty parametrů, např. záporná velikost alokace. Tímto způsobem však není ošetřena smysluplnost obsahu (číslo řádku, který je mimo rozsah souboru), protože ta se řeší až na další úrovni. Z validního souboru se vezmou záznamy tak jak jsou a vytvoří se z nich objekt `AllocationJSON`.

K získání Java symbolů je potřeba, aby měl uživatel nainstalované rozšíření *Language Support for Java(TM) by Red Hat*. Načítání definic probíhá tak, že se nejprve vyhledají všechny soubory momentálně otevřeného projektu končící příponou `.java` a následně je nad nimi volána metoda `vscode.commands.executeCommand` s parametrem `vscode.executeDocumentSymbolProvider`. Následně se uloží informace o nalezených třídách a všech jejich metodách a konstruktorů do přepravky `ClassRecord`, která je uchována jako záznam v kolekci `Map` zvané `classFileMap`. Významnou informací je rozmezí řádků, na kterých se daný prvek v souboru nachází. Podle ní se provádí agregace záznamů v dalším kroku.

Mezi hledané symboly patří pouze balík, třída, metoda a konstruktor. Přestože analyzátor dokáže poskytnout data alokací i o jiných konstrukcí, např. výčtovém typu, byly z důvodu zjednodušení podstaty vizualizace vynechány.

6.4.2 Mapování a agregace údajů

Poté, co se načtou všechna vstupní data a Java symboly, vytváří mezi nimi `ExtensionManager` vazbu. Nejprve se zpracovávají záznamy alokací a následně až duplicit, protože nalezené kopie jsou ze své podstaty závislé na údajích o přidělené paměti.

Položky alokací v objektu `AllocationJSON` jsou postupně procházeny a podle jejich třídy a balíku se vyhledá stejnojmenný symbol v kolekci `classFileMap`. Následně `ExtensionManager` ověřuje, zda je záznam platný, tj. že zkouší, jestli rozmezí řádků nalezené třídy může obsahovat číslo řádku vstupních dat. Pokud ano, přičte velikost přidělené paměti ke třídě a stejným způsobem dále zkouší přiřadit tuto alokaci ke všem jejím metodám. Z každého validního záznamu je vytvořena přepravka `AllocationRecord`, která uchovává všechny informace o velikosti přidělené paměti, duplicit a konkrétním řádku ve zdrojovém textu. Objekty této přepravky jsou ukládány pro každý soubor zvlášť do kolekce pod názvem `allocationFileMap`. Pokud data nejsou validní, záznamy se přeskakují (např. když název třídy neexistuje nebo číslo řádku je mimo rozsah). Poté co se projdou všechny záznamy alokací, u nalezených metod a tříd bez přidělené paměti se také vytvoří objekt `AllocationRecord`, jehož celková velikost bude rovna 0.

Záznamy duplicit jsou rovnou porovnávány s vytvořenými objekty alokací uvnitř `allocationFileMap`. Data se zde neagregují do metod a tříd, jako u předchozího postupu, ale pouze se přiřadí počet kopií vzniklých na řádku k záznamu `AllocationRecord`. Zároveň se v tomto objektu uchová informace o každé trase z detekovaných stejných instancí. Stejně jako u alokací jsou i nevalidní záznamy duplicit ignorovány. To zahrnuje i případ, kdy jsou ve vstupních datech údaje o vytvořených kopií, kterým nebyla přidělena paměť.

U porovnávání řádků je podstatný detail v rozdílné reprezentaci jejich číslování, protože čísla řádků vstupních dat jsou indexovány od 1, zatímco ve VS Code začínají

od 0. Tuto skutečnost je důležité mít na paměti i u dalších kroků vizualizace. Všechny datové struktury v rámci pluginu jsou navrženy tak, aby měly čísla řádků indexované od 1.

Přestože vstupní data představují přidělení paměti na haldě, rozšíření už neprovádí žádnou kontrolu, jestli daný řádek zdrojového textu doopravdy dokáže alokovat paměť. Mohou tak nastat situace, kde vizualizace bude ukazovat nesmyslné hodnoty. Tento problém je nejvíce zřejmý, když se uživatel rozhodne načíst JSON, který byl vygenerovaný před modifikací zdrojového kódu. U vstupního souboru je předpokládáno, že obsahuje záznamy analyzátoru získané za běhu aktuální verze pozorovaného softwaru a údaje skutečně odpovídají řádkům alokující paměť.

6.4.3 Realizace podbarvení řádků

Podbarvení řádků řeší třída `Highlighter` v adresáři `highlight`. Ta přemění údaje alokací daného souboru `AllocationRecord` na dekorace řádků `HighlightData`, které si uloží do vlastní kolekce zvanou `highlightMap`. Následně se po každém otevření okna se zdrojovým textem načtou vytvořené dekorace daného souboru, není tedy nutné znovu provádět výpočty přímo z modelu rozšíření.

Zvýraznění se řídí návrhem popsaném v sekci 4.1.1. Všechny požadované funkce byly ve VS Code podporovány a podařilo se je implementovat, tj. barevné podbarvení a dekorativní text, který se neukládá spolu se souborem. Mimo jiné byl do textu přidán i záznam o počtu jednotlivých alokací na řádku, čímž je možné snadněji detekovat např. opakované vytváření stejných instancí uvnitř cyklu. Na obrázku 6.5 je vidět ukázka této části výstupu rozšíření.

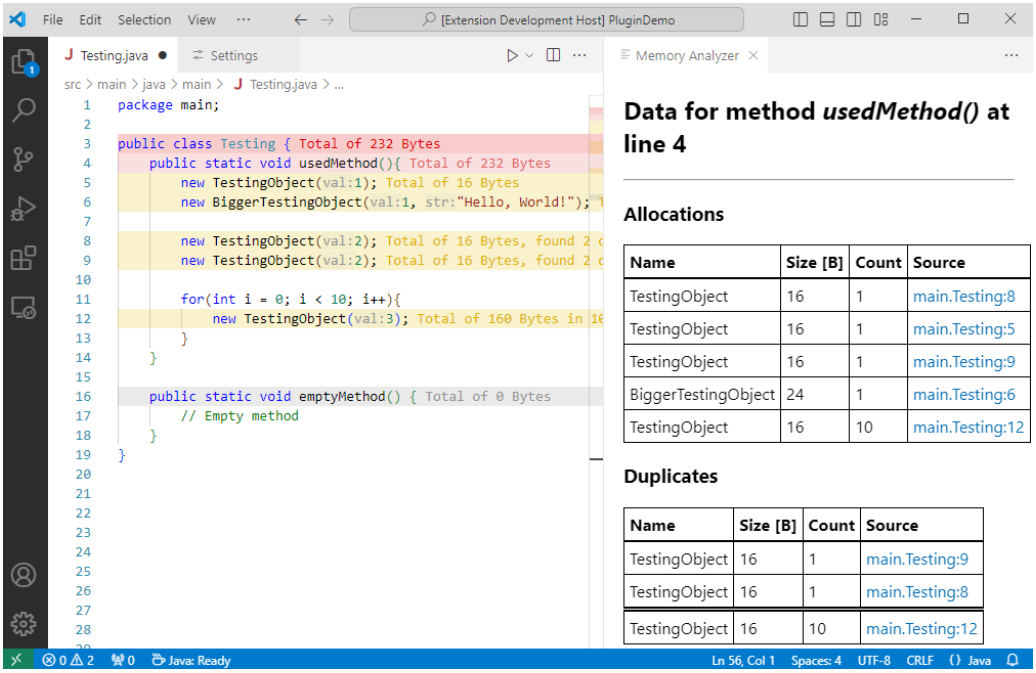
```
1  package main;
2
3  public class Testing { Total of 232 Bytes
4      public static void usedMethod(){ Total of 232 Bytes
5          new TestingObject(val:1); Total of 16 Bytes
6          new BiggerTestingObject(val:1, str:"Hello, World!"); Total of 24 Bytes
7
8          new TestingObject(val:2); Total of 16 Bytes, found 2 duplicates
9          new TestingObject(val:2); Total of 16 Bytes, found 2 duplicates
10
11         for(int i = 0; i < 10; i++){
12             new TestingObject(val:3); Total of 160 Bytes in 10 instances, found 10 duplicates
13         }
14     }
15
16     public static void emptyMethod() { Total of 0 Bytes
17         // Empty method
18     }
19 }
```

