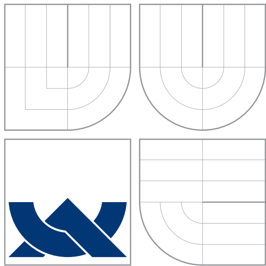
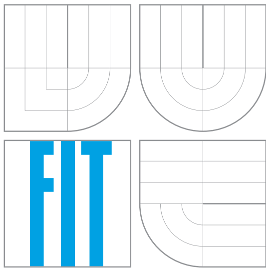


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

NÁVRHOVÉ VZORY V PARALELNÍCH A DISTRIBUOVANÝCH SYSTÉMECH

DESIGN PATTERNS FOR PARALLEL AND DISTRIBUTED SYSTEMS

DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE — Ing. PETER JURNEČKA
AUTHOR

ŠKOLITEL — Doc. Dr. Ing. PETR HANÁČEK
SUPERVISOR

BRNO 2016

Abstrakt

V tejto práci je opísaný návrh spôsobu zápisu a práce s paralelnými návrhovými vzormi, ktorého prínosom je možnosť navrhovania automatických opráv existujúcich paralelných zdrojových kódov pomocou refaktoringu. Na to, aby bolo možné navrhovaný spôsob zápisu využiť je potrebné, aby táto práca pokrývala oblasti statickej analýzy kódu, formálneho zápisu paralelných návrhových vzorov a refaktoringu. Statická analýza kódu umožňuje porozumieť existujúcim paralelným zdrojovým kódom a definovať miesta, kam sa má vložiť návrhový vzor. Formálny zápis návrhového vzoru umožňuje automaticky aplikovať daný vzor do existujúceho zdrojového kódu. Nakoniec refaktoring umožňuje upraviť existujúci zdrojový kód bez zmeny funkčnosti. Prvá časť práce sa venuje popisu súčasného stavu v týchto troch oblastiach t.j. analýze kódu, návrhovým vzorom a refaktoringu. Druhá časť práce sa venuje opisu metodiky a experimentálnemu overeniu jej nasadenia.

Abstract

This Ph.D. thesis describes proposed notation and method for working with parallel design patterns, which allowes proposing of automatic corrections to existing parallel source code with help of refactoring. In order to define the proposed notation, this work must cover areas of static code analysis, formal description of parallel design patterns and refactoring. Static code analysis is used to analyse the existing parallel source code for definition of places where you want to insert specified design pattern. Formal description of design pattern allows you to automatically apply the pattern to the existing source code. Finally, refactoring allows you to edit an existing source code without changing its functionality. The first part is devoted to the description of the current status in these three areas e.g. code analysis, design patterns and refactoring. The second part is devoted to a description of the methodology and experimental verification of its deployment.

Klíčová slova

návrhové vzory, paralelné programovanie, refaktoring, statická analýza, BPSL, spôsob zápisu návrhových vzorov

Keywords

design patterns, parallel programming, refactoring, static analysis, BPSL, notation of design patterns

Citace

Peter Jurnečka: Návrhové vzory v paralelních a distribuovaných systémech, disertační práce, Brno, FIT VUT v Brně, 2016

Návrhové vzory v paralelních a distribuovaných systémech

Prohlášení

Prohlašuji, že jsem tuto dizertační práci vypracoval samostatně pod vedením pana docenta Hanáčka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . .
Peter Jurnečka
31. srpna 2016

Poděkování

Ďakujem môjmu školiteľovi pánovi docentovi Petrovi Hanáčkovi za výborné vedenie počas môjho štúdia, za cenné rady a pripomienky vedúce k úspešnému dokončeniu mojej dizertačnej práce.

© Peter Jurnečka, 2016.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Úvod **5**

1 Motivácia a ciele práce **7**

2 Analýza kódu **9**
2.1 Dynamická analýza . . . 9
2.2 Statická analýza . . . 10
2.3 Metriky kódu . . . 13
2.4 Vyhľadávanie v zdrojovom kóde . . . 15
2.5 Zhrnutie . . . 17

3 Návrhové vzory **18**
3.1 Automatická detekcia návrhových vzorov . . . 19
3.2 Spôsoby zápisu návrhových vzorov . . . 20
3.2.1 Neformálne spôsoby zápisu návrhových vzorov . . . 20
3.2.2 Formálne spôsoby zápisu návrhových vzorov . . . 20
3.3 Vybraná sada paralelných návrhových vzorov . . . 23
3.3.1 Návrhové vzory pre súbežný beh . . . 23
3.3.2 Synchronizačné návrhové vzory . . . 27
3.4 Zhrnutie . . . 33

4 Refaktoring **35**

5 Teoretické východiská práce **38**
5.1 Analýza mechanizmov pre vzájomnú výlučnosť . . . 38
5.1.1 Využitie analýzy zámkov . . . 40
5.2 Metriky pre aplikáciu návrhových vzorov . . . 42
5.2.1 Metriky . . . 44
5.3 Zhrnutie . . . 48

6 Návrh spôsobu zápisu návrhových vzorov **49**
6.1 Analýza kódu . . . 49
6.2 Špecifikácia vzoru . . . 52
6.2.1 Špecifikácia štruktúry . . . 53
6.2.2 Špecifikácia správania . . . 53
6.3 Príklad zápisu paralelného návrhového vzoru . . . 55
6.4 Zhrnutie . . . 56

7 Návrh systému na vkladanie návrhových vzorov do existujúcich paralelných zdrojových kódov 58
7.1 Analýza kódu . . . 58
7.1.1 Analýza toku dát . . . 59
7.1.2 Použitý model . . . 68
7.2 Špecifikácia návrhových vzorov . . . 71
7.3 Vkladanie návrhových vzorov . . . 72
7.4 Príklad zápisu vzoru pomocou navrhnutého systému . . . 72
7.5 Zhrnutie . . . 74

8 Experimentálne overenie pomocou vybraných návrhových vzorov 76
8.1 Thread safe interface . . . 76
8.2 Future . . . 77
8.3 Guarded Suspension . . . 78
8.4 Scoped Locking . . . 79
8.5 Immutable Value . . . 80
8.6 Vyhodnotenie experimentu a zhrnutie . . . 80

9 Záver 82

Literatura 86

Zoznam obrázkov

2.1 Priemerná cena opravy chyby v závislosti na čase jej nájdenia. 11
2.2 Princíp Semantic Based Code Search. 15
2.3 Princíp Test Driven Code Search. 16

3.1 Špecifikácia vzoru Singleton v jazyku SPINE. 21
3.2 Vzťahy návrhových vzorov Half-Sync/Half-Async a Leader/Followers. . . . 24
3.3 Vzťahy návrhových vzorov Active Object a Monitor Object. 24
3.4 Návrhový vzor Half-Sync/Half-Async. 25
3.5 Návrhový vzor Leader/Followers. 26
3.6 Návrhový vzor Active Object. 26
3.7 Návrhový vzor Monitor Object. 27
3.8 Vzťahy návrhových vzorov Guarded Suspension a Future. 28
3.9 Vzťahy návrhových vzorov Thread-Safe Interface, Strategized Locking a Scoped Locking. 28
3.10 Návrhový vzor Guarded Suspension. 29
3.11 Návrhový vzor Future. 30
3.12 Návrhový vzor Thread-Safe Interface. 30
3.13 Návrhový vzor Strategized Locking. 31
3.14 Návrhový vzor Scoped Locking. 31
3.15 Návrhový vzor Thread-Specific Storage. 32
3.16 Návrhový vzor Copied Value. 32
3.17 Návrhový vzor Immutable Value. 33

5.1 Analýza zámkov jednoduchého programu. 39
5.2 Cyklická závislosť medzi zámkami. 40
5.3 Chýbajúci zámok. 41
5.4 Veľmi všeobecná synchronizácia. 42
5.5 Príklad podmieneného príkazu 45
5.6 Vzťahy medzi metrikami a návrhovými vzormi. 47

6.1 Notácia použitá v popise algoritmu analýzy kódu. 50
6.2 Definícia zamykania použitého zámku. 52

7.1 Príklad grafu toku riadenia s príslušným zdrojovým kódom. 59
7.2 Viacero hrán vychádzajúcich z vrcholu obsahujúceho podmienený príkaz `if`. 59
7.3 Viacero hrán smerujúcich do vrcholu grafu. 60
7.4 Algoritmus intro-procedurálnej analýzy toku dát 60
7.5 Príklad analyzovaného kódu. 61

7.6 Graf toku riadenia analyzovaného kódu z obrázka 7.5 po prvom prechode algoritmu. . . . 62
7.7 Graf toku riadenia analyzovaného kódu z obrázka 7.5 po druhom prechode algoritmu. . . . 63
7.8 Inter-procedurálna analýza toku dát. . . . 64
7.9 Algoritmus inter-procedurálnej analýzy toku dát 64
7.10 Príklad analyzovaného kódu. . . . 65
7.11 Graf toku riadenia analyzovaného kódu z obrázku 7.10. . . . 65
7.12 Graf toku riadenia analyzovaného kódu z obrázku 7.10. . . . 66
7.13 Graf toku riadenia analyzovaného kódu z obrázku 7.10, po prvom prechode inter-procedurálnej analýzy. . . . 66
7.14 Graf toku riadenia analyzovaného kódu z obrázku 7.10, po prvom prechode inter-procedurálnej analýzy. . . . 67
7.15 Graf toku riadenia analyzovaného kódu z obrázku 7.10, po druhom prechode inter-procedurálnej analýzy. . . . 67
7.16 Graf toku riadenia analyzovaného kódu z obrázku 7.10, po druhom prechode inter-procedurálnej analýzy. . . . 68
7.17 Diagram objektov syntaktického stromu rozšíreného o informácie o vláknach a typoch vytvorený pomocou analýzy kódu. . . . 69
7.18 Príklad špecifikácie vzoru. . . . 73
7.19 príklad analyzovaného kódu. . . . 74

8.1 Návrhový vzor Thread - Safe Interface. . . . 76
8.2 Kód dopytu pre návrhový vzor Thread - Safe Interface. . . . 77
8.3 Návrhový vzor Future. . . . 77
8.4 Kód dopytu pre návrhový vzor Future. . . . 78
8.5 Návrhový vzor Guarded Suspension. . . . 78
8.6 Kód dopytu pre návrhový vzor Guarded Suspension 79
8.7 Scoped Locking design pattern. . . . 79
8.8 Kód dopytu pre návrhový vzor Scoped Locking. . . . 79
8.9 Návrhový vzor Immutable Value. . . . 80
8.10 Kód dopytu pre návrhový vzor Immutable Value. . . . 80

Úvod

Programovanie paralelných alebo viacvláknových aplikácií sa čím ďalej tým viac rozširuje. Nové technológie, ako sú viacjadrové procesory alebo masívne paralelné procesory grafických kariet sa stali široko dostupnými a použiteľnými aj v bežných počítačoch. Programovanie paralelných systémov však kladie vyššie nároky na znalosti programátorov a tieto vyššie nároky sa ešte násobia pri údržbe a úpravách existujúcich projektov.

Medzi oblasti, v ktorých môže mať každá chyba fatálne následky, patrí letectvo alebo medicína. Bezpečnostné štandardy [24][6] majú v letectve dôležitú úlohu, pretože aj malé zlyhania, môžu mať fatálne následky. Keď hovoríme o softvéri v oblasti letectva, máme hlavne na mysli softvér pre avioniku. Jedná sa o termín používaný pre elektronické systémy používané v prostredí letectva, názov je odvodený od slov letectvo a elektronika. Príklady elektronických systémov používaných v letectve sú systémy riadenia letu (autopilot), navigačné systémy alebo antikolízne systémy. Bezpečnosť softvéru je neoddeliteľnou súčasťou bezpečnosti celého systému.

V medicíne je bezpečnosť zabezpečená pomocou Food and Drug Administration (FDA) validačných štandardov [39] [7], ktorých účelom je posúdenie a validácia softvéru v lekárskych zariadeniach. Normy odporúčajú integráciu správy životného cyklu a riadenia rizík počas vývoja. Vývojár konkrétneho softvéru by si mal stanoviť špecifický prístup a úroveň úsilia, ktoré sa použijú na základe týchto noriem. Na druhú stranu, FDA validačné štandardy neodporúčajú nejaké konkrétne modely životného cyklu a ani špecifické techniky.

Zabránenie chybám je hlavným cieľom softvérových štandardov v spomínaných oblastiach. Jednou z možností ako uľahčiť programátorom prácu, je používanie návrhových vzorov. V súčasnej dobe bolo urobené veľa výskumu v oblasti návrhových vzorov a automatického refaktoringu zdrojových kódov. Avšak dané výskumy sa nevenovali návrhovým vzorom paralelných a distribuovaných systémov.

Spoločnou požiadavkou všetkých týchto štandardov je požiadavka na spoľahlivosť, ktorú možno dosiahnuť pomocou návrhových vzorov. V tejto práci je navrhnutý spôsob zápisu paralelných návrhových vzorov, ktorý umožní ich vkladanie do existujúcich paralelných zdrojových kódov. Navrhovaný spôsob zápisu detekuje nesprávne používanie súbežnosti a synchronizácie a doporučuje vhodné riešenie pomocou príslušného návrhového vzoru. Tento systém je založený na statickej analýze kódu slúžiacej na vyhľadávanie v kóde a formálnom opise paralelných návrhových vzorov.