Obrázek 6.5: Zvýraznění řádků alokací přímo ve zdrojovém textu

6.4.4 Realizace tabulek s podrobnostmi

Vykreslení tabulek je řešeno přes technologii *webview*, která zařizuje integraci webového prohlížeče přímo ve VS Code. V tomto prohlížeči je zobrazen HTML dokument s kaskádními styly CSS a skriptem psaným v programovacím jazyku JavaScript. Všechny tyto části jsou umístěny v adresáři `webview`, přičemž kostra HTML je přímo obsazena ve třídě `WebviewTable`, která spravuje okno prohlížeče. Všechna data určená k zobrazení dostane správce od třídy `ExtensionManager`, který obsluhuje požadavek uživatele. Zmíněný skript zajišťuje interaktivitu mezi zdrojovým textem a tras obsažených v tabulkách. Také se stará o opětovné použití webview, jehož jeho vytvoření je pro editor zatěžující proces. Poté, co uživatel klikne na odkaz v tabulce, se předá požadavek z prohlížeče třídě `ExtensionManager`, který otevře odkazovaný soubor a najde specifický řádek.

Uživatel má možnost podrobně zkoumat alokace a duplicity vzniklé v libovolné třídě, metodě nebo na řádku. K tomu musí konkrétní záznam vybrat a v kontextovém menu kliknout na zobrazení detailů. Na obrázku 6.6 je vidět celý výstup vytvořené vizualizace, kde v pravé části jsou vypsány informace o zvolené metodě na řádku 4. Dolní tabulka také vizuálně seskupuje různé trasy jedné skupiny duplicit, a to tučnou čarou a mezerou. Uživatel tak vidí, které řádky vytvořily stejné instance, i přestože nezná konkrétní hodnotu atributů. K tomu byla využita znalost tvarové psychologie ze sekce 3.1.3.



Obrázek 6.6: Ukázka vytvořeného rozšíření pro VS Code

6.4.5 Další funkce

Nastavení vizualizace

Do rozšíření byla přidána možnost měnit vzhled vizualizace podle osobní preference a automatické načtení souboru z předem určeného adresáře. Tyto drobné změny by měly uživateli ještě více zpříjemnit používání pluginu.

Spuštění analyzátoru ve VS Code

V rámci zkoumání jak spustit analyzátor nad právě otevřeným projektem se došlo k závěru, že tuto část si musí uživatel obstarat sám. VS Code podporuje spuštění vlastní konfigurace běhu uvedenou v souboru `.vscode/launch.json`. Do té lze přidat parametr `vmArgs`, kterým je možno spustit zkompilovaný Java program s analyzátorem a jeho přídavnými moduly. Jeho podoba je vidět ve výpisu 6.1. Kompletní postup jak vytvořit konfiguraci běhu je popsán v dokumentaci VS Code [19].

Výpis 6.1: Parametr `vmArgs` v konfiguraci běhu `.vscode/launch.json`

```
"vmArgs": [
  "--add-opens", "<modul_1>",  # Otevření modulů pro reflexi
  "--add-opens", "<modul_2>",
  "-javaagent:analyzer.jar",   # Cesta k analyzátoru paměti
  "-jar", "target/app.jar"     # Zkompilovaný Java program
]
```

6.5 Artefakty rozšíření

Při vývoji pluginu vznikly i další artefakty, které přímo nesouvisí s implementací navržené vizualizace. V této sekci jsou uvedeny pouze ty, které se netýkají testování. Zbytek bude uveden v další kapitole.

6.5.1 Uživatelská příručka

V rámci přívětivosti a přehlednosti byl kladen důraz na vytvoření co nejlepší uživatelské příručky. Ta obsahuje všechny potřebné informace od sestavení pluginu a jeho instalaci ve VS Code až po jeho užívání. Mimo to zmiňuje také všechny parametry nastavení a známé nedostatky rozšíření. Právě v této příručce jsou dopodrobna popsány všechny implementované funkce pluginu. Očekává se, že ji uživatel bude mít vždy k dispozici v elektronické podobě na repozitáři GitLab, protože zahrnuje animace, které mohou pomoci s pochopením jejího textu. Stejná příručka, bez obrázků, je uvedena v příloze A. Psána je v anglickém jazyce.

6.5.2 Instalační soubor

Po sestavení zdrojového kódu projektu nástrojem `npm` je zapotřebí vytvořit instalační soubor `VSIX`. To se provádí softwarem `vsce`. Pro zjednodušení distribuce pluginu je v adresáři `release` již připravena nejnovější verze tohoto instalátoru. Celý proces získání balíku `VSIX` a jeho následné instalace je popsán v uživatelské příručce.

6.5.3 Ukázkové programy

Součástí této práce jsou i různé ukázkové Java programy, na kterých je možné spustit vizualizaci. Většina z nich byla upravena a převzata z předchozí práce [1]. Mezi nově vytvořené patří pouze `PluginDemo`, `JSONSerialize` a `JSONSerializeOOP`. Všechny programy jsou uvedeny v adresáři `demo` a lze je sestavit nástrojem Apache Maven. Jejich součástí je i vygenerovaný vstupní JSON soubor, tudíž stačí pouze otevřít projekt ve VS Code a načíst tato data.

Pro účel zjištění funkcí vizualizace byl vytvořen speciální projekt `PluginDemo`, který je doporučeno spustit jako první. Tento program přiděluje paměť vždy stejným instancím, je tak možné manuálně porovnávat očekávané hodnoty se skutečnými. Zdrojový kód obsahuje komentáře u všech řádků alokací, které ukazují, jaké informace by měl plugin přesně zobrazovat.

Programy `JSONSerialize` a `JSONSerializeOOP` vznikly kvůli experimentům spojených se serializací parametrů instancí u výstupu analyzátoru, které byly popsány v sekci 5.4. U projektu `JSONSerializeOOP` bylo zjištěno, že pokud program obsahuje konstrukce objektově orientovaného programování, jako např. dědění, záznamy alokací potomků se zobrazí u rodičovské třídy. Není jasné, zda se jedná o závadu (analyzátoru/pluginu), či nikoliv. Příčiny tohoto problému nebyly dále zkoumány.

6.5.4 Ikona

Za účelem snadné rozpoznatelnosti v seznamu mezi ostatními pluginy byla vytvořena vlastní ikona rozšíření, kterou je možné vidět na obrázku 6.7.

Obrázek 6.7: Ikona rozšíření

Testování prototypu vizualizace

7

Rozšíření bylo testováno manuálně v prostředí *Extension Development Host* [20], které umožňuje otevřít nové okno s editorem VS Code a připojit na něj vyvíjený plugin. Postupně tak byly ověřovány jak funkce technologické části vizualizace, tak smysluplnost zobrazovaných informací ohledně alokací a duplicit. Krom toho byla vytvořena sada jednotkových testů, která kontroluje funkci zodpovědnou za správné načítání vstupního JSON souboru.

Testování rozšíření bylo zaměřeno především na uživatelské testy. Jejich cílem bylo zkontrolovat uživatelskou přívětivost a přehlednost tohoto prototypu vizualizační metody. Na základě zpětné vazby 5 dobrovolníků bylo rozšíření postupně upravováno za účelem zlepšení pluginu v obou zmíněných oblastech.

Pro usnadnění orientace mezi různými vydáními pluginu byl do repozitáře GitLab přidán štítek označující jeho verzi. Aktuální verze rozšíření je 0.9.6.

7.1 Manuální testy

Během procesu vývoje byl plugin spouštěn tak, že se vždy připojil na novou instanci editoru VS Code. Tento přístup eliminuje potřebu opakovaného vytváření a instalaci pluginu pomocí souboru `VSIX`. Při správném nastavení tak lze spustit rozšíření stisknutím klávesy `F5`. K tomu je potřeba sestavit zdrojový kód a zároveň mít otevřený soubor `src/extension.ts` [20]. Příkaz pro sestavení projektu je uveden v uživatelské příručce A.

Po spuštění editoru může být klávesovou zkratkou `Ctrl+Shift+P` zadán libovolný příkaz k vykonání. Veškeré příkazy spolu s parametry nastavení pluginu jsou rovněž uvedeny v uživatelské příručce A.

Testování samotných záznamů alokací a duplicit probíhalo manuálním porovnáváním skutečných hodnot s očekávanými. K tomu byly využity ukázkové programy v adresáři `demo`, které již byly představeny v sekci 6.5.3.

7.2 Jednotkové testy

Pro kontrolu správnosti formátu načteného JSON souboru byla vytvořena sada jednotkových testů `LoadJSON.test.ts` v adresáři `test`. Tyto testy postupně zkouší načítat soubory ze dvou adresářů, ve kterých jsou jak validní, tak nevalidní vstupy. Cílem je ověřit, zda rozšíření dokáže rozpoznat očekávanou strukturu vstupu. Není však už dále zkoumáno, jestli obsah souboru dává z hlediska vizualizace smysl, s výjimkou logicky nemožných hodnot, jako je např. záporné číslo řádku.

Před spuštěním testů je nutné zkompilovat projekt příkazem `npm run compile`. Testy se vykonají voláním `npm run test-coverage`, což rovnou zobrazí jejich pokrytí zdrojového kódu. To zařizuje nástroj c8. Z výpisu 7.1 lze vyčíst, že připravená sada testů pokrývá 99 % řádků v souboru `AllocationJSON.ts`. Ostatní soubory již nebyly testovány jednotkově.

Výpis 7.1: Pokrytí řádek kódu jednotkovými testy v souboru `AllocationJSON.ts`

```
-------------------|---------|----------|---------|---------
File               | % Stmts | % Branch | % Funcs | % Lines
-------------------|---------|----------|---------|---------
AllocationJSON.ts  |   99.21 |    98.59 |     100 |   99.21
-------------------|---------|----------|---------|---------
```

7.3 Uživatelské testy

V různých fází vývoje bylo osloveno celkem 5 studentů Západočeské Univerzity v Plzni, kteří měli za úkol otestovat vytvořené rozšíření podle připraveného formuláře. Testování probíhalo od verze 0.9.2 až po verzi 0.9.5. Podoba rozšíření i testovacího protokolu se v každém vydání měnila na základě nalezených defektů nebo jiných nedostatků. Formulář, který měli testeři k dispozici, a ve kterém měli popsané veškeré pokyny, je součástí přílohy B. Výsledkem testování měla být zpráva, ve které byly uvedené vlastní poznámky k pluginu a nalezené chyby. Jednotlivé zprávy testerů jsou v příloze C. Dále měli testeři odpovědět na dotazník, který vznikl později ve verzi 0.9.4. V tabulce 7.1 jsou shrnuty jeho výsledky.

Tabulka 7.1: Výsledky dotazníku formuláře verzí 0.9.4 a 0.9.5

	T3	T4	T5
Formulář byl pochopitelný	✓	✗	✓
Příručka byla použita	✓	✗	✓
Data se vždy zobrazila	✓	✓	✓
Vizualizace dávala smysl	✓	✓	✓
Plugin by používali	✓	✗	✓

7.3.1 Průběh testů

Testeři dostali instrukce, za pomoci kterých měli zcela sami provést celý proces instalace pluginu s následnou vizualizaci alokací a duplicit. K tomu mohli využít různé poznatky jak z testovacího formuláře, tak z uživatelské příručky. Cílem tohoto přístupu bylo zjistit, jestli příručka obsahuje dostatečné množství informací k běžnému používání rozšíření, včetně jeho instalace.

Testování bylo rozděleno na dvě části. V první měly být ověřeny funkce samotného pluginu na ukázkových příkladech v adresáři `demo`. V další části byl popsán postup, kterým je možné vytvořit konfiguraci běhu analyzátoru ve VS Code, za účelem získání dat vlastního programu. Druhá fáze testování již nesouvisí přímo s vizualizací, z toho důvodu byl tento úsek považován za méně podstatný.

Úkolem dobrovolníků bylo popsat své pocity z vizualizace a poskytnout doporučení ke zlepšení uživatelské přívětivosti a přehlednosti softwaru. Zároveň také měli zaznamenat všechny nalezené chyby a pokusit se popsat jejich příčiny, aby mohly být následně opraveny.

7.3.2 Vyjádření ke zprávám testerů

V této části jsou uvedeny komentáře ke všem zprávám testerů. Na konci některých vyjádření je i jejich rekapitulace. Zprávy jsou součástí přílohy C.

Zpráva 1

Popisovaná chyba `Total of 0 bytes` byla opravena. Jednalo se o nedostatek, který se neshodoval s původním návrhem vizualizace.

Do uživatelské příručky na platformě GitLab byla přidána videoukázka ilustrující, jakým způsobem je možné měnit barvy zvýraznění. Počet barev ke změně zůstal stejný. Vzhledem k tomu, že jde o nastavení, které provede uživatel ve většině případů pouze jednou, nevadí, že je složitější. Navrhovaná metoda by navíc neumožňovala kombinovat různé barvy na stejném řádku, např. zelenou a modrou. Přestože je nepravděpodobné, že by někdo chtěl používat zrovna tuto kombinaci barev, plugin pořád umožňuje její nastavení.

Bylo přidáno automatické zobrazení detailů aktuální zvolené řádky. Funkcionalitu lze vypnout/zapnout podle osobní preference v nastavení. Jedná se o přepínač `Show Details After Line Is Selected`, který je ve výchozím nastavení zapnutý.

Zpráva 2

Příkaz `Toggle visualization` byl opraven. Jednalo se o chybu, která byla způsobena nevhodnou interpretací cesty k souboru v operačním systému Windows. Přestože celý vývoj také probíhal na Windows 10, testování pluginu bylo prováděno

přes integrovaný ladící program `vscode-test` a zmíněný defekt se neprojevoval. Problém s různou interpretací cesty byl vzhledem k nejednoznačné interpretaci cesty ve VS Code API [21] vyřešen až ve verzi 0.9.5, kde byla vytvořena metoda `normalizeWindowsPath()` v souboru `Constants.ts`.

Upravování barev jejich vizuálním výběrem by bylo opravdu lepší než dosavadní řešení. Přesto nebyl nalezen žádný způsob, kterým by se tato možnost dala ve VS Code implementovat.

Tlačítka pro spouštění příkazu nebyla do pluginu přidána, protože cílem prototypu bylo vytvořit minimalistické rozšíření s co nejmenšími zásahy do integrity VS Code. Přidáním nových grafických prvků by mohla být také narušena kompatibilita s jinými rozšířeními. Namísto toho se od uživatelů očekává, že si sami vytvoří klávesové zkratky pro příkazy, které skutečně používají. Toto doporučení bylo připsáno do uživatelské příručky.