Hlavným cieľom tejto práce je pomocou kombinácie existujúcich techník a metodík vytvoriť novú metodiku určenú na zápis paralelných návrhových vzorov, ktorá bude vyu-

žiteľná na automatické vkladanie návrhových vzorov do existujúcich zdrojových kódov. Na to, aby bolo možné vytvoriť takúto metodiku je nutné odpovedať na nasledujúce 3 otázky:

1. Ako určiť miesto, kde treba vložiť vzor – analýza kódu.
2. Ako daný vzor reprezentovať, aby ho bolo možné vložiť do kódu – špecifikácia / definícia vzoru.
3. Pomocný problém: ako daný vzor vložiť do kódu – refaktoring.

Práca je **štruktúrovaná** do ôsmich kapitol. V prvej kapitole je uvedená motivácia a ciele práce. Druhá kapitola sa zaoberá prehľadom súčasného stavu poznania v problematike analýzy kódu. Je zameraná na techniky, ktoré sú využiteľné pri plnení cieľov tejto práce. Tretia kapitola prináša prehľad súčasného stavu výskumu návrhových vzorov, konkrétne popisuje spôsoby zápisu návrhových vzorov, spôsoby detekcie návrhových vzorov v existujúcom kóde a v neposlednom rade opisuje vybranú podmnožinu paralelných návrhových vzorov, ktoré boli použité na pokusy. Štvrtá kapitola sa v krátkosti venuje tématike refaktoringu, ktorý môže byť použitý na automatické vkladanie návrhových vzorov do existujúcich zdrojových kódov. Téma refaktoringu je spomenutá len okrajovo a daná kapitola naznačuje možnosti tejto techniky z pohľadu tejto práce. Zameriava sa na jak na formálne neopísateľné spôsoby refaktoringu, tak aj na formálne opísateľné, ktoré sa dajú automatizovať pomocou programu a sú využiteľné pri automatickom vkladaní návrhových vzorov do existujúcich zdrojových kódov.

Piata kapitola sa venuje teoretickým východiskám práce, ktoré slúžili ako inšpirácia pre analýzu kódu popísanú v jadre práce. V šiestej kapitole je riešené jadro práce. Na začiatku je opísaný vybraný spôsob analýzy kódu a v druhej polovici je opísaný jazyk použitý na špecifikáciu návrhových vzorov, ktorý umožňuje automaticky doporučovať vhodné návrhové vzory do existujúcich zdrojových kódov. Siedma kapitola obsahuje návrh systému určeného na vkladanie návrhových vzorov do existujúcich zdrojových kódov, zameriava sa na kľúčové časti použitia navrhnutej metodiky v praxi.

Ôsma kapitola ukazuje praktické využitie výsledkov výskumu na vybraných návrhových vzoroch, čím demonštruje použiteľnosť navrhnutej metodiky. Posledná kapitola je záver, ktorý sumarizuje prezentovaný výskum a navrhuje ďalšie kroky a smerovanie možného výskumu.

Kapitola 1

Motivácia a ciele práce

Základnou myšlienkou dizertačnej práce je využiť formálne popísané paralelné návrhové vzory pri navrhovaní automatických úprav existujúcich paralelných zdrojových kódov. Cieľom dizertačnej práce je potom vytvoriť systém na automatickú pomoc programátorom s refaktoringom zdrojových kódov paralelných systémov do tvaru vychádzajúceho z návrhových vzorov. To znamená upraviť existujúci paralelný zdrojový kód tak, aby bol vytvorený na základe návrhových vzorov a tým pádom efektívnejší a jednoduchšie upravovateľný. Na vytvorenie takéhoto systému treba spojiť a prípadne upraviť pre podporu paralelných algoritmov: statickú analýzu a spôsob zápisu návrhových vzorov. Pri úpravách existujúcich zdrojových kódov môže pomôcť refaktoring, ale ten nieje predmetom podrobného skúmania v tejto práci.

Ako už bolo spomenuté v úvode, hlavným cieľom tejto práce je pomocou kombinácie existujúcich techník a metodík vytvoriť nový spôsob zápisu paralelných návrhových vzorov, ktorý bude využiteľný na automatické vkladanie návrhových vzorov do existujúcich zdrojových kódov. V súčasnosti neexistuje výskum, ktorý by sa venoval automatickému vkladaniu paralelných návrhových vzorov do existujúcich paralelných zdrojových kódov za účelom zvýšenia kvality kódu. Na to, aby bolo možné vytvoriť nový spôsob zápisu musíme odpovedať na nasledujúce 3 otázky:

1. Ako určiť miesto, kde treba vložiť vzor.
2. Ako daný vzor reprezentovať, aby ho bolo možné vložiť do kódu.
3. Pomocný problém: ako daný vzor vložiť do kódu.

Problém ako určiť miesto, kde by bolo vhodné vložiť návrhový vzor je riešený pomocou navrhnutého Searchable Code Modelu, ktorý je plnený pomocou statickej analýzy. Problematika statickej analýzy je popísané v kapitole 2, ktorá sa zaoberá prehľadom súčasného stavu. Ďalej v kapitole 5 sú podrobnejšie preskúmané dve techniky, ktoré stoja za nápadom použiť statickú analýzu na definovanie miest, kam by sa mal vložiť návrhový vzor a za nápadom využitia rozšírenej statickej analýzy na zisťovanie informácií o vláknach a ich synchronizácii. Využitie statickej analýzy je opísané v jadre dizertačnej práce, presnejšie v podkapitolách 6.1 a 7.1.

Druhý problém, t.j. ako daný vzor reprezentovať, je riešený pomocou využitia existujúcich jazykov s ich miernym rozšírením. Existujúce jazyky na zápis návrhových vzorov sú spomenuté v kapitole 4, ktorá sa venuje všetkým témam spojeným s návrhovými vzormi. V jadre práce je opísaný navrhnutý spôsob zápisu paralelných návrhových vzorov, konkrétne v podkapitolách 6.2 a 7.2. Navrhnutý spôsob zápisu definuje návrhový vzor ako dvojicu, ktorá dokáže popísať jak miesto, kam je vhodné daný vzor vložiť, tak aj vzor ako taký.

Tretí problém samotného refaktoringu je riešený okrajovo, aby bolo možné tému uzavrieť, keďže sa jedná o pomerne široký problém, ktorý je v kontexte tejto práce iba dodatočný. Možnosti refaktoringu sú zjednodušene opísané v kapitole 4. Využitie refaktoringu v našom kontexte je popísané na konci kapitol 6 a 7.

Práca je delená do troch logických celkov: úvod do problematiky a popis súčasného stavu pre všetky tri riešené problémy. Kapitola 5 je prechod medzi prehľadom súčasného stavu a navrhnutého riešenia. Od kapitoly 6 ďalej práca popisuje navrhnutý spôsob zápisu paralelných návrhových vzorov, čo je vlastne jadro a hlavný prínos tejto práce.

V nasledujúcej kapitole sa nachádza prvý logický celok opisujúci súčasný stav a úvod do problematiky analýzy kódu, návrhových vzorov a refaktoringu. Analýza kódu umožňuje v definovať miesta, na ktoré je vhodné vložiť návrhový vzor.

Kapitola 2

Analýza kódu

Cieľom tejto kapitoly je uviesť základný prehľad k analýze kódu. Kapitola sa hlavne sústredí na prehľad rôznych prístupov iných autorov. Analýza softvéru sa využíva na sledovanie správania programu, prípadne na získanie užitočných informácií o programe. V podstate sa jedná o proces automatického odvodzovania vlastností správania určitého programu [38]. Tieto vlastnosti môžu zahŕňať tok dát, využitie pamäte, volanie funkcií a pod. V rámci analýzy sa používa množstvo rôznych techník, z ktorých každá používa iný prístup a vedie ku skúmaniu iných vlastností. Podľa základnej povahy týchto techník sa analýza programov delí na dve hlavné časti - dynamickú a statickú analýzu. Okrem dynamickej a statickej analýzy sa kapitola venuje aj témam metrík kódu a vyhľadávaniu v zdrojových kódoch.

Vybrané témy statickej analýzy a metrík sú podrobnejšie opísané v kapitole 5. Daná kapitola prináša detaily vybraných článkov pri ktorých opisuje ich riešenie, ktoré ďalej slúži ako základ pre navrhovanú metodiku opísanú kapitolách neskôr.

2.1 Dynamická analýza

Dynamická analýza je založená na spúšťaní analyzovaného programu (väčšinou binárneho kódu). Môže sa jednať iba o jedno spustenie (napr. pri technikách určených na získanie štatistík o programe), alebo o opakované spúšťanie analyzovaného programu. V takom prípade sú výsledné vlastnosti odvodené analýzou všetkých získaných behov [38]. Na rozdiel od statickej analýzy, ktorá obsiahne všetky možné behy programu, je tá dynamická limitovaná množinou reálne vykonaných behov. To môže znamenať v určitých prípadoch nevýhodu, keď je dynamická analýza chybná, pretože nepokryla všetky rôzne možnosti. Na druhej strane má táto skutočnosť aj svoju výhodu, keďže nikdy nedôjde k problému analýzy falošných poplachov (angl. false positives, niekedy aj false alarms), teda chybných varovaní analyzátoru o možnej chybe, ktoré sa môžu vyskytnúť v statickej analýze [14]. Ďalšou z výhod dynamickej analýzy je, že nie je nutné vytvárať nijakú abstrakciu, takže nemusí dôjsť ku strate informácií (aj keď vo väčšine prípadov sa abstrakcia robí kvôli zjednodušeniu a urýchleniu) [14].

Hoci sa v rámci dynamickej analýzy používa veľké množstvo techník, je možné ich rozdeliť do troch základných skupín. Toto delenie uvažuje množstvo behov programu, ktoré

daná technika používa pri analýze a prístup k vytváraniu týchto behov. Tieto tri skupiny spolu s dedukciou, používanou v statickej analýze, tvoria kompletné delenie analýzy software [82].

Analýza pozorovaním: Táto technika používa iba jediný beh programu a umožňuje preskúmať jeho ľubovoľné aspekty. Existuje veľké množstvo nástrojov využívajúcich danú techniku, väčšinou sa jedná o ladiace programy, tzv. "debuggery". Ďalšími sú napríklad programy navrhnuté na kontrolu využívania pamäte, alebo kontrolu porušenia hraníc poľa [82].

Analýza indukciou: Indukcia je prechod od konkrétneho ku všeobecnému. V analýze sa využíva na zhrnutie viacerých behov do určitej formy abstrakcie (napríklad grafu). Táto technika pracuje s viacerými behmi programu získanými pomocou jeho opätovného spúšťania s rôznymi vstupmi. Využívajú ju napríklad nástroje na pokrytie kódu, ktoré spájajú príkazy a vetvy programu do výsledku, ktorý je možné vizualizovať [82].

Analýza experimentovaním: Hoci predchádzajúce metódy sú schopné sledovať správanie programu, ani jedna z nich nedokáže nájsť príčinu tohoto správania. Na to je nutné použiť posledný typ dynamickej analýzy, ktorý okrem toho, že používa viacero behov, tieto behy priamo ovláda. Vďaka tomu dokáže vytvoriť sériu experimentov, pomocou nich izolovať a nájsť príčiny daného správania [82].

2.2 Statická analýza

Statická analýza [45] je založená na analýze v čase kompilácie, takže nepotrebuje aby bol zdrojový kód spustiteľný. Existuje mnoho rôznych prístupov k statickej analýze od pomerne jednoduchých, ktoré hľadajú kód podľa vzorov opisujúcich nesprávne postupy po pomerne zložité a niekedy aj úplné analýzy. Medzi najznámejšie metódy statickej analýzy patrí analýza toku dát (data-flow analysis). Abstraktná interpretácia a model checking sú niekedy považované za súčasť statickej analýzy.

Statická analýza, na rozdiel od testovania a dynamickej analýzy, nie je obmedzená na posúdenie správania programu na základe jeho behu. Môže teoreticky pokryť všetky možné správania programu. Statická analýza musí bojovať s exponenciálnym počtom možných scenárov plánovania, čo činí analýzu viacvláknových programov pomerne ťažkú. Z tohto dôvodu existujú rôzne statické analýzy, ktoré využívajú iba približné správanie vlákien. Čím viac aproximácie používajú, tým väčšie množstvo zdrojového kódu dokážu analyzovať, ale za cenu viac falošných poplachov. Úlohy riešené statickou analýzou môžeme deliť do troch kategórii:

1. Detekcia chýb v programe.
2. Odporúčania pre formátovanie kódu. Niektoré statické analyzátory umožňujú skontrolovať, či zdrojový kód zodpovedá formátovacím normám vo firmách.
3. Výpočet metrík kódu. Softvérové metriky sú zobrazenia, ktoré umožňujú získať číselnú hodnotu nejakej vlastnosti softvéru alebo jeho špecifikácie.

Existujú aj iné spôsoby využitia statických analytických nástrojov. Napríklad, statická

analýza môže byť použitá ako metóda pre riadenie a učenie nových pracovníkov, ktorí ešte nie sú dostatočne oboznámený s programovacími pravidlami vo firme.

Rovnako ako akákoľvek iná metóda detekcie chýb, tak aj statická analýza má svoje silné a slabé stránky. Hlavnou výhodou statickej analýzy je, že umožňuje významne znížiť cenu za elimináciu chýb v softvéri. Čím skôr je zistená chyba, tým nižšia je cena opravy. Podľa údajov uvedených v knihe "Code Complete"[57], chyby nájdené vo fáze testovania stoja desaťkrát viac ako tie ktoré sa nájdu počas fázy písania kódu.

Čas vzniku	Čas detekcie				
	Požiadavky	Architektúra	Implementácia	Test	Vydanie
Požiadavky	1	3	5 - 10	10	10 - 100
Architektúra		1	10	15	25 - 100
Implementácia			1	10	10 - 25

Analýza kódu

Obrázok 2.1: Priemerná cena opravy chyby v závislosti na čase jej nájdenia.