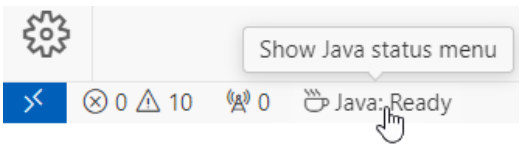
Podle této zprávy je vizualizace přehledná a uživatelská příručka obsahuje potřebné informace k instalaci a použití pluginu.

Zpráva 3

Nalezené chyby ve skutečnosti nepředstavovali žádný nedostatek, plugin se v obou případech zachoval tak, jak měl. Do uživatelské příručky byly přidány další informace ohledně těchto skutečností.

V případě zvýraznění řádků je uživateli umožněno barvu nebo dekorativní text nezávisle na sobě deaktivovat zadáním prázdné (neplatné) hodnoty. Barva se vždy nastaví na výchozí po restartování parametru nastavení.

Hláška o nenalezených Java symbolů informuje o všech souborech, ve kterých nedokázal VS Code detekovat třídu. Zobrazení této informace je sporné, protože se může jednat buď o kód, ve kterém není třída definována, nebo o závadu vizualizace. Bylo totiž zjištěno, že je možné načíst JSON soubor, ještě před získáním Java symbolů z externího LSP pluginu. Chybu se nepodařilo vyřešit a od uživatele se očekává, že bude čekat, dokud se úplně neinicializuje podpora Javy. Tato událost je v rozšíření *Language Support for Java(TM) by Red Hat* indikována zprávou `Java: Ready`, která se zobrazí v dolním stavovém pruhu VS Code, jako je vidět na obrázku 7.1. V případě, kdy se zobrazí podbarvení řádek ve všech souborech s definovanou třídou, je možné hlášku ohledně chybějících Java symbolů ignorovat.



Obrázek 7.1: Indikace plně inicializované podpory jazyka Java ve VS Code

Autor této zprávy byl s vizualizací spokojený a oceňuje možnost měnit barvy podbarvených řádků. Chybělo mu však odlišení textu kurzívou, proto bylo přidáno do nastavení spolu se zvýrazněním tučným písmem. Obě možnosti lze kombinovat. Názvy metod a tříd u tabulek podrobností byly v nadpisu odlišeny kurzívou, aby lépe vynikaly mezi ostatním textem.

Zpráva 4

Podle vyplněného dotazníku měl tester potíže s pochopením formuláře a ze začátku při testování nejspíš ani nevěděl o uživatelské příručce. To mohlo být způsobeno tím, že prohlížeč ve kterém byl testovací formulář otevřen, nepodporoval interaktivní odkazy, tudíž nebyla možnost se dostat na GitLab projektu. Skutečný důvod je ovšem neznámý.

Do testovacího formuláře byla přidána adresa směřující na repozitář s pluginem. Budoucí testeři by již neměli mít problém s hledáním uživatelské příručky, pokud by jim nefungovaly odkazy v prohlížeči dokumentu.

Tomuto testerovi se líbilo intuitivní a barevné provedení pluginu, který se mu také později podařilo zprovoznit.

Zpráva 5

Navrhovaný souhrn všech problémů ve vizualizaci není triviální záležitost, protože je obtížné definovat, co by přesně mělo pro vývojáře představovat problém. Programy mohou záměrně obsahovat konstrukce, které alokují mnoho paměti nebo vytvářejí několik duplicitních instancí, jelikož jsou tak navrženy. Prototyp vizualizace umožňuje uživatelům prohlížet data analyzátoru v jiné než textové formě a dále už neprovádí analýzu nevhodných konstrukcí či problematických úseků v kódu. Z těchto důvodů si musí každý vývojář prozatím sám definovat problémy, kterými se chce zabývat.

Popisovaná chyba ohledně reakce pluginu byla způsobena neposkytnutím stejné verze zdrojového kódu a JSON souboru. Tester se totiž pokusil načíst stará data, která byla vygenerována před úpravou programu, a proto vizualizace nedokázala přesně najít řádky alokací. Podle návrhu rozšíření je uživatel zodpovědný za poskytnutí správných vstupů, z toho důvodu se nejedná o problém, který je třeba u tohoto prototypu řešit.

Rozšíření bylo ohodnoceno pozitivně, zejména kvůli jeho jednoduchosti. Tester zkoušel i načítat náhodná data, přičemž plugin zobrazil chybovou hlášku, kterou měl. Uživatelská příručka byla pro testera srozumitelná a s její pomocí dokázal úspěšně nainstalovat rozšíření.

Závěr 8

Cílem této práce bylo navrhnout a vytvořit prototyp implementující vlastní vizualizační metodu záznamů alokací paměti a duplicitních instancí. Za tímto účelem vzniklo rozšíření textového editoru Visual Studio Code, které umožňuje statický pohled na získaná dynamická data. Záměr bakalářské práce byl úspěšně splněn.

V kapitole 2 byl představen analyzátor paměti, pro který měla být vytvořena grafická prezentace dat. Nejprve byl stručně vysvětlen princip, na kterým se nástroj zakládá a následně byl podrobněji rozebrán jeho výstup. Byly zde identifikovány a popsány některé nedostatky jak analyzátoru, tak formátu výstupních dat.

Další kapitola byla zaměřena na vizualizace spolu se způsoby, kterými mohou pomáhat vývojářům s hledáním problémů. Toto povídání bylo doprovázeno názornými ukázkami, na kterých byly vysvětleny principy vnímání grafiky. Dále byly představeny vizualizace metrik zdrojového kódu několika existujících nástrojů.

Kapitola 4 se zabývala návrhem vlastní vizualizační metody na základě poznatků z obou předchozích částí. Byl také předveden koncept architektury navrhovaného systému spolu se všemi rozhodnutími nezbytných k jeho realizaci. V této části byla uvedena myšlenka, že vizualizace bude realizována nezávisle na konkrétním analyzátoru, a to ve formě rozšíření vývojového prostření či textového editoru.

Popis implementace celého systému byl rozdělen mezi dvě kapitoly – 5 a 6. V první z nich byly popsány všechny modifikace analyzátoru paměti, které i vyřešily některé jeho nedostatky. Další část se zabývala popisem realizace rozšíření, přičemž byl kladen důraz na jeho uživatelskou přívětivost a přehlednost. Zejména z toho důvodu vznikla detailní uživatelská příručka, ve které je možné dohledat všechny implementované funkce, požadavky a nedostatky nástroje. Příručka je uvedena v příloze A a je psána v anglickém jazyce.

Poslední část této práce byla věnována testování vytvořeného prototypu. To bylo realizováno zejména průzkumnými testy s užitím formuláře, který je uveden v příloze B. Cílem bylo zjistit, zda je prototyp uživatelsky přívětivý a přehledný. Za tímto účelem bylo osloveno celkem 5 studentů, na základě jejichž zpráv byl program příslušně upraven. Samotní testeři hodnotili plugin pozitivně, přičemž měli výhrady osobní preference. Všechny zprávy jsou součástí přílohy C.

Uživatelská příručka A

V této příloze se na dalších stranách vyskytuje uživatelská příručka vytvořeného rozšíření, která je psaná v anglickém jazyce. Očekává se, že uživatelé pluginu budou spíše používat její elektronickou podobu zveřejněnou na školním repozitáři GitLab[1], protože obsahuje videoukázky různých postupů, které je obtížné popsat v textové podobě. Na obrázku A.1 je zobrazen QR kód směřující na adresu zmíněného repozitáře. Z důvodu úplnosti bakalářské práce je text příručky uveden i zde.

Obrázek A.1: QR kód směřující na repozitář rozšíření

[1] https://gitlab.kiv.zcu.cz/lipka/visualisation-of-allocations

Memory Analyzer for Java (Visual Studio Code extension)

Visualize Java object memory allocations and duplicates line-by-line.
IMPORTANT: In the current version, this tool provides only the data visualization of the output of an external Memory Analyzer application and does not analyze the application by itself.

Features

- Visualize data provided by an external Memory Analyzer application.
- Highlight lines that allocated memory on the heap at runtime.
- Display how many objects, what types and how much memory has each line allocated.
- Aggregate line data to see a method and class total allocation size.
- Tables containing detailed information with interactive links.
- Trace duplicates to their sources.
- Customize highlight color settings.

Requirements

Plugin is made for the desktop application Visual Studio Code 1.86.0 and newer.
Before using, ensure that Language Support for Java(TM) by Red Hat is also installed.

How to install

Obtaining VSIX file

In order to install the extension, you have to acquire VSIX (Visual Studio extension installer) file. You can either create it from the source code yourself, or download it from release folder, where is the latest build.

Creating VSIX from the source code

If you have decided to download the VSIX file from the release folder, skip this section.
To build the souce code yourself, you will need a Node.js with `npm` (node package manager) installed on your machine. If you have `npm` ready, follow these steps:

1. Download the source code from this repository.
2. Build the extension: `npm install`

After you have successfully built the project, you will have to create VSIX file using vsce (Visual Studio Code extension). Do the following:

1. To install the `vsce` tool, run: `npm install -g @vscode/vsce`
2. In the root folder of the built extension, run: `vsce package`
2. Alternatively you can run a prepared script, which will create the VSIX file in the `release` folder: `npm run pack-vsix`

Doing the steps above will provide you with `java-memory-analyzer.vsix`.

Installing the extension in Visual Studio Code

If you have the VSIX file ready, you can install the extension inside Visual Studio Code:

1. Open Visual Studio Code.
2. Click Extensions.
3. Click Views and more actions...
4. Select Install from VSIX...

How to use

In order to visualize allocation data, you have to provide a JSON file that can be generated with an external Memory Analyzer application. If you have the data ready, you can do the following:

1. Open your Java project in Visual Studio Code and wait for Java symbols to fully load (indicated by Java: ready in the bottom status bar).
2. Run command Memory Analyzer: Load JSON file.
3. To show more details, run Memory Analyzer: Show line details with a line selected.
4. Clicking on the link of the details table will move the cursor to the position of that particular allocation.

If you get prompted with a warning Found no Java symbols in files, Java language support could not find any class, method or constructor in the file, therefore no allocation data will be shown for these files. If the file contains some of these symbols, be sure that you run Memory Analyzer: Load JSON file with the language support fully loaded (indicated by Java: ready).

Tips:

- Commands can be found in the context menu with shortcut `Ctrl+Shift+P`.
- To quickly navigate, search for Memory Analyzer.

How to obtain JSON file

This repository includes demo applications, each featuring pre-generated `data.json` that you can use for a quick demonstration of this extension.
In order to create `data.json` yourself, you will need to build the external Memory Analyzer application using Maven. Instructions are available in the project repository.
To build the demo applications, you can call the following command from root folder of this repository, which will create a JAR file `<demoApp>/target/app.jar` for each demo respectively: `mvn -f demo clean install`
Each demo in this repository contains configuration `.vscode/launch.json` that can be used to easily run compiled memory analyzer application.

Commands

- `Memory Analyzer: Load JSON file`: load the generated JSON file
- `Memory Analyzer: Toggle visualization`: toggle the visualization on/off
- `Memory Analyzer: Show line details`: show allocation and duplicate details for the currently selected line
  - this option is also available under mouse right-click context menu

Tip: It is recommended to map the commands to key bindings for personal convenience, refer to the official Visual Studio Code documentation.

Settings

JSON load settings

- `java-memory-analyzer.json.defaultPath`: absolute path where the JSON is located. If not empty, this path will be always have priority over the popup window
- `java-memory-analyzer.json.askToSavePath`: turn off to stop asking if you want to set currently loaded path as default

Tips:

- It is recommended to keep these settings to their default values under User settings and change only Workspace settings, refer to the official Visual Studio Code documentation.
- If you are unable to load a proper JSON file and no popup appeared, check settings if there is a valid Default path.
- If you leave Default path empty and set Ask to save path to false, the analyzer will always ask for the location of the JSON file.

Line details settings

- `java-memory-analyzer.details.goToLineImmediately`: turn on to show a newly selected line details when the reference link is clicked
- `java-memory-analyzer.details.showDetailsAfterLineIsSelected`: after a line is selected and both visualisation and table panel are toggled on, the tables will show details of the selected line

Color settings

- `java-memory-analyzer.color.lineBackground`: background color of line allocation
- `java-memory-analyzer.color.lineText`: text color of line allocation
- `java-memory-analyzer.color.methodBackground`: background color of method allocation
- `java-memory-analyzer.color.methodText`: text color of method allocation
- `java-memory-analyzer.color.classBackground`: background color of class allocation
- `java-memory-analyzer.color.classText`: text color of class allocation
- `java-memory-analyzer.color.emptyBackground`: background color of no allocation (only for method/class)
- `java-memory-analyzer.color.emptyText`: text color of no allocation (only for method/class)

Tips:

- All colors accept a string in the CSS color format.
- If you have the visualization toggled on, the changes can be seen immediately.

Highlight font settings

- `java-memory-analyzer.highlightFont.bold`: turn on to have bold highlight font
- `java-memory-analyzer.highlightFont.itelic`: turn on to have italic highlight font

Known Issues and limitations

- The format of JSON file is same as an output of the external Memory Analyzer application, no other JSON format is supported.
- The visualization can only show as much data as it is provided with the JSON file.

- Extension does not actually detect the keyword responsible for memory allocation (such as `new`), as it only highlights lines according to the provided file.
- No support for visualization of nested classes, nested methods and enumerations.
- If you try to run Load JSON file without the Java language support fully loaded (indicated by Java: ready), some files may be missing allocation data. If this problem persists, the best solution is to restart the Visual Studio Code.

Testovací formulář

B

Obsahem této přílohy je třístránkový formulář, který dostali dobrovolníci elektronicky za účelem otestování vytvořeného rozšíření. Dokument obsahoval interaktivní prvky zvýrazněné modrou barvou a žádné jiné instrukce nebyly poskytnuty. Účelem tohoto formuláře bylo vytvořit samostatný dokument, který by obsahoval všechny potřebné informace ohledně testování pluginu.

Testovací formulář

Vašim cílem je otestovat a napsat recenzi k pluginu, který zobrazuje informace o zabrané paměti za běhu aplikace přímo ve zdrojovém kódu. Testování je rozděleno na dvě části, kde v první si zobrazíte alokace na předpřipravených datech a v druhé si vytvoříte datový soubor pro váš libovolný projekt, které následně vizualizujete. První část se zaměřuje čistě na funkce pluginu, té věnujte maximální úsilí. Druhá část je vedlejší, netestuje přímo naprogramované rozšíření, ale popisuje postup jak jednodušeji dostat data z jiné aplikace. Prosím, **zaměřte se použitelnost pluginu, ne tohoto druhého programu**. Všechny instrukce byste měli být schopni zvládnout sami, případně s užitím uživatelské příručky README v repozitáři projektu[1].

V tomto dokumentu jsou prvky, které umožňují zápis v prohlížeči PDF. Než začnete vyplňovat všechna pole, doporučuji, abyste si zkusili uložit soubor. Pokud chcete, můžete poslat i vlastní PDF (ideálně s obrázky). Nemusíte se nutně držet osnov tohoto dokumentu, jsou spíše inspirativní. Samotné testování a hodnocení je omezeno pouze vaší fantazií. U každé chyby kterou najdete popište podrobný postup k její reprodukci.