Nástroje statickej analýzy umožňujú rýchlo odhaliť veľa chýb vo fáze programovania, čo výrazne znižuje náklady na vývoj celého projektu.

Medzi ďalšie výhody statickej analýzy kódu patrí plné pokrytie zdrojového kódu. Statické analyzátory dokážu skontrolovať aj tie fragmenty kódu, ktoré sa dostanú k riadeniu veľmi zriedka. Tieto fragmenty kódu zvyčajne nie je možné testovať pomocou iných metód. To vám umožní nájsť chyby v ošetrení výnimiek alebo v logovacom systéme.

Statická analýza nezávisí na prekladači, ktorý používate a ani na prostredí, kde bude zostavený spúšťaný program. To umožňuje nájsť skryté chyby, ktoré sa môžu prejaviť po niekoľkých rokoch používania výslednej aplikácie. Napríklad sa môže jednať o chyby nedefinovaného správania. Takéto chyby môžu nastať pri prechode na inú verziu prekladača alebo pri použití iného optimalizátora kódu. Iným zaujímavým príkladom skrytých chýb je prepis vlastnej pamäte (buffer overflow).

Statický analyzátor upozorňuje na podozrivé fragmenty kódu, aj keď kód môže byť v skutočnosti úplne správny. To sa nazýva falošné alarmy. Iba programátor môže pochopiť, ak analýza ukazuje na skutočnú chybu, alebo je to len falošný poplach. Nutnosť preskúmania falošných poplachov zaberá pracovný čas a oslabuje pozornosť pred tými fragmentami kódu, ktoré v skutočnosti obsahujú chyby.

Chyby zistené pomocou statických analyzátorov sú pomerne rôznorodé. Niektoré analyzátory sa zameriavajú na určitú oblasť alebo typ poruchy, zatiaľ čo iné podporujú určité normy, napríklad Misra-C: 2004 [58], pravidlá Sutter-Alexandrescu [76], atď.

Oblasť statickej analýzy sa aktívne rozvíja. Objavujú sa nové diagnostické pravidlá a normy, zatiaľ čo niektoré pravidlá sa stávajú zastarané. To je dôvod, prečo nedáva zmysel porovnávanie analyzátorov na základe chýb, ktoré môžu detekovať. Jediný spôsob, ako porovnať rôzne nástroje je ich kontrola nad súborom projektov a spočítanie počtu skutočných chýb, ktoré dokázali detekovať.

Existujúce nástroje na statickú analýzu

Existujú rôzne spôsoby, ako zabezpečiť kvalitu softvéru, vrátane revízie kódu a dôkladného testovania. Chyby softvéru môžu stáť spoločnosti značné množstvo peňazí, najmä keď vedú k zlyhaniu softvéru [57]. Statické analytické nástroje poskytujú prostriedky pre analýzu kódu, bez toho aby musel byť daný kód spustený, čo pomáha zaistiť kvalitnejší softvér v celom procese vývoja. Existuje celý rad spôsobov, ako vykonávať automatickú statickú analýzu [21].Po spustení vývojárom, neustále pri písaní kódu vo vývojovom prostredí, alebo tesne predtým, než sa softvér odošle do systému pre správu verzií. Tieto nástroje umožňujú vývojárom nakonfigurovať, aké druhy chýb majú hľadať, a niekedy dokonca umožňujú definovať nové chybové vzory.

FindBugs [48] je detektor chybových vzorov pre Javu. FindBugs používa množinu adhoc techník, ktorých cieľom je vyvážiť presnosť, efektivitu a použiteľnosť. Jednou z hlavných techník, ktoré FindBugs používa, je syntaktické porovnávanie zdrojových kódov so známymi podozrivými kódmi.

FindBugs je statický analytický nástroj, ktorý skúma triedy alebo súbory JAR. Hľadá potenciálne problémy tým, že porovnáva zostavené aplikácie so zoznamom chybových vzorov. FindBugs používa vzor Visitor na realizáciu functionalít jeho detektorov vzorov.

Napríklad FindBugs kontroluje, či volanie wait(), ktoré sa používa u viacvláknových programov v jazyku Java, je vždy volané v cykle, ktoré je správnym použitím vo väčšine prípadov. V niektorých prípadoch FindBugs tiež používa analýzu toku dát (dataflow) pre kontrolu chýb. Napríklad FindBugs využíva jednoduchú intraprocedurálnu (v rámci jednej metódy) analýzu toku dát na kontrolu NULL ukazovateľov. FindBugs môže byť rozšírený pomocou pridania vlastných detektorov napísaných v Jave.

JLint [4] je open source nástroj na statickú analýzu kódu, ktorý uľahčuje kontrolu kódu a hľadanie chýb v bytekóde jazyka Java, chyby a problémy so synchronizáciou pomocou analýzy toku dát a vytvárania grafu zámkov. Jlint bol vyvinutý Konstantinom Knizhnikom a ďalej rozšírený Cyrille Arthom o dôkladnejšiu kontrolu synchronizácie. Jlint sa skladá z dvoch rôznych programov určených na kontrolu syntaxe a sémantiky. Sémantický verifikátor Jlint predovšetkým extrahuje informácie zo súborov tried Java a využíva informácie o ladení na asociáciu hlásených chýb so zdrojovými kódmi. Vzhľadom k tomu, Java väčšinou dedí C / C ++ syntax, je teda Jlint je schopný overiť syntax pre všetky jazyky z rodiny C jazykov, ako je C, C ++, Objective-C atď. Pôvodne bol tento program nazvaný AntiC, pretože opravoval väčšinu problémov s C gramatikou, ako sú chyby operátorov priority, absencia príkazu break v kóde príkazu switch.

PMD [5], rovnako ako FindBugs a JLint, vykonáva syntaktickú kontrolu zdrojového kódu programu, ale bez analýzy dátového toku. Okrem detekcie jasne chybného kódu, mnohé z "chýb"ktoré PMD vyhľadá sú štylistické konvencie, ktorých porušenie môže byť podozrivé za určitých okolností. Napríklad, príkaz try s prázdnym blokom catch môže znamenať, že zachytená chyba je nesprávne ošetrená. Vzhľadom k tomu, PMD zahŕňa mnoho detektorov chýb, ktoré sú závislé na štýle programovania. PMD poskytuje možnosť výberu, ktorý detektor alebo skupiny detektorov by mal byť spustený. PMD je ľahko rozšíriteľný pomocou nových detektorov chybových vzorov ktoré môžu byť napísané buď pomocou Java alebo XPath

Checkstyle [3] vytvára syntaktický strom zo zdrojového kódu jazyka Java a vyvolá submoduly nazvané kontroly, pri priechode uzlami stromu. Každý uzol syntaktického stromu definuje token. Návšteva uzla počas priechodu spustí všetky kontroly, ktoré sú konfigurované pre daný token. Napríklad v prípade, že kontrola MethodLength bola nakonfigurovaná ako submodul, potom návšteva uzla s metódou alebo token definície konštruktora spúšťa MethodLength na kontrole počtu riadkov kódu bloku uzla.

Niektoré kontroly, ako FileLength a LineLength sa aplikujú priamo na zdrojové súbory a nezahŕňajú tokeny zo syntaktického stromu. Ostatné kontroly sú spojené s nastaviteľnými sadami tokenov, ktoré vedú ku kontrolám.

StyleCop je voľne šíriteľný statický analyzátor zdrojových kódov pre C# vývojárov, ktorý bol pôvodne vyvinutý spoločnosťou Microsoft. Riadenie a koordinácia projektu StyleCop je riadený .NET komunitou. StyleCop je dobre integrovaný do Visual Studia a varuje vývojárov, ak nenasledujú štandardy jazyka. Kódovacie štandardy sú určené na zlepšenie čitateľnosti, konzistencie, a udržateľnosti. StyleCop je statický analytický nástroj, ktorý poskytuje vývojárom efektívny spôsob, ako sledovať štandard kódovania pomocou definície široko používaného C# štandardu programovania. C# štandard, ktorý definuje StyleCop je široko používaný a mnoho vývojárov ho používa. Štandardy sú o definícii štýlu programovania a písania kódu. Zámerom zriadenia a dodržiavania štandardov písania kódu je, aby bol zdrojový kód čitateľnejší a jednoduchšie udržiavateľný

2.3 Metriky kódu

Softvérové metriky poskytujú prostriedky na získanie užitočných a merateľných informácii o štruktúre softvérového systému. To vysvetľuje, prečo sa prvé metriky, ako je LOC (počet riadkov kódu) objavil veľmi skoro. Dnešný softvérový inžinieri majú k dispozícii veľmi rozsiahly zoznam metrík poskytujúci možnosť získať potrebný prehľad na pochopenie a vyhodnotenie štruktúry a kvality systému. Rozvoj metrík priniesol potrebu určiť, ktoré metriky sú najvhodnejší pre ktoré prostredie [81]. Populárne metriky sú Halsteadove merania zložitosti [46], McCabeho Cyklomatická zložitosť [56] a index udržiavateľnosti. Pre objektovo orientované systémy sa najčastejšie používajú metriky navrhnuté Henrym a Kafuramom , Chidamber a Kemererom a inými.

Niekoľko štúdií sa pokúša spojiť softvérové metriky s kvalitou [73, 74], alebo potvrdiť dôležitosť jednotlivých ukazovateľov navrhnutých v literatúre [22]. Iní používajú metriky na predpoklad náchylnosti tried k chybám už v štádiu skorého vývoja, alebo na hodnotenie ich vplyvu na údržbu a úsilie potrebné na udržanie systému v aktuálnom stave s meniacimi sa požiadavkami. V oboch prípadoch sú metriky ukazovateľmi kvality systému. Tieto metódy sa používajú na spoľahlivú detekciu častí systému, ktoré sú najväčšou pravdepodobnosťou zdrojom chýb alebo majú náročnú údržbu, a preto je potrebná ich úprava.

Pravdepodobne najvýznamnejším faktorom ovplyvňujúcim kvalitu softvéru je jeho dizajn. Dobrý dizajn umožňuje vyvinúť softvérový systém s minimálnym úsilím a za menej peňazí. Objektovo orientované systémy umožňujú jednoduchšie definovať toto správanie, pretože oproti procedurálnym jazykom ponúkajú mnoho silnejšie mechanizmy, ako je dedičnosť, polymorfizmus a zapúzdrenie. Preto už podľa hodnotenia kvality návrhu systému, môžeme odhadnúť kvalitou celého výsledného systému. Rôzne štúdie sa pokúšajú zladiť

atribúty návrhu systému (často reprezentované pomocou metrík zložitosti návrhu)s jeho kvalitou. Tieto štúdie sa predovšetkým zameriavajú na hustotu závad a čas alebo náročnosť údržby a poskytujú prediktívne modely založené na hodnotách metriky pre tieto atribúty [15]. Iní definujú formálne modely pre objektovo orientovaný návrh a používajú ich na formálnu definíciu existujúcich objektovo orientovaných metrík za účelom zjednodušenia hodnotenia automatizovaného návrhu [27, 69]. Je teda zrejmé, že metriky môžu byť použité na detekciu kvality systémov, potrebu ich úprav, ako aj určenie, ktoré časti je potrebné preprogramovať.

Práca Murakiho [61] predstavuje určitý druh metrík návrhu, ktoré pomáhajú správne určiť použitie návrhových vzorov pri refaktoringu. Refaktoring pomocou návrhových vzorov je jedným zo sľubných prístupov k zlepšeniu návrhu v priebehu vývoja. Zásadnou otázkou je zistiť, kedy, kde a ktoré návrhové vzory použiť. Ako je známe, niektoré softvérové metriky, vyjadrujú kvalitu návrhu len z určitých pohľadov, napríklad súdržnosť, podobnosť, veľkosť, štrukturálna zložitosť, vnorené cykly a podobne. Pokiaľ ide o objektovo orientovaný návrh, existuje niekoľko ďalších užitočných metrík, napríklad CK metriky [29], ktoré počítajú počet metód v triede, hĺbku stromu dedičnosti, pomer počtu atribútov, ktoré sú zdieľané s verejnými triedami a tak ďalej. Tieto metriky vyjadrujú len povrchnú zložitosť statickej štruktúry návrhu, popisujú veľkosť projektu, avšak nič nehovoria o rozšíriteľnosti, alebo opätovnej použiteľnosti tried. Výsledok použitia návrhových vzorov môže viesť k nižším hodnotám týchto metrík, tj k návrhom nižšej kvality, pretože návrhové vzory podporujú pridávanie abstraktných tried a dedičnosti.

Metriky pre aplikáciu návrhových vzorov Muraki analyzoval niekoľko reálnych objektovo orientovaných návrhov s nízkou kvalitou ktoré bolo treba refaktorovať. Vo svojej práci sa zameral na vlastnosti podmienených príkazov, metód a dedičnosti, ktoré sa zdajú byť príčinou nízkej kvality daného návrhu. Navrhol 20 metrík, objektívne zisťujúcich problematické charakteristiky v objektovo orientovaných návrhoch. Tieto metriky vyjadrujú zložitosť vetvenia v podmienených príkazoch a silu závislosti medzi podtriedami v strome dedičnosti. Tieto metriky môžu návrhárom pomôcť pri rozpoznaní, kedy, kde a aké návrhové vzory by mali byť použité. Svoj prístup aplikoval na návrh editora obrázkov ktorý bol navrhnutý začiatočníkom. Na danom editore overil efektívnosť navrhovaných metrík. Táto práca je detailnejšie opísaná v kapitole 5.

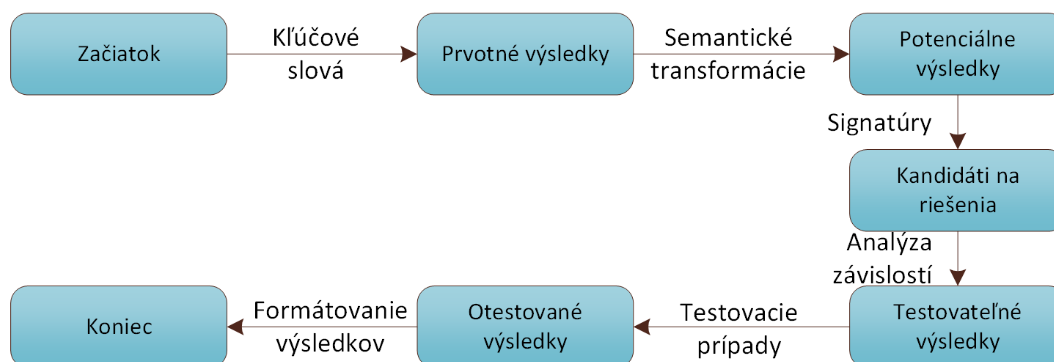
Muraki sa vo svojej práci zameral na konkrétne metriky kódu, ktoré môžu charakterizovať špecifické aspekty návrhu softvéru. Ako je dobre známe, niektoré metriky kódu vyjadrujú kvalitu návrhu zo špecifických pohľadov, napr. súdržnosť, previazanosť, podobnosť, veľkosť, zložitosť, vnorené cykly a tak ďalej. Čo sa týka objektovo orientovaného návrhu, máme niekoľko užitočných metrík, napríklad CK metriky [29], ktoré počítajú počet metód na triedu, hĺbku stromu dedičnosti, pomer množstva atribútov, ktoré sú zdieľané s verejnými triedami a iné. Tieto metriky vyjadrujú len vonkajšiu zložitosť návrhu, ale neopisujú zložitosť a znovupoužiteľnosť a ani rozšíriteľnosť kódu. Výsledok použitia návrhových vzorov môže viesť k nižším hodnotám týchto metrík, tj k návrhom nižšej kvality, pretože niektoré návrhové vzory nás nútia pridať abstraktné triedy a dedia nejaké abstraktné vlastnosti z nich do konkrétnych tried.

2.4 Vyhľadávanie v zdrojovom kóde

Tretím možným prístupom k definovaniu miesta kam sa má vložiť návrhový vzor je vyhľadávanie v zdrojovom kóde. Všetky tieto prístupy pomocou na statickej analýzy zdrojového kódu, vytvoria model daného kódu, ktorý je uložený v databáze nad ktorou sa potom vykonávajú vyhľadávacie príkazy. Účelom nasledujúcej kapitoly je zoznámiť čitateľa s touto tematikou, keďže idea uloženia modelu analyzovaného zdrojového kódu do databázy sa ukázala ako výhodná, pretože umožňuje zjednodušiť architektúru riešenia pomocou oddelenia analýzy zdrojového kódu od následného vyhľadávania v zdrojovom kóde a definuje jasné rozhranie medzi týmito dvoma modulmi systému.

Vyhladávanie založené na transformáciach

Steven P. Reis v jeho článku Semantic Based Code Search [68] popisuje svoj systém, ktorý využíva lokálne úložisko online dostupného open source kódu pre nájdenie špecifickej funkcie alebo triedy, ktoré spĺňajú požiadavky užívateľov. Umožňuje užívateľom určiť, čo hľadajú, čo najpresnejšie pomocou kľúčových slov, triedy alebo špecifikácie metódy, testovacími prípadmi, kontraktami alebo bezpečnostnými obmedzeniami. Jeho systém potom využíva množinu transformácií na mapovanie známeho kódu do toho, čo používateľ žiadal. Prvý krok použitého algoritmu je vyhľadanie pomocou kľúčových slov. Výstupom prvého kroku ju množina počiatočných riešení. Tieto riešenia sú potom upravované pomocou transformácii a porovnávané so zadanou špecifikáciou triedy alebo metódy. Nakoniec sú navrhované riešenia podrobené testovaniu pomocou zadaných testovacích prípadov. Prínosom opisovaného systému je vyhľadávanie nielen pomocou kľúčových slov, ale aj pomocou dynamickej špecifikácie t.j. pomocou špecifikácie testovacích prípadov.

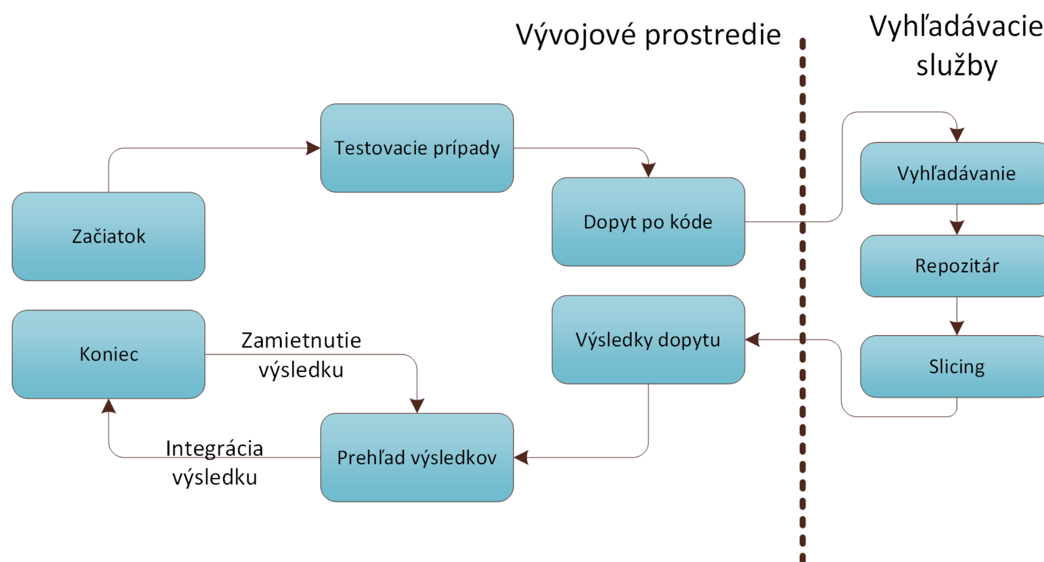


Obrázok 2.2: Princíp Semantic Based Code Search.

Vyhladávanie založené na testoch

Lemos vo svojom článku Applying TestDriven Code Search to the Reuse of Auxiliary Functionality [53] uvádza, že vývojári softvéru trávia značné množstvo času, nad tvorením pomocných funkcií používaných hlavnými modulmi systému (napr kompresia / dekompresia súborov, šifrovanie / prenos dát). S rastúcim množstvom open source kódu ktorý je k dispozícii na internete, je možné ušetriť množstvo času a úsilia pomocou znovupoužitia tohoto existujúceho kódu. Avšak, ak je tento typ opätovného použitia vykonávaný ručne, tak to môže byť únavné a viezť k chybám. Výsledný kód musí byť kontrolovaný a často krát prepi-

sovaný. Vo svojom príspevku sa zavádza používanie testovacích prípadov ako rozhrania pre automatizáciu vyhľadávania kódu. Tento prístup nazýva TestDriven Code Search (TDCS). Testovacie prípady slúžia na dva účely: (1) definujú správanie vyhľadávanej funkcionality, a (2) testujú zodpovedajúce výsledky z hľadiska vhodnosti použitia v konkrétnom kontexte.



Obrázok 2.3: Princíp Test Driven Code Search.

Systém bol implementovaný formou rozšírenia do vývojového prostredia Eclipse pre jazyk Java a funguje nasledovným spôsobom. Programátor napíše sadu testov. Vývojové prostredie spustí definované testy, a pre všetky testy ktoré sa nepodarilo skompilovať kvôli chýbajúcej funkcií v testovanom kóde. Pomocou analýzy syntaktického stromu sa z testov vyextrahuje informácia o špecifikácii chýbajúcej metódy (názov, typy parametrov, návratový typ) a pomocou tejto informácie a syntaktického stromu sa prehľadáva online repozitár s riešeniami.

Vyhladávanie za účelom štatistickej analýzy kódu

Posledné z nájdených vhodných riešení je Sourcerer [13] vyhľadávací nástroj pre open source softvér založený na vyhľadávaní pomocou štruktúry kódu. Článok opisujúci toto riešenie [13] je popisuje na aktuálne ciele výskumu a vyhľadávacie schopnosti nástroja Sourcerer. Sourcerer umožňuje vyhľadávanie, ktoré sú založené na štrukturálnych vlastnostiach a vzťahoch medzi prvkami kódu, takže primárne pracuje s abstraktným syntaktickým stromom. Sourcerer indexuje existujce open source softvéry do lokálnej databázy nad ktorou potom umožňuje vykonávať štatistické analýzy. Opisovaná verzia Sourcerera pracuje s indexom existujúcich Java open source projektov nad ktorým napríklad dokázala spočítať koreláciu medzi autormi kódu a riešenými problémami. Prvá verzia Sourcerera je verejne dostupná v jeho vývojovej verzii na stránkach projektu: http://sourcerer.ics.uci.edu/.

2.5 Zhrnutie

V tejto kapitole sa nachádza základný prehľad o problematike analýzy zdrojového kódu, ktorá je jednou z nosných tém tejto práce. Kapitola sa sústredí hlavne na prehľad prác rôznych autorov a uvedenie do problematiky. Kapitola popisuje vybrané druhy analýzy kódu, ku ktorým pridáva metriky kódu a vyhľadávanie v kóde. Všetky tieto analýzy spája jeden cieľ, a to zlepšovanie kvality a spoľahlivosti analyzovaného kódu.

Pri statickej analýze sú uvedené existujúce nástroje určené na analýzu kódu za účelom nájdenia potencionálnych chýb. Pri metrikách sa bol kladený dôraz na uvedenie metrík súvisiacich s návrhovými vzormi a ich aplikáciou. Problematika vyhľadávania v kóde poskytuje prehľad spôsobov uloženia modelu zdrojového kódu pre potreby neskorších dopytov, ktoré sú využité v navrhovanom systéme opísanom neskôr.

Kapitola 3

Návrhové vzory

Táto kapitola prináša prehľad súčasných prístupov k riešeniu problematiky návrhových vzorov, opisuje prístupy k automatickej detekcii návrhových vzorov, opisuje rôzne spôsoby zápisu a prináša vybranú sadu návrhových vzorov.

Návrhové vzory boli prvý krát použite a popísane Alexandrom, ale nie v kontexte softvérového inžinierstva, ale v architektúre a priestorovom plánovaní. [10, 9] V architektúre sa stretol s odmietnutím, avšak na poli softvérového inžinierstva boli jeho publikácie inšpiráciou pre tvorbu podobných zbierok znovu použiteľných vzorov.

V oblasti návrhu mal Alexander mnoho nasledovníkov. Najznámejším bol však Gamma, ktorý už v rámci svojej dizertačnej prace a neskôr v publikácii [44] definoval 23 návrhových vzorov, pri ktorých jasne definoval daný vzor a štandardný prípad jeho použitia. Návrhové vzory sú primárne určené pre objektovo orientovaný vývoj, konkrétne pre fázu podrobného návrhu. Model použitý na popis návrhových vzorov sa líši od publikácie k publikácii, avšak idea opakovaného použitia vzoru v podobnom kontexte zostava, vid nasledujúce definície.

Návrhové vzory v oblasti objektového návrhu softvéru a kvality zdrojového kódu sú definované ako osvedčené postupy riešenia opakujúcich sa všeobecných problémov návrhu, naopak nesprávne postupy sa nazývajú anti-vzory / chybové-vzory. Návrhové vzory sú zvyčajne definované ako vzťahy medzi jednotlivými objektami softvérového systému, prípadne sú definované priamo na úrovni tried a definujú ich konkrétnu štruktúru tried. Napríklad návrhový vzor dekorátor umožňuje dynamicky pridať funkcionalitu na objekt, jednoduchšie než vopred definovaná funkcionalita prostredníctvom dedičnosti, takže umožňuje pridať funkcionalitu k objektom v čase behu. Tento vzor taktiež znižuje väzbu medzi komponentmi, takže môžu byť modifikované bez ovplyvnenia sa navzájom.

Opakom návrhových vzorov sú anti-vzory, ktoré sú v podstate vzory uplatňované v nevhodnom kontexte. Existujú dva typy Anti-vzorov. Prvý z nich je použitie vzoru v nesprávnom kontexte, druhý je zlý vzor, ktorý možno použiť kdekoľvek. Ďalším príkladom nevhodného kódu sú takzvané pachy v kóde, ktoré ničia kód napríklad dlhými metódami alebo duplicitou kódu. Pachy v kóde sa najčastejšie vyskytujú na úrovni funkcií prípadne tried. Anti-Vzory sú zvyčajne spojené so štrukturálnymi problémami, ako je napríklad nevhodná hierarchia tried. Pachy kódu vznikajú najčastejšie ako chyby implementácie, zatiaľ čo anti-vzory bývajú spojené s nevhodným návrhom. V mnohých prípadoch nie je jednodu-

ché určiť, či sa jedná o pach kódu, alebo o vážnejší anti-vzor.

F. Buschmann vo svojej publikácii [23] opisuje návrhové vzory nasledovne. Vzor definuje ako dvojicu: opakujúci sa problém - riešenie v danom kontexte. Vzor však nie je čisto len popis problému, alebo popis štruktúry riešenia, vzor obsahuje oboje vrátane odôvodnenia, ktoré ich spája. Problém je posudzovaný s ohľadom na možne konflikty v návrhu a dôvody prečo daný problém je problém. Navrhované riešenie je uvádzané v podmienkach jeho očakávanej štruktúry a zahŕňa jasný popis prínosov a nevýhod daného riešenia.