Co budete potřebovat

Na testování obou částí je nutné mít nainstalovaný textový editor Visual Studio Code verze alespoň 1.86.0 s pluginem pro podporu Javy Language Support for Java(TM) by Red Hat. Tyto požadavky jsou dostatečné pro první část testování, samotnou Javu pro spuštění vizualizace na předpřipravených datech nepotřebujete.

V druhé části bude potřeba mít nainstalovanou Javu, Maven a balíček rozšíření pro plnohodnotné IDE Javy ve VS Code. Je zde také potřebná znalost ohledně vytváření konfigurací běhu. Přestože ukázkové programy obsahují tyto konfigurace, nelze je použít jako univerzální řešení pro všechny Java projekty.

Informace o vás

Studijní obor:

Datum testování:

Operační systém:

Verze pluginu (GitLab tag):

Verze VS Code (Help → About):

Jak často používáte VS Code:

[1] https://gitlab.kiv.zcu.cz/lipka/visualisation-of-allocations

Pokyny pro testování

První část:

1. V adresáři `release` je instalační balíček `java-memory-analyzer.vsix`. Nainstalujte si jej do vlastního editoru.

2. V adresáři `demo` je několik Java projektů pro demonstraci pluginu. Otevřete libovolný z nich (File → Open Folder) a načtěte datový soubor `data.json` příkazem `Memory Analyzer: Load JSON file`. Doporučuji začít s `PluginDemo`.

3. Prohlédněte si zobrazené informace alokací, jestli vám dávají smysl. Kliknutím pravého tlačítka na myši, příkazem `Memory Analyzer: Show line details`, se koukněte na detaily libovolné řádky.

4. Vypněte/zapněte vizualizaci příkazem `Memory Analyzer: Toggle visualization`. Zkuste nejdříve bez otevřené tabulky detailů a poté s otevřenou.

5. Upravte nastavení barvy zvýraznění podle vašich představ. Použijte různé zápisy CSS.

6. Otevřete projekt `Sudoku` a postupně načítejte datové soubory a prohlížejte si jejich výsledky. Každý bude ukazovat jiné údaje, jelikož se jedná o souhrn odlišných alokací stejného programu.

Druhá část:

1. Ve VS Code otevřete projekt s vaší libovolnou Java aplikací a vytvořte její JAR (např. pomocí Mavenu v příkazové řádce, nebo přes vlastní konfiguraci VS Code). Následně stáhněte a udělejte JAR externího analyzátoru paměti.

2. Každé demo obsahuje nastavení běhu `.vscode/launch.json` pro pohodlné spuštění analyzátoru přímo ve VS Code. Tuto konfiguraci zkopírujte a upravte tak, aby obsahovala cestu k JAR souborům analyzátoru paměti a vašeho programu. Pokud aplikaci potřebujete předat parametry, postupujte podle oficiální příručky Visual Studio Code. Analyzátor spustíte stisknutím klávesy `F5`.

Dotazník

Byl pro vás tento protokol dostatečně srozumitelný? Ano Ne

Použili jste k testování pluginu README z repozitáře? Ano Ne

Proč?

Zobrazila se data při každém volání `Load JSON file`? Ano Ne

Dávala vám smysl vizualizace alokací a duplicit? Ano Ne

Jaký byl váš hlavní dojem z používání pluginu?

Používali byste tento nebo podobný plugin? Ano Ne

Co se vám líbilo?

Co se vám nelíbilo?

Co byste upravili/přidali/odstranili?

Nalezené chyby, poznámky:

Zprávy testerů C

V této příloze jsou uvedeny jednotlivé zprávy testerů. Vzhledem k tomu, že někteří z nich vyplnili testovací formulář B, který zde nemá smysl znovu uvádět, byly výsledky zpracovány do vhodnější podoby. Význam všech nalezených chyb a poznámek k pluginu zůstal zachován, ve větách byly akorát opraveny případné gramatické chyby a překlepy. Kvůli lepší čitelnosti byly invertovány barvy některých obrázků. Z toho důvodu nemusí přesně odpovídat barvy zvýraznění.

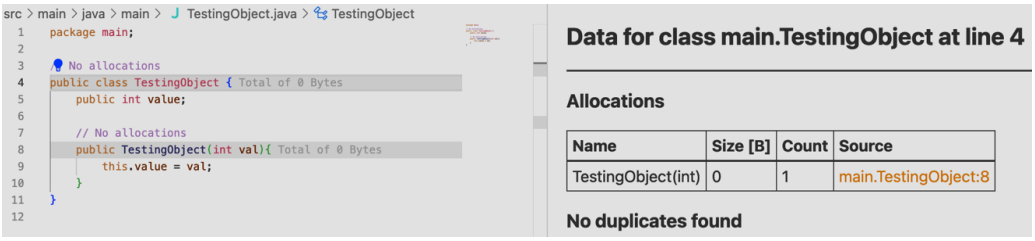
Zpráva 1

Informace o testerovi:

Studijní obor:	Informatika a výpočetní technika
Datum testování:	12. 3. 2024
Operační systém:	macOS Sonoma
Verze pluginu (GitLab tag):	0.9.2
Verze VS Code (Help → About):	1.87.1
Jak často používáte VS Code:	zřídka kdy

Nalezené chyby, poznámky:

Alokace `Total of 0 bytes` → v okně na pravé straně VS Code se zobrazuje, že byla provedena 1 alokace o velikosti 0 bytů – lepší by bylo napsat 0 alokací.



(po kliknutí na 4. řádek)

Přidat obrázek jak efektivněji měnit barvu do README.

Barev na měnění je moc, radši bych měnila jen např. 3 barvy… z každé barvy by se vygeneroval text + víc průsvitné pozadí.

Líbilo by se mi, kdyby se v okně vpravo zobrazovala aktuální zvolená řádka.

Zpráva 2

Informace o testerovi:

Studijní obor:	Informatika a výpočetní technika
Datum testování:	12. 3. 2024
Operační systém:	Windows 10 Home, 22H2
Verze pluginu (GitLab tag):	0.9.3
Verze VS Code (Help → About):	1.88.1
Jak často používáte VS Code:	docela často

Nalezené chyby, poznámky:

Příkaz `Toggle visualization` je nefunkční (nic neprovede).

README.md dobře ukazuje instalaci a použití.

Při upravování barev by byl lepší lepší colorpicker než upravování stringu.

Pro spouštění příkazů by byla lepší tlačítka než ruční psaní.

Vizualizace je přehledná a líbí se mi.

Zpráva 3

Informace o testerovi:

Studijní obor:	Informatika a výpočetní technika
Datum testování:	15. 3. 2024
Operační systém:	macOS Sonoma
Verze pluginu (GitLab tag):	0.9.4
Verze VS Code (Help → About):	1.87.2
Jak často používáte VS Code?	nepoužívám vůbec

Dotazník:

Byl pro vás tento protokol dostatečně srozumitelný?	Ano
Použili jste k testování pluginu README z repozitáře?	Ano
Zobrazila se data při každém volání `Load JSON file`?	Ano
Dávala vám smysl vizualizace alokací a duplicit?	Ano
Používali byste tento nebo podobný plugin pro Javu?	Ano

Proč jste použili/nepoužili k testování README z repozitáře?

Potřeboval jsem se kouknout, jak nainstalovat plugin pomocí .vsix souboru.
Potřeboval jsem se kouknout, jak získat JSON soubory.

Jaký byl váš hlavní dojem z používání pluginu?

spokojený

Co se vám líbilo?

Líbila se mi možnost měnit si barvy prostředí, protože jsem člověk co si potřebuje u programování měnit barvy vzhledu (tj. kódu, UI), aby se mi lépe a pohodlněji programovalo.