T. Taibi vo svojej publikácii [78] opisuje návrhové vzory nasledovne. Návrhové vzory sú abstrakcie generované z cenných skúsenosti vývojárov ktoré získali pri riešení problémov s ktorými sa opakovane stretli v určitých kontextoch. Návrhové vzory sú podrobne testované a používane v mnohých vývojových a výskumných projektoch, ich opakovane použitie im poskytuje zlepšenie kvality softvéru pri súčasnom znížení času potrebného na vývoj.

Návrhové vzory bývajú často definovane neformálnym spôsobom technickou angličtinou, pripadne polo formálne pomocou UML diagramov. Neformálny zápis sa nedá algoritmicky spracovať a tým pádom slúži len pre vývojárov. Danému problému sa venuje Bayley vo svojej práci a publikáciách [16, 19, 83, 18, 17] nasledovne. Pôvodný účel návrhových vzorov uvedený v [10] je "zachytiť skúsenosť vývojára v podobe ktorú môžu ľudia efektívne používať". Preto sú návrhové vzory definovane vysvetlením všeobecných zásad v neformálnej angličtine a objasnené s formálnymi všeobecnými diagramy tried a konkrétnymi príkladmi kódu. Tato kombinácia je dostatočná pre vývojárov softvéru, ktorý dôkazu odhadnúť, ako aplikovať návrhové vzory pri riešení svojich vlastných problémov. Avšak, takýto zápis spôsobuje problémy pri snahe o automatické úpravy zdrojových kódov. Ak by návrhové vzory boli formalizované, mohli by softvérové nástroje refaktorovať zdrojové kódy v súlade s vybranými návrhovými vzormi.

3.1 Automatická detekcia návrhových vzorov

Problematike automatickej detekcie kompozitných návrhových vzorov sa venuje Heričko so svojim tímom. V publikácii A Composite Design-Pattern Identification Technique [47] prezentujú nimi navrhovanú techniku identifikácie návrhových vzorov v existujúcich programoch. Proces identifikácie sa skladá z použitia a kombinácie metrík vzorov. Zatiaľ čo ostatný výskum sa zameriava na identifikáciu návrhových vzorov pomocou metrík zdrojového kódu, Heričko definuje metriky vzoru. Pomocou popisovaného príkladu demonštrujú silu ich techniky pomocou identifikácie známeho kompozitného návrhového vzoru MVC (model view controller) [2]. Identifikácia kompozitných návrhových vzorov pomocou vzorových metrík môže viezť k identifikácii množiny možných vzorov. Z tohto dôvodu Heričko pridáva dodatočne vyhodnotenie pomocou metrík návrhu ktoré jasne určia použité návrhové vzory.

Espinoza vo svojom výskume [36] definuje kanonický model reprezentácie návrhových vzorov ktorý potom využívajú na detekciu návrhových vzorov v existujúcich zdrojových kódoch. Na definíciu takéhoto modelu použil vzťahy medzi triedami, ako sú napríklad dedičnosť, agregácia, inštanciácia. Na demonštráciu funkčnosti vytvoril systém DEPAIC++ (DEsign PAtterns Identification of C++ programs) ktorý vychádza z analýzy štruktúry existujúcich programov napísaných v C++ a overuje či analyzovaný kód je alebo nieje

založený na návrhových vzoroch.

3.2 Spôsoby zápisu návrhových vzorov

Návrhové vzory bývajú typicky definované neformálne a poloformálne, čo ich robí ľahko pochopiteľnými pre vývojárov. Gamma na popis vzorov v [10] používa 4 základné elementy: Názov vzoru, popis problému, popis riešenia, následky použitia vzoru. Na popis problému a riešenia používa kombináciu technickej angličtiny, UML diagramov a úryvkov zdrojových kódov. Následky použitia vzoru opisuje technickou angličtinou.

3.2.1 Neformálne spôsoby zápisu návrhových vzorov

Vzory sa katalogizujú do rôznych zbierok, či už v elektronickej podobe [30, 35] alebo v podobe tlačených publikácií [23, 8, 12]. Každá z týchto zbierok je členená rovnako ako pôvodné dielo Gammu [10]. Existuje však aj rada výskumov zaoberajúca sa formalizáciou zápisu návrhových vzorov. Dietrich vo svojej publikácii [32] definuje zápis návrhových vzorov pomocou ontológii a OWL. Kim vo svojej publikácii [42] používa upravený metamodel UML, pomocou ktorého dokáže modelovať presnejšie vzťahy medzi jednotlivými entitami návrhových vzorov a dokáže takto vytvorený UML diagram algoritmicky spracovať.

3.2.2 Formálne spôsoby zápisu návrhových vzorov

Cieľom formálnej špecifikácie návrhových vzorov je odstrániť nevýhody neformálnych opisov, zabezpečiť ich precízny a jednoznačný popis a umožniť ich automatizovanú verifikáciu. Formálne špecifikácie nenahrádzajú neformálne, iba ich dopĺňajú, pomáhajú zlepšiť pochopenie sémantiky, môžu pomôcť vývojárom pri rozhodovaní, ktorý vzor použiť. Viac informácií o formálnych špecifikáciách návrhových vzorov je v [19, 30, 35].

V súčasnosti sa možnosti formálnej špecifikácie návrhových vzorov delia do dvoch kategórií. Do prvej kategórie patria špeciálne vyvinuté formálne jazyky alebo polo-formálne grafické modelovacie jazyky. Tieto jazyky sú úplne nové a vyvinuté špeciálne pre potreby formálnych zápisov vzorov. V prípade grafických modelovacích jazykov sa väčšinou jedná o formálne doplnenie jazyka UML. Do druhej kategórie potom patria spôsoby, ktoré využívajú už existujúce formálne systémy, ktoré upravujú a vhodne kombinujú tak, aby ich bolo možné použiť pre formálnu špecifikáciu návrhových vzorov.

Hlavnou výhodou prvej kategórie je, že je jej jednoduchá ich použiteľnosť, pretože tieto formálne jazyky sú priamo koncipované na formálnu špecifikáciu vzorov. Nie je teda nutné ich akokoľvek upravovať. Nevýhodou je, že nie sú príliš rozšírené, a teda aj podpora nástrojov pre verifikáciu vzoru je veľmi nízka. Do tejto kategórie možno zaradiť napr. GEBNF (Graphical Extended Bacus Normal Form).

Výhodou druhej kategórie je, že väčšina vývojárov pozná existujúce formálne systémy. Tieto systémy už existujú dlho a vďaka tomu je aj podpora nástrojov pre automatizovanú verifikáciu na vysokej úrovni. Zásadnou nevýhodou ale je, že neumožňuje úplnú definíciu

vzoru, tj. definíciu správania a štruktúry. Dôvodom je, že tieto formálne systémy neboli primárne koncipované na opis vzorov, ale teraz sú na tento problém adaptované. Toto sa najčastejšie rieši kombináciou dvoch formálnych systémov: predikátovej logiky 1. rádu, pomocou ktorej sa definuje štruktúra vzoru, a temporálnej logiky akcie, ktorá definuje správanie.

SPINE

SPINE je formálny jazyk založený na programovacom jazyku Prolog, ktorý umožňuje formálnu špecifikáciu návrhového vzoru pomocou vstavaných funkcií a predikátov. Jazyk podporuje jednoduché existenčné kvantifikátory a umožňuje iterácie cez štruktúru tried (metódy, rozhranie a pod.) a implementáciu metód. Vďaka tomu umožňuje špecifikáciu štruktúry i chovania vzoru.

Termy v jazyku SPINE sú premenné, zoznamy alebo zložené termy (predikáty). Pravidlá definujúce zložené termy sú oddelené čiarkami a ich vyhodnocovanie prebieha zľava doprava. SPINE definuje špeciálne kvantifikátory forall a exists, ktoré pracujú nad zoznamom termov a sú ekvivalentné logickým operáciam AND respektíve OR. Nižšie je uvedený príklad špecifikácie vzoru Singleton v jazyku SPINE.

```
realises('PublicSingleton', [C]) :-
  exists(constructorsOf(C), true),
  forAll(constructorsOf(C), Cn.isPrivate(Cn)),
  exists(fieldsOf(C), F.and([
    isStatic(F), isPublic(F), isFinal(F), typeOf(F, C), nonNull(F)
  ])
)
```

Obrázok 3.1: Špecifikácia vzoru Singleton v jazyku SPINE.

Predikát *realises* špecifikuje definíciu vzoru, (v tomto prípade *PublicSingleton* bude porovnaný voči akejkoľvek triede C). Predikáty $constructorsOf$ a $fieldsOf$ vracajú zoznam konštruktorov danej triedy, respektíve jej atribúty. Predikáty *exists* a $forAll$ potom tieto zoznamy spracovávajú. Vyššie uvedená definícia teda znamená, že ľubovoľná trieda C je Singleton, ak:

1. obsahuje aspoň jeden konštruktor,
2. všetky konštruktory tejto triedy sú privátne,
3. všetky atribúty triedy sú statické, verejné, final, nie sú typu null a sú typu, ako je sama trieda

GEBNF

EBNF (Extended Backus Naur Form) je notácia, pomocou ktorej sa formálne popisuje syntax programovacích jazykov. Syntax je popísaná pomocou pravidiel v tvare: *<číslo so znamienkom> :: = [znamienko], <číslo>;* . Pravá strana pravidlá určuje prvok, pre ktorý sa definuje syntax, ľavá strana potom obsahuje samotnú syntax. Vyššie uvedené pravidlo teda

znamená, že číslo so znamienkom sa skladá z čísla a prípadného znamienka (vo väčšine programovacích jazykov sa pre kladné čísla znamienko neuvádza). Rozšírená Bacus-Naurova forma (EBNF) zavádza práve do definície pravidiel voliteľnosť výskytu prvkov, ktorá sa značí pomocou hranatých zátvoriek (na rozdiel od Bacus-Nauru formy, ktorá toto neumožňuje).

GEBNF je potom grafický variant EBNF. Je to teda notácia, ktorá umožňuje popísať syntax grafických modelovacích jazykov. V zmysle formálnej špecifikácie návrhových vzorov sa používa ako doplnok modelovacieho jazyka UML.

GEBNF definuje modelovací jazyk ako n-ticu (R, N, T, S), kde N je konečná množina neterminálnych symbolov, T je konečná množina terminálnych symbolov, $R \in N$, označovaný ako počiatočný symbol a S je konečná množina pravidiel zapísaných v tvare: $Y ::= Exp$, kde $Y \in N$ a Exp je v tvare: $L1 : X1L2 : X2...Ln : Xn$ alebo tiež $X1|X2|...|Xn, L1...Ln$ označujú názvy položiek a $X1...Xn$ sú položky, ktoré môžu byť v tvare Y, Y^*, Y^+, $[Y]$, $\underline{Y}$. Význam tejto notácie je uvedený v tabuľke 3.1.

Zápis	**Význam**	**Príklad použitia**
$X1\|...\|Xn$	Výber z X_1, X_2 ... X_n	ActorNode | UseCaseNode znamená, že entita je aktívnym účastníkom udalosti alebo prípadom použitia.
$L_1 : X_1$ $L_2 : X_2$ $L_n : X_n$	Usporiadaná postupnosť položiek typu X_1, X_2 ... X_n, ktoré sú sprístupnené pod menami L_1, L_2 ... L_n	ClassName : Text, Attributes : Attribute*, Methods : Method* znamená, že entita sa skladá z troch častí nazvaných ClassName, Attributes a Methods
X^*	$X^i, i \geq 0$	Diagrams* znamená, že entita sa skladá z n diagramov kde $n \geq 0$
X^+	$X^i, i \geq 1$	Diagrams* znamená, že entita sa skladá z n diagramov kde $n \geq 1$
$[X]$	X je voliteľné	Prvok X je voliteľný
$\underline{X}$	Odkaz na existujúci prvok typu X	<u>ClassNode</u>: je odkaz na existujúci prvok typu trieda

Tabuľka 3.1: Význam notácie GEBNF.

Formálna špecifikácia pomocou Prologu

Ďalšou možnosťou, ako formálne špecifikovať návrhové vzory, je programovací jazyk Prolog. Návrhové vzory sú reprezentované pomocou pravidiel Prológu a sú uložené v jeho databáze. Výhodou tohto prístupu je, že návrhové vzory môžu byť znovu použité jednoduchou inštanciáciou patričných pravidiel. Typické vlastnosti a obmedzenia vzoru sú tiež zapísané pomocou pravidiel a vďaka tomu je ich možné jednoducho verifikovať. Pridávanie alebo odoberanie prvkov štruktúry vzoru je realizované pomocou príkaov *assert* a *retract*, a prevod takého zápisu vzoru do kódu programovacieho jazyka je veľmi jednoduchý, a podporovaný mnohými nástrojmi (napr. DRACO-PUC).

3.3 Vybraná sada paralelných návrhových vzorov

Nasledujúca kapitola prináša neformálny prehľad vybraných paralelných návrhových vzorov zo zbierky Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects [71], ktorá je aktívne citovaným zdrojom mnohých autorov [31, 75, 70, 37, 43], pričom všetci z uvedenej množiny uvádzajú danú zbierku ako štandard. Vybrané paralelné návrhové vzory sa vždy venujú dvom základným témam a to architektúre ktorá má umožniť súbežný beh a synchronizácii.

Návrhové vzory riešiace problémy paralelnosti definujú spôsoby návrhu paralelných systémov. To znamená spôsoby ako rozdeliť objekty v programoch. Napríklad návrhový vzor Half-Sync/Half-Async rozdeľuje aplikáciu do dvoch vrstiev synchrónnej a asynchrónnej, a medzi tieto vrstvy vkladá nezávislú komunikačnú medzivrstvu.

Návrhové vzory riešiace problémy synchronizácie definujú spôsoby prístupu k zdieľaným komponentom systému. Ukazujú spôsoby ako riešiť situácie pri ktorých potrebujeme synchronizáciu viacerých vlákien na prístup k zdieľaným komponentom.

Nasledujúce kapitoly obsahujú prehľad návrhových vzorov, ich vzájomné väzby, využiteľnosť a podrobný popis jednotlivých návrhových vzorov.

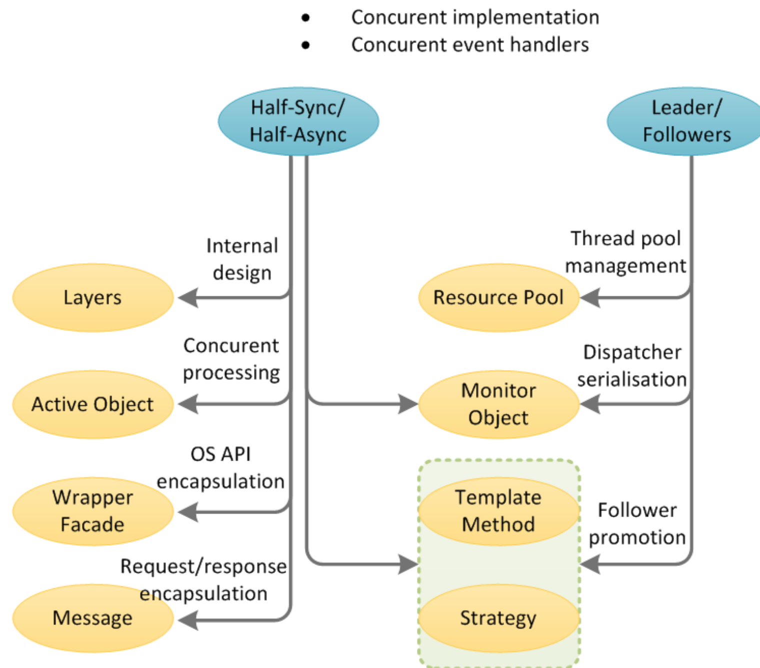
3.3.1 Návrhové vzory pre súbežný beh

Výber základnej paralelnej architektúry má veľký vplyv na návrh a výkon paralelného systému. Žiadna z nižšie spomenutých paralelných architektúr nie je výlučne určená na určitú množinu úloh, avšak vhodnou kombináciou prístupov vieme vyriešiť radu paralelných problémov. Nasledujúce diagramy zobrazujú vzťahy medzi návrhovými vzormi.

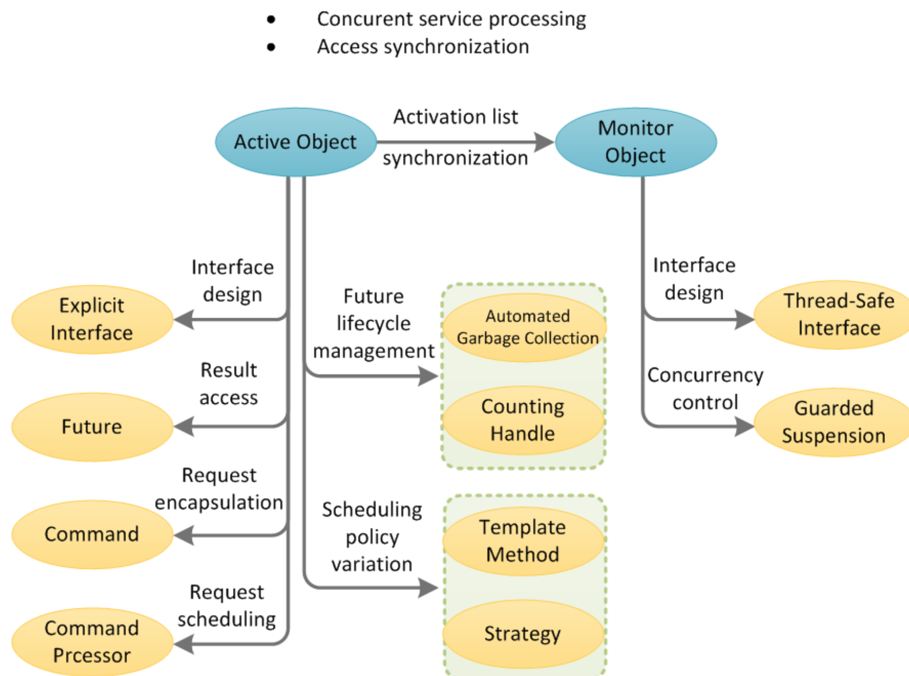
Návrhový vzor Half-Sync / Half-Async rozdeľuje jednotlivé funkcie komponentov do dvoch skupín a to synchrónne a asynchrónne. Medzi tieto dve skupiny funkcií pridáva tretiu medzivrstvu slúžiacu na komunikáciu pomocou front. Ako vidno z obrázku na svoju implementáciu môže používať radu ďalších návrhových vzorov ako napríklad Layers na rozdelenie do vrstiev, Message na posielanie správ medzi vrstvami, alebo Monitor Object na prístup k zdieľanej medzivrstve.

Návrhový vzor Leader/Followers definuje množinu dopredu alokovaných vlákien, ktoré sa striedajú pri spracovávaní prichádzajúcich udalostí. Jedno z vlákien je líder čakajúci na príchod udalosti, ostatné vlákna buď spracovávajú svoje udalosti, alebo čakajú na zmenu svojej úlohy z nasledovníka na vedúceho lídra. Na svoju implementáciu môže využívať napríklad návrhový vzor Resource Pool, alebo Monitor Object na prístup k zdieľanému zdroju udalostí.

Návrhové vzory Active Object a Monitor Object definujú možnosti prístupu k zdieľaným objektom. Active Object používa samostatné vlákno aktívneho objektu, ktoré spracováva požiadavky na daný objekt. Na svoju implementáciu môže využívať viacero návrhových vzorov, ako napríklad Command na predávanie príkazov, Future na predávanie návratových hodnôt, alebo Command procesor na vnútornú reprezentáciu aktívneho objektu.



Obrázok 3.2: Vzťahy návrhových vzorov Half-Sync/Half-Async a Leader/Followers.



Obrázok 3.3: Vzťahy návrhových vzorov Active Object a Monitor Object.

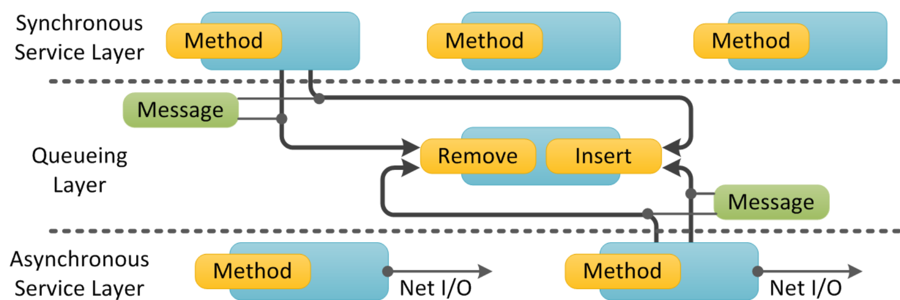
Návrhový vzor Monitor Object neobsahuje žiadne vlákno, avšak synchronizáciu viacerých klientskych vlákien rieši pomocou ich blokovania. Jedná sa o jednoduchší systém ako

Active Object ktorého hlavnou výhodou je jednoduchšia implementácia, medzi nevýhody patrí horšia rozšíriteľnosť vyplývajúca z jednoduchej implementácie.

Half-Sync/Half-Async

Paralelný softvér často vykonáva synchrónne aj asynchrónne volania funkcií. Asynchrónne sa používajú na efektívne spracovanie nízkoúrovňových systémových volaní, synchrónne volania zjednodušujú implementáciu aplikačných služieb. Aby sme mohli maximalizovať úžitok z oboch prístupov musíme efektívne koordinovať jednotlivé volania a ich vzájomné väzby.

Odporúčanie vzoru: *počas návrhu rozdeľte funkcie paralelného softvéru do dvoch samostatných vrstiev: synchrónnej a asynchrónnej a pridajte frontovú medzivrstvu slúžiacu na komunikáciu medzi nimi.*



Obrázok 3.4: Návrhový vzor Half-Sync/Half-Async.

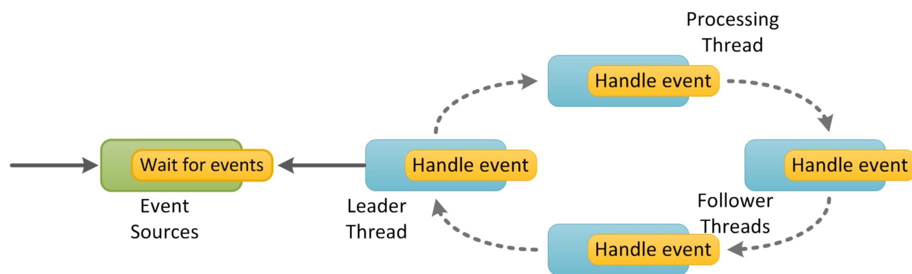
Návrhový vzor Half-Sync/Half-Async presadzuje striktné rozdelenie funkcií medzi tri vrstvy, ktoré robia paralelný softvér jednoduchšie pochopiteľný a upravovateľný. A čo viac, asynchrónne a synchrónne volania funkcií sa neovplyvňujú vzájomnými nevýhodami: výkon asynchrónne volaných funkcií nie je degradovaný blokujúcimi synchrónnymi volaniami a jednoduchosť programovania synchrónnych funkcií nie je ovplyvnená komplexnosťou asynchrónne volaných funkcií. Použitie frontovej medzivrstvy zabraňuje pevne naprogramovaným závislostiam medzi synchrónnou a asynchrónnou vrstvou a tým zlepšuje udržiaveteľnosť programu.

Leader/Followers

Väčšina udalosťami riadeného softvéru používa milti-threading na paralelné spracovanie udalostí. Ktoré zvyšuje výkon, avšak pri nesprávnom návrhu, môže spôsobiť viac problémov ako úžitku.

Odporúčanie vzoru: *použite predalokovanú množinu vlákien na koordináciu detekcie, demultiplexáciu, vybavenie a spracovanie udalostí. V tejto množine vždy len jedno vlákno (líder) čaká na udalosť. Po príchode udalosti líder povýši svojho nasledovníka na lídra a pôvodný líder začne obsluhovať príslušnú udalosť vo svojom vlákne.*

Predalokovaním pola vlákien, návrhový vzor Leader/Followers zabraňuje strate času slúžiaceho na vytváranie a rušenie samostatných vlákien pre jednotlivé udalosti. Predalokované vlákna pre jednotlivé udalosti taktiež odstraňujú problémy so synchronizáciou vlákien, predávaním hodnôt a riadenia medzi vláknami, správou pamäte. Určenie jedného z vlákien



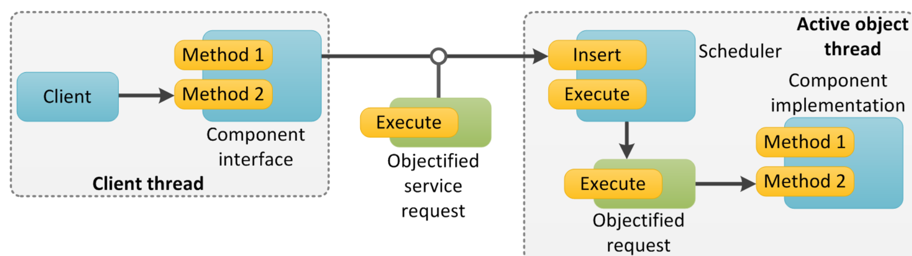
Obrázok 3.5: Návrhový vzor Leader/Followers.

ako lídra odstraňuje potrebu centrálnej správy pridelovania vlákien ktorá býva často úzkym hrdlom.

Active Object

Paralelnosť môže zlepšiť kvalitu služby, napríklad súbežným spracovaním viacerých požiadaviek klienta bez blokovania. Návrhári systémov sa však musia vhodne rozhodnúť ako vyjadriť jednotky paralelnosti a ako s nimi pracovať počas behu programu.

Odporúčanie vzoru: *definujte jednotky paralelnosti ako volania služieb na komponentoch, a tieto volania spracovávajte v separátnom vlákne inom ako volajúcom danú službu. Navrhnite systém tak aby klient aj príslušný komponent mohli asynchrónne pracovať a navzájom komunikovať.*



Obrázok 3.6: Návrhový vzor Active Object.

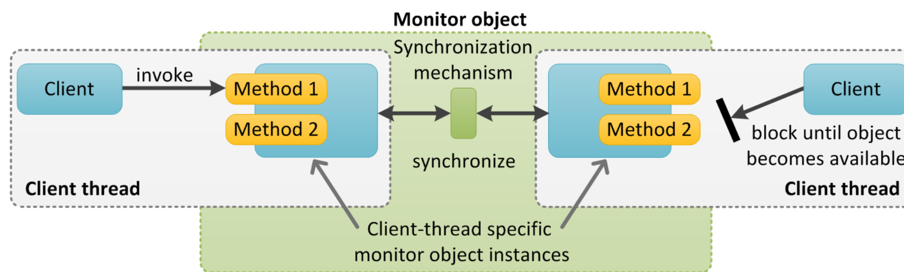
Návrhový vzor Active Object a zlepšuje paralelnosť v programe vykonávaním klientskych vlákien nezávisle od volaní potrebných služieb. Zložitosť synchronizácie je redukovaná použitím interného plánovača pre každý komponent (aktívny objekt).