Co se vám nelíbilo?

Při zobrazení podrobností o metodě je v nadpisu metoda napsána klasickým textem a ne kurzívou, což ovšem neberu jako chybu, jen jako osobní preferenci.

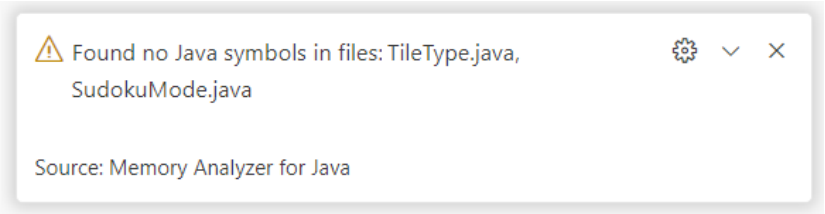
Co byste upravili/přidali/odstranili?

Nejspíše bych přidal možnost změnit text názvů metod v nadpisu podrobností o metodě na kurzívu.

Nalezené chyby, poznámky:

1. změny barvy v nastavení – při zadání `rgb(test, 0, 0)` či `#5555555` např. u "Class Background" zvýraznění backgroundu úplně zmizí; očekával bych, že u neplatně zadané barvy bude nastavená nějaká výchozí barva.

2. načtení datového souboru u Sudoku projektu – při načtení jakéhokoliv datového souboru se mi zobrazí chyba: *Found no Java symbols in files: TileType.java, SudokuMode.java*; ovšem detaily řádek se nejspíše správně zobrazí, takže nevím, jestli se jedná jen o vyhození nějaké výjimky či nikoliv. Každopádně bych očekával, aby se toto upozornění nezobrazovalo, protože nevím, zda je chyba na mé straně.



Zpráva 4

Informace o testerovi:

Studijní obor:	Informační Systémy
Datum testování:	29. 3. 2024
Operační systém:	macOS Sonoma
Verze pluginu (GitLab tag):	0.9.5
Verze VS Code (Help → About):	1.87.2
Jak často používáte VS Code?	několikrát týdně

Dotazník:

Byl pro vás tento protokol dostatečně srozumitelný?		Ne
Použili jste k testování pluginu README z repozitáře?		Ne
Zobrazila se data při každém volání `Load JSON file`?	Ano	
Dávala vám smysl vizualizace alokací a duplicit?	Ano	
Používali byste tento nebo podobný plugin?		Ne

Proč jste použili/nepoužili k testování README z repozitáře?

Nevěděl jsem, že tam je. Nejspíš proto pro mne tento protokol nebyl dostatečný, protože to, co jsem nevěděl, bylo v README.

Jaký byl váš hlavní dojem z používání pluginu?

intuitivní

Co se vám líbilo?

barvy, jednoduchost

Co se vám nelíbilo?

nic

Co byste upravili/přidali/odstranili?

nic

Nalezené chyby, poznámky:

Vše fungovalo, když jsem dělal kroky korektně.

Zpráva 5

1. Informace o testerovi

Studijní Obor:	Informatika a výpočetní technika
Datum testování:	30. 3. 2024
Operační systém:	Windows 11 Pro, 23H2
Verze pluginu:	0.9.5
Verze VS Code:	1.86.2
Jak často používáte VS Code:	Používal jsem jej denně pro web development. Pro Javu jsem ho nikdy nepoužil.

2. Dotazník

1) Byl pro vás tento protokol dostatečně srozumitelný?
Ano, pokud by zde ale nebyl uveden odkaz na soubor README, trvalo by mi dohledání některých informací déle.
2) Použil jste k testování pluginu README z repozitáře?
Ano. Nevěděl jsem, jak se instalují extensions do VS Code mimo integrovaný obchod ve VS Code. Návod na instalaci v README byl srozumitelný.
3) Zobrazila se data při každém volání Load JSON file?
Ano, zkoušel jsem i načíst soubory s jinou příponou a stejnými daty a také to bylo možné. V momentě, kdy jsem vyzkoušel JSON soubor s náhodnými daty, program zobrazil správnou chybovou hlášku.
4) Dávala vám smysl vizualizace alokací a duplicit?
Ano, tato funkce se mi líbila nejvíce.
5) Jaký byl váš hlavní dojem z používání pluginu?
Pozitivní.
6) Používali byste tento nebo podobný plugin?
Ano, ale musel by být lépe integrovaný do vývojového prostředí.

7) Co se vám líbilo?
Líbí se mi, že vizualizátor paměti je přímo integrovaný do vývojového prostředí. Oceňuji jednoduchost a jak fungovalo rozšíření nad celým projektem s více třídami.
8) Co se vám nelíbilo?
Neříkám, že se mi to nelíbí, ale neměl jsem žádnou potřebu měnit barvy CSS v rozšíření. To však neznamená, že by rozšíření nemělo tuto možnost nabízet. Právě naopak si vážím, že autor umožnil uživatelům přizpůsobení podle jejich preference.
9) Co by byste upravili/přidali/odstranili?
Přidal bych souhrn, který by mohl být pravém panelu s výpisem všech problémů, kterým by programátor měl dát svoji pozornost. V aktuální verzi byly zatím zobrazeny jen informace o duplicitních alokacích paměti, včetně jejich odkazu do programu.

3. Nalezené chyby

Chyba č. 1:

Nadpis:	Žádná reakce analyzátoru během editace zdrojového kódu.
Popis	Očekával bych, že analyzátor bude reagovat na moji editaci programu. To se ale neděje. Po smazáni instance zde zůstane řádek s informací o počtu alokovaných instancí.
Postup reprodukce:	1) Načtu projekt sudoku do VS Code, v mém případě Sudoku 2) Spustím příkaz: *Memory analyzer: Load JSON* 3) Smažu libovolný řádek, který obsahuje velikost alokované paměti CHYBA: v kódu zůstala informace o analyzátoru, ale řádek už v programu není.

```
33  public Sudoku(int size) { Total of 888 Bytes
34      super();
35
36      this.setMinWidth(size);
37      this.setMinHeight(size);
38
39      this.setMaxWidth(size);
40      this.setMaxHeight(size);
41
42      final double percentSize = 100.0 / 9;
43
44      for (int i = 0; i < 9; ++i) {
45          RowConstraints rowConst = new RowConstraints(); Total of 432 Bytes in 9 instances, found 9 duplicates
46          rowConst.setFillHeight(true);
47          rowConst.setVgrow(Priority.ALWAYS);
48          rowConst.setValignment(VPos.CENTER);
49          rowConst.setPercentHeight(percentSize);
50          this.getRowConstraints().add(rowConst);
51
52          ColumnConstraints colConst = new ColumnConstraints(); Total of 432 Bytes in 9 instances, found 9 duplicates
53          colConst.setFillWidth(true);
54          colConst.setHgrow(Priority.ALWAYS);
55          colConst.setHalignment(HPos.CENTER);
56          colConst.setPercentWidth(percentSize);
57          this.getColumnConstraints().add(colConst);
58      }
```

Zdrojový kód Sudoku před editací

```
33  public Sudoku(int size) { Total of 888 Bytes
34      super();
35
36      this.setMinWidth(size);
37      this.setMinHeight(size);
38
39      this.setMaxWidth(size);
40      this.setMaxHeight(size);
41
42      final double percentSize = 100.0 / 9;
43  Total of 432 Bytes in 9 instances, found 9 duplicates Total of 432 Bytes in 9 instances, found 9 duplicates
44      initSudokuTiles();
45
46      dataModel.tiles.addListener(this.tileChangeListener);
47
48      this.getStyleClass().addAll("blackBorder"); Total of 24 Bytes
49      this.setStyle("-fx-border-width: 6;");
50  }
```

Zdrojový kód Sudoku po editaci

Obsah přílohy

D

Zde je popsán obsah elektronické přílohy ZIP.