Monitor Object

Paralelný softvér často obsahuje objekty ktorých metódy sú volané rôznymi klientskymi vláknami. Aby sme ochránili vnútorný stav týchto objektov, je nutné synchronizovať a plánovať prístup vlákien k jednotlivým funkciám. Klienti taktiež nepotrebujú vidieť rozdiel medzi prístupom k zdieľaným a nezdieľaným objektom.

Odporúčanie vzoru: *povoľte vykonanie funkcií klientskym vláknam, avšak pomocou systému zámkov skoordinujte vykonávanie volaných funkcií. Pristupujte k zdieľanému objektu len pomocou synchronizovaných metód, ktoré povoľujú vykonanie iba jednej inštancie bez*

paralelnosti.



Obrázok 3.7: Návrhový vzor Monitor Object.

Každý objekt obsahuje monitorovací zámok, ktorý slúži na serializáciu prístupu k zdieľanému stavu daného objektu. Vnútri synchronizovaných metód najskôr klientske vlákno najskôr získa exkluzívny prístup k objektu, a potom sa vykoná príslušná funkcia upravujúca stav objektu. Ak klientské vlákno automaticky nezíska exkluzívny prístup k objektu, tak je dané vlákno uspané a čakajúce na svoje prebudenie.

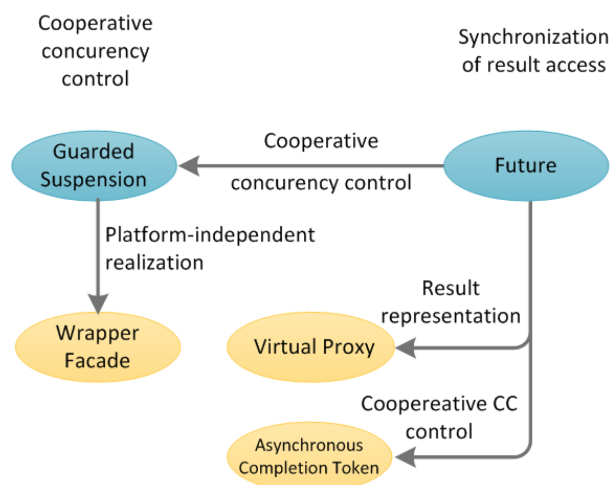
3.3.2 Synchronizačné návrhové vzory

Delenie programu na paralelne bežiace celky je iba polkou problémov spojených s paralelnosťou. V prípade zdieľaných objektov medzi viacerými vláknami vzniká problém synchronizácie prístupu ku zdieľaným dátam. Táto kapitola obsahuje prehľad základných návrhových vzorov, slúžiacich na zlepšenie synchronizácie alebo na odstránenie zbytočnej synchronizácie v prípadoch kedy nie je nutná.

Nutnosť vzájomnej synchronizácie robí paralelné programovanie zložitejším. Paralelne bežiace vlákna musia zdieľať jeden prístup k zdieľaným objektom. Bez potrebnej synchronizácie by paralelný prístup k zdieľaným objektom mohol spôsobovať ich zničenie, v lepšom prípade nekonzistentnosť ich stavu. Na odstránenie možnej nekonzistentnosti stavov zdieľaných objektov slúži kritická sekcia. Kritická sekcia je blok programu, ktorý môže byť vykonávaný len jedným vláknom, a počas vykonávania jedným vláknom, nesmie byť prerušený alebo ukončený druhým vláknom.

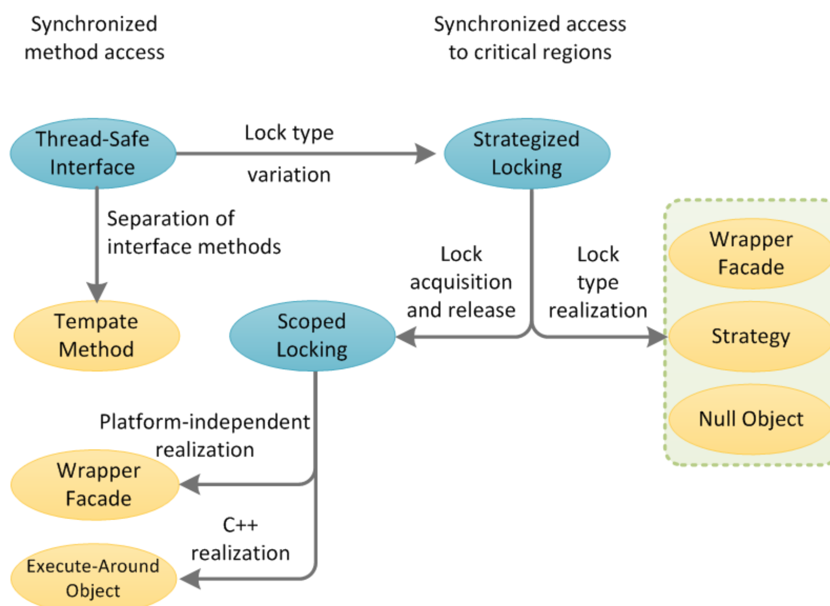
Nasledujúce diagramy zobrazujú vzťahy medzi návrhovými vzormi. Návrhový vzor Guarded Suspension slúži na kooperatívnu kontrolu paralelnosti a na svoju implementáciu môže využívať návrhový vzor Wrapper Facade. Pod kooperatívnou kontrolou paralelnosti rozumieme ovplyvňovanie behu jedného vlákna druhým vláknom.

Návrhový vzor Future slúži na synchronizáciu prístupu k výsledkom paralelného výpočtu a na svoju implementáciu môže využívať návrhové vzory Virtual Proxy a Asynchronous Completion Token.



Obrázok 3.8: Vzťahy návrhových vzorov Guarded Suspension a Future.

Návrhový vzor Thread Safe Interface slúži na synchronizáciu prístupu k privátnym metódam zdieľaného komponentu. Na svoju implementáciu môže využívať návrhový vzor Template Method.



Obrázok 3.9: Vzťahy návrhových vzorov Thread-Safe Interface, Strategized Locking a Scoped Locking.

Návrhový vzor Strategized Locking predstavuje zjednotenie spôsobu zamykania kritickej sekcie pomocou jednotného rozhrania pre všetky druhy zámkov. Tento návrhový vzor slúži rovnako ako Thread Safe Interface na synchronizáciu prístupu ku kritickej sekcií programu.

Návrhový vzor Scoped Locking slúži už na zamknutie konkrétnej kritickej sekcie. Scped

Locking odporúča používať dopredu pripravené kľúčové slová jazyka, ako napríklad synchronized v Jave.

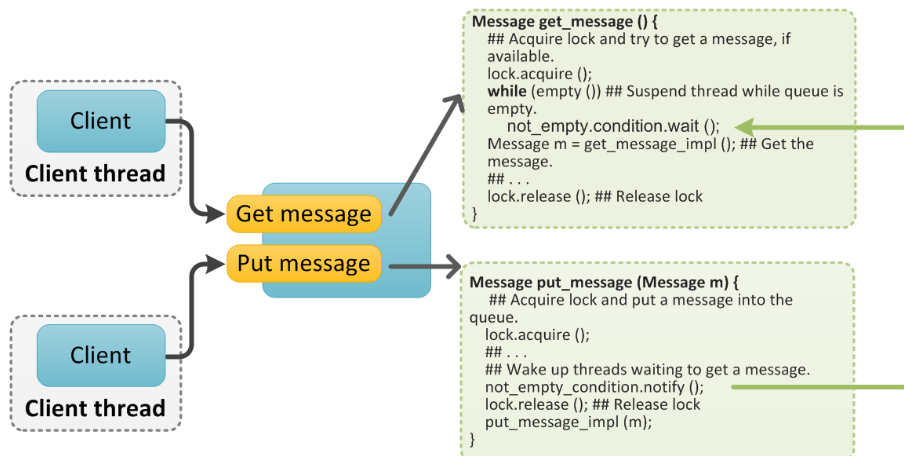
Návrhové vzory Thread Safe Storage, Copied Value a Immutable Value sa vyhýbajú synchronizácii. Tread Safe Storage poskytuje samostatnú kópiu spracovávaneho objektu pre každé vlákno. Copied Value používa pri každom volaní iného vlákna skopírovaný objekt a tým pádom úplne odstraňuje potrebu synchronizácie. Immutable Value zakazuje zápis hodnôt objektu a poskytuje iba read only hodnoty. Keďže vlákna nemôžu do objektu zapisovať, takisto odpadá nutnosť synchronizácie ako pri Copied Value.

Nasleduje detailný popis jednotlivých návrhových vzorov.

Guarded Suspension

V paralelných programoch často môžeme vykonať funkciu na komponente, iba v prípade splnenia určitých podmienok nazývaných guards (ochranné podmienky). Avšak stav ochranných podmienok sa môže meniť pomocou behu iných paralelných funkcií na danom komponente. Nie vždy je možné ukončiť zablokovanú metódu aby mohla byť vykonaná funkcia meniaca ochranné podmienky zablokovanej funkcie.

Odporúčanie vzoru: *namiesto ukončovania zablokovanej funkcie uspite jej vlákno, tak že ostatné vlákna získajú prístup k zdieľanému komponentu a môžu tak zmeniť hodnotou ochrannej podmienky a tým uvoľniť vlákno zablokovanej funkcie.*



Obrázok 3.10: Návrhový vzor Guarded Suspension.

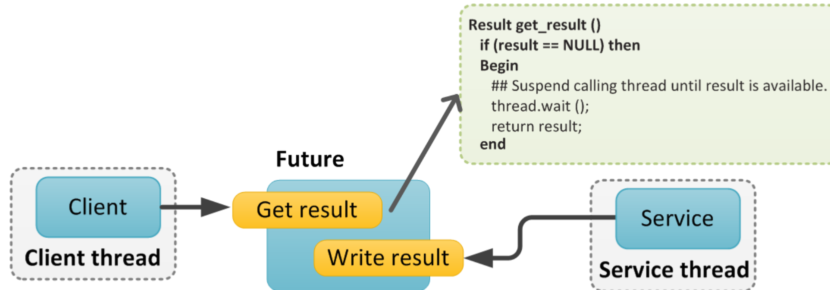
Hlavným prínosom návrhového vzoru Guarded Suspension je, že minimalizuje náklady spojené s paralelizáciou a taktiež zvyšuje dostupnosť zdieľaných komponentov. Ak je uspanie vlákien riešené na vrstve OS, tak cena daného uspania a príslušnej synchronizácie je minimálna.

Future

Funkcie ktoré sú volané paralelne na komponentoch môžu potrebovať vrátiť vypočítanú hodnotu klientskemu vláknu. Aj keď klient nemusí byť blokovaný po volaní zvolenej funkcie a môže pokračovať vo svojich výpočtoch, výsledok volanej funkcie nemusí byť dopočítaný

v čase keď ho klientské vlákno už potrebuje.

Odporúčanie vzoru: *ihneď po volaní funkcie vráťte „virtuálny“ dáta objekt, nazvaný Future. Future objekt obsahuje informáciu o stave výpočtu a platnosti obsiahnutého výsledku a vráti hodnotu iba v prípade platnosti výsledku.*



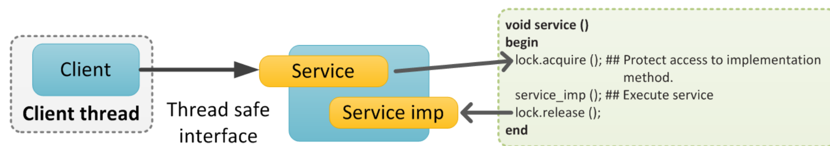
Obrázok 3.11: Návrhový vzor Future.

Ak klientske vlákno chce prečítať hodnotu z future objektu pred tým než je výsledok platný, tak future objekt uspí klientske vlákno do doby než je platný výsledok zapísaný vo future objekte. Future objekt taktiež môže obsahovať neblokujúcu funkciu slúžiacu na neblokujúce skontrolovanie platnosti uloženej hodnoty.

Thread-Safe Interface

Komponenty v paralelných programoch musia byť bezpečné (thread-safe). Často sa stáva že ich funkcie vyžadujú zámky zabezpečujúce synchronizáciu a serializáciu volaní daných funkcií. Avšak ak funkcia volá inú funkciu toho istého komponentu, môže nastať uviaznutie. (self-deadlock) Uviaznutie môže vzniknúť ak funkcia volá inú funkciu, alebo rekurzívne sama seba.

Odporúčanie vzoru: *rozdeľte funkcie komponentu na verejne prístupné rozhranie zabezpečujúce zámky a synchronizáciu a internú implementáciu zabezpečujúcu vykonanie jednotlivých funkcií.*



Obrázok 3.12: Návrhový vzor Thread-Safe Interface.