Aplikace_a_knihovny

Ve složce `analyzer` je zdrojový kód analyzátoru se všemi úpravami, které byly popsány v kapitole 5. Zdrojový kód vytvořeného rozšíření se nachází v adresáři `plugin`. Součástí je i soubor `package.json`, který se používá k sestavení projektu. Celý postup instalace pluginu je uveden v uživatelské příručce A.

Text_prace

Adresář obsahuje zdrojový text této bakalářské práce, který je psán v typografickém systému TEX. Součástí jsou všechny soubory nezbytné k jeho přeložení spolu se zdrojovými texty testovacího formuláře a upravených zpráv testerů. Všechny jsou uvedeny v podadresáři `prilohy/prilohy_src`.

Vstupni_data

V tomto adresáři se nachází zkompilovaný analyzátor paměti `analyzer.jar`, instalační soubor pluginu `plugin.vsix` a interaktivní verze testovacího formuláře B s názvem `formular.pdf`. V tom jsou všechny potřebné informace ohledně rozšíření spolu s odkazem na repozitář GitLab, kde se nachází uživatelská příručka.

Dále jsou zde všechny ukázkové Java programy, které jsou umístěné v adresáři `demo`. Každý projekt obsahuje soubor `data.json`, který je možné okamžitě načíst vizualizací. Pro rychlé sestavení programů byl i vložen dávkový soubor `build.bat`, to ale není nutné pro samotné vyzkoušení pluginu. Většina z těchto programů byla upravena a převzata z předchozí práce. Podrobnější informace jsou v sekci 6.5.3.

Readme.txt

Soubor, ve kterém je znění této přílohy.

Bibliografie

[1] VELEK, Jiří. *Monitorování alokací paměti za běhu Java aplikací*. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky, 2023. Dostupné také z: http://hdl.handle.net/11025/53749.

[2] SCHULTS, Carlos. *What Are Java Agents and How to Profile With Them* [online]. 2024-03. [cit. 2024-03-20]. Dostupné z: https://stackify.com/what-are-java-agents-and-how-to-profile-with-them/.

[3] JAIN, Ansu. *Understanding JVM Settings: -Xmx, -Xss, and Java Thread States in Applications* [online]. Medium, 2023-11 [cit. 2024-03-20]. Dostupné z: https://medium.com/@ansujain/understanding-jvm-settings-xmx-xss-and-java-thread-states-in-applications-017f11c1e2b0.

[4] WARD, Matthew; GRINSTEIN, Georges; KEIM, Daniel. *Interactive Data Visualization: Foundations, Techniques, and Applications*. Natick, MA: A. K. Peters, Ltd., 2010. ISBN 978-1-56881-473-5.

[5] TAVARAGI, M.; SUSHMA, C. Colors and Its Significance. *International Journal of Indian Psychology*. 2016, roč. 3, č. 2. ISSN 2348-5396. Dostupné z DOI: 10.25215/0302.126.

[6] CARBON, Claus-Christian. Understanding human perception by human-made illusions. *Frontiers in Human Neuroscience*. 2014, roč. 8. ISSN 1662-5161. Dostupné z DOI: 10.3389/fnhum.2014.00566.

[7] SUGIHARA, Kokichi. Evolution of Impossible Objects. In: ITO, Hiro; LEONARDI, Stefano; PAGLI, Linda; PRENCIPE, Giuseppe (ed.). *9th International Conference on Fun with Algorithms (FUN 2018)*. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018, sv. 100, 2:1–2:8. Leibniz International Proceedings in Informatics (LIPIcs). ISBN 978-3-95977-067-5. ISSN 1868-8969. Dostupné z DOI: 10.4230/LIPIcs.FUN.2018.2.

[8] SCHUSTER, D. H. A New Ambiguous Figure: A Three-Stick Clevis. *The American Journal of Psychology*. 1964, roč. 77, č. 4, s. 673. ISSN 00029556. Dostupné z DOI: 10.2307/1420787.

[9] BIRIHANU, Ermiyas Belachew. Analysis of Software Quality Using Software Metrics. *International Journal on Computational Science & Applications*. 2018, roč. 8, č. 4/5, s. 11–20. ISSN 22000011. Dostupné z DOI: 10.5121/ijcsa.2018.8502.

[10] *Software Testing vs. Analysis* [online]. 2006-04. [cit. 2024-03-26]. Dostupné z: https://jamesmccaffrey.wordpress.com/2006/04/16/software-testing-vs-analysis/.

[11] FEW, Stephen; EDGE, Perceptual. Data visualization effectiveness profile. *Perceptual Edge*. 2017, roč. 10, s. 12. Dostupné také z: https://www.perceptualedge.com/articles/visual_business_intelligence/data_visualization_effectiveness_profile.pdf.

[12] *Grappl - Visual Studio Marketplace* [online]. 2024-03. [cit. 2024-03-30]. Dostupné z: https://marketplace.visualstudio.com/items?itemName=grappl.grappl&ssr=false#overview.

[13] *VisualVM: Features* [online]. [cit. 2024-03-29]. Dostupné z: https://visualvm.github.io/features.html.

[14] HEMANT, K. Mishra. *Data Serialization and Deserialization: What is it?* [online]. Medium, 2023-11 [cit. 2024-03-21]. Dostupné z: https://medium.com/@khemanta/data-serialization-and-deserialization-what-is-it-29b5ca7a756f.

[15] SINGH, Mandeep. *Mastering Data Serialization: A Comprehensive Guide to Efficient Data Exchange and Storage* [online]. Medium, 2023-07 [cit. 2024-03-21]. Dostupné z: https://medium.com/@mndpsngh21/mastering-data-serialization-a-comprehensive-guide-to-efficient-data-exchange-and-storage-60cb9a41c53c.

[16] JOHNS, Robert. *11 Best Java IDE & Code Editors [2024 Update]* [online]. 2024-01. [cit. 2024-04-05]. Dostupné z: https://hackr.io/blog/best-java-ides.

[17] MICROSOFT. *Language Server Extension Guide* [online]. Microsoft, 2021-11 [cit. 2024-04-05]. Dostupné z: https://code.visualstudio.com/api/language-extensions/language-server-extension-guide.

[18] MICROSOFT. *Built-in Commands* [online]. Microsoft, 2021-11 [cit. 2024-04-05]. Dostupné z: https://code.visualstudio.com/api/references/commands.

[19] MICROSOFT. *Run and Debug Java in Visual Studio Code* [online]. Microsoft, 2021-11 [cit. 2024-04-12]. Dostupné z: https://code.visualstudio.com/docs/java/java-debugging#_configuration-options.

[20] MICROSOFT. *Your First Extension* [online]. Microsoft, 2021-11 [cit. 2024-04-26]. Dostupné z: `https://code.visualstudio.com/api/get-started/your-first-extension`.

[21] *Uri scheme isn't normalized · Issue #138397 · microsoft/vscode* [online]. 2021-12. [cit. 2024-04-14]. Dostupné z: `https://github.com/microsoft/vscode/issues/138397`.

Seznam obrázků

Seznam tabulek

Seznam výpisů

Seznam zkratek

JVM – Java Virtual Machine
JSON – JavaScript Object Notation
XML – Extensible Markup Language
IDE – Integrated Development Environment
DRY – Don't Repeat Yourself
SOLID – pětice principů objektového návrhu softwaru
VS Code – Visual Studio Code
API – Application Programming Interface
LSP – Language Server Protocol
VSIX – Visual Studio extension installer
HTML – Hypertext Markup Language
CSS – Cascading Style Sheets