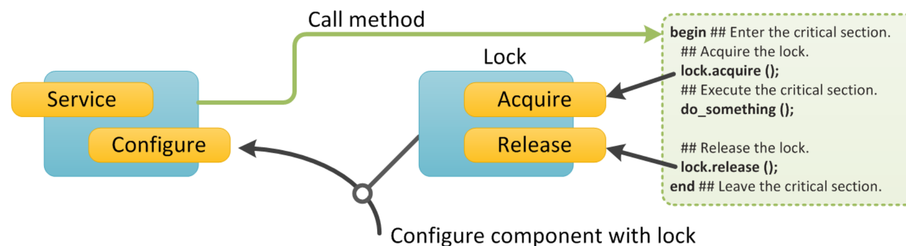
Na zabezpečenie správnej synchronizácie klientske vlákna volajú iba funkcie rozhrania. Funkcia rozhrania získa potrebný zámok a zavolá príslušnú implementačnú funkciu ktorá sa už nestará o zamykanie a môže slobodne volať iné implementačné funkcie. Uviaznutie (self-deadlock) nemôže nastať pretože zámok sa získava iba raz na začiatku v rozhraní a pri rekurzívnom volaní funkcií je zámok získavaný iba raz, čo taktiež zvyšuje výkon programu.

Strategized Locking

Komponenty ktoré sú zdieľané medzi vláknami v paralelnom prostredí musia chrániť

svoje kritické sekcie pred paralelným prístupom. Na druhú stranu, rôzne konfigurácie komponentov a ich rôzne stavy môžu vyžadovať rôzne spôsoby zamykania ako napríklad semafory, binárne zámky, reader-writer zámky.

Odporúčanie vzoru: *definujte zámky vo forme zásuvných modulov ktorých každý typ definuje jeden spôsob zamykania. Vybavte všetky tieto zásuvné moduly rovnakým rozhraním, takže zamykanie nebude závisieť od príslušnej implementácie, ale len od zvoleného typu zámku.*



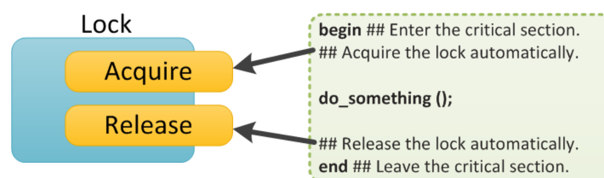
Obrázok 3.13: Návrhový vzor Strategized Locking.

Návrhový vzor Strategized Locking ponúka viaceré výhody. Namiesto separátnej implementácie pre každý jeden zámok, existuje len jedna spoločná implementácie pre všetky zámky. Týmto spôsobom sa znižuje redundancia kódu a zlepšuje sa taktiež udržiavateľnosť daného programu. Výber použitého spôsobu zamykania sa dá hocikedy jednoducho zmeniť, prípadne upraviť.

Scoped Locking

Kritická sekcia kódu ktorá by mala bežať sekvenčne býva často chránená zámkami, ktoré sú získavané vždy keď vlákno vstúpi ku kritickej sekcii, a sú uvoľňované vždy keď z nej vystúpi. Ak však programátor explicitne pomocou vlastného kódu nadefinuje zamykanie, tak musí explicitne definovať aj odomykanie. Problém je v tom že pri komplexnejších zdrojových kódoch rastie riziko zabudnutia uvoľnenia zámku.

Odporúčanie vzoru: *orámujte kritickú sekciu a automaticky získajte zámok pri vstupe do nej, a automaticky ho uvoľnite pri výstupe ľubovoľnou cestou z "rámu".*



Obrázok 3.14: Návrhový vzor Scoped Locking.

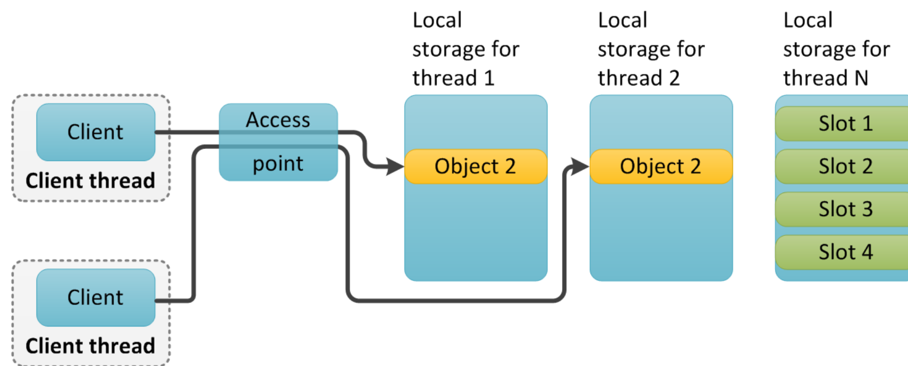
Návrhový vzor Scoped Locking zvyšuje robustnosť paralelného softvéru tým, že eliminuje časté programátorské chyby spojené so synchronizáciou viacerých vlákien. Zámky sú automaticky získavané keď vlákno vstúpi do kritickej sekcie, a automaticky uvoľňované keď z nej vystúpi. Implementácia tohto návrhového vzoru závisí od použitého programovacieho jazyka. Napríklad programovací jazyk Java obsahuje kľúčové slovo synchronized ktoré pri-

kazuje kompilátoru automaticky vygenerovať príslušné inštrukcie obsluhujúce zamykanie a odomykanie zámkov.

Thread-Specific Storage

Prístup k objektu ktorý je spojený s prostredím (envoroment variables) robí tento objekt prirodzene globálny. Avšak čo ak je nutné aby tento objekt bol špecifický pre každé vlákno? Daný objekt potom nemôže byť fyzicky globálny s jedinou inštanciou držiacou stav objektu. Zamykanie každého prístupu k objektu, alebo zamykanie príslušnej hodnoty pri každom prístupe vlákna môže znížiť výkon systému.

Odporúčanie vzoru: *použite spoločný prístupový bod k objektom spojeným s prostredím, avšak jednotlivé inštancie držte v separátnom bloku pamäte pre každé vlákno.*

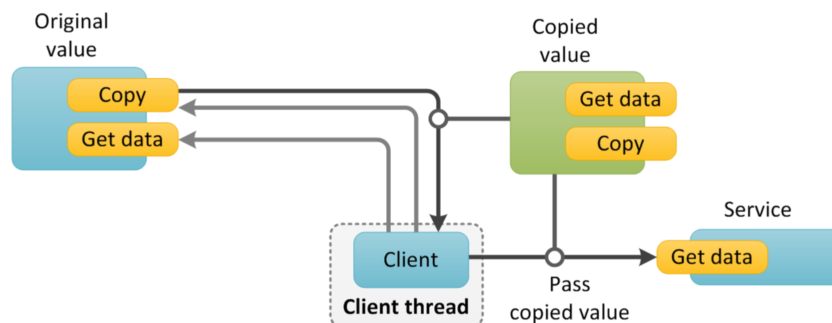


Obrázok 3.15: Návrhový vzor Thread-Specific Storage.

Návrhový vzor Thread-Specific Storage nepotrebuje žiadne zámky na prístup k dátam každého vlákna. (thread specific data). Implementácia globálneho prístupového bodu je možná použitím klasického návrhového vzoru proxy.

Copied Value

Hodnoty objektov bývajú uložené v parametroch. Ak je nejaký objekt zdieľaný medzi viacerými vláknami, tak môže nastať problém pri zmene dát v jednom vlákne. Zmena parametrov objektu v jednom vlákne, môže mať nežiaduce následky ak je ten istý objekt používaný v inom vlákne.



Obrázok 3.16: Návrhový vzor Copied Value.

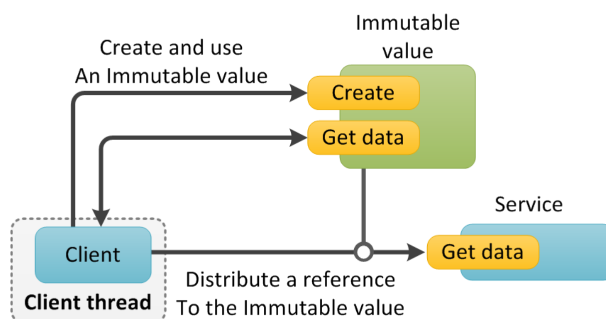
Odporúčanie vzoru [3.16]: *definujte objekty, ktorých inštancie sú kopírovateľné. Keď takýto objekt používate v komunikácii s iným vláknom, uistite sa že druhému vláknu predávate kópiu objektu.*

Každé vlákno používa svoju kópiu objektu, takže takéto objekty nie sú zdieľané medzi vláknami. Absencia zdieľania znamená že neexistuje žiaden dôvod na synchronizáciu a tým pádom jednotlivé vlákna sa neblokujú pri prístupe k danému objektu. Napríklad v jazyku c# existuje kľúčové slovo `struct`, pomocou ktorého sa dajú takéto jednoducho kopírovateľné objekty vytvoriť.

Immutable Value

Ak je nejaký objekt zdieľaný medzi viacerými vláknami, tak môže nastať problém pri zmene dát v jednom vlákne. Zmena parametrov objektu v jednom vlákne, môže mať nežiaduce následky ak je ten istý objekt používaný v inom vlákne. Ak však vieme že daný objekt bude použitý iba na čítanie a druhé vlákno nebude do neho zapisovať, tak je zbytočné vytvárať kópiu objektu iba na predanie daného objektu ako parametra do druhého vlákna. Konštrukcia nového objektu zaberie určitú dobu, odstránením zbytočných konštrukcií a kopírovaní vieme získať procesorový čas.

Odporúčanie vzoru [3.17]: *pri návrhu definujte objekty, ktorých inštancie sú nemenné. Vnútorný stav objektu je nastavený v konštruktore a žiadne ďalšie zmeny nie sú povolené.*



Obrázok 3.17: Návrhový vzor Immutable Value.

V nemennom objekte sa nachádzajú iba read only parametre. Absencia ľubovoľnej možnosti zmeniť daný objekt odstraňuje nutnosť akejkoľvek synchronizácie a tým pádom zjednodušuje návrh systému a zefektívňuje prácu programu. Odstránením nutnosti kopírovania objektov, taktiež zlepšuje výkon programu.

3.4 Zhrnutie

Táto kapitola priniesla prehľad súčasných prístupov k riešeniu problematiky návrhových vzorov. Na začiatku je čitateľ uvedený do problematiky návrhových vzorov ako takých. Ďalej sú opisané prístupy k automatickej detekcii návrhových vzorov, ktoré boli študované počas vypracovávania tejto práce. Kapitola vo svojom strede opisuje rôzne spôsoby zápisu návrhových vzorov, ktoré boli študované za účelom znovu použitia existujúceho riešenia.

Na svojom konci kapitola prináša vybranú sadu návrhových vzorov zo zbierky Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects [71], ktorá je aktívne citovaným zdrojom mnohých autorov a preto vzory v nej spomenuté slúžia ako základná množina pre ďalšie pokusy a výskum.

Kapitola 4

Refaktoring

Nasledujúca kapitola prináša zjednodušený pohľad do problematiky refaktoringu, ktorá je v tejto práci preberaná okrajovo ako pomocný problém. Účelom tejto kapitoly je ponúknuť úvod do problematiky a základný prehľad existujúcich prístupov a výsledkov výskumov.

Prvé použitie termínu refaktoring v literatúre bolo v práci Opdyka a Johnsona [63, 64], hoci v praxi bola táto technika používaná dávno pred tým. Opdyke definuje refaktoring ako "reštrukturalizáciu programu zachovávajúcu správanie". Fowler používa podobnú definíciu, ale zdôrazňuje, že od procesu refaktoringu očakávame zlepšenie dizajnu: Refaktoring je proces zmeny softvérového systému takým spôsobom, že sa nemení vonkajšie správanie kódu, ale zlepšuje jeho vnútorná štruktúra [41, 40]. Roberts mení definíciu radikálne tým, že umožňuje refaktoring, ktorý mení správanie programu .

Počas svojho výskumu Opdyke definoval sadu transformácií programov, použiteľných na programy v jazyku C++. V ďalšej svojej práci ukázal, ako by mohli byť dané transformácie použité na definíciu transformácií vyššej úrovne, napríklad na prevod dedičnosti na agregáciu, a naopak [50]. Roberts vo svojom výskume rozšíril prácu Opdyka tým, že poskytol formálny model pre skladanie transformácií, a skúmal využitie dynamických informácií v refaktoringu. Nasledujúce kapitoly sa venujú prístupom k problematike automatického refaktoringu. V mnohých prípadoch sa termín refaktoring nepoužíva, ale podstata daných prác spočíva v transformácii kódu so zachovaním vonkajšieho správania.

Jedným z hlavných prínosov objektovo orientovaného návrhu je dedičnosť. Projektovanie tried a ich hierarchie je však stále náročné, aj keď bolo veľa pokusov o vytvorenie automatickej podpory tejto činnosti. Pravdepodobne najstaršia práca, ktorá sa touto otázkou zaoberala bola od Puna a Windera [67]. Keď dizajnér pridá triedu do hierarchie, môže sa stať, že hierarchia tried spôsobí dedenie nežiaducich vlastností. To znamená, že hierarchia by mala byť nanovo usporiadaná, aby oddelila nežiaduce atribúty od tých, ktoré majú byť zdedené. Pun a Winder ukazujú, ako môže byť tento proces reorganizácie automatizovaný a čiastočne formalizovaný.

Casais rieši problém nežiaducej dedičnosti trochu odlišným spôsobom, s uvedením celkového i čiastkových algoritmov, ktoré reorganizujú hierarchiu tried tak, aby sa odstránila nežiaduca dedičnosť atribútov [25]. To zlepšuje prácu Puna a Windera tým, že umožňuje postupnú reorganizáciu hierarchie tried vždy po pridaní novej triedy. Casais tiež presne defi-

nuje, ako automatizovať proces reštrukturalizácie algoritmov a v [26] predstavuje výsledky uplatňovanie jeho prístupu. Jeho „refaktoring“ je určený na prevádzku v automatickom režime, ktorý má výhodu, že môže byť použitý v projektoch s veľkou hierarchiou tried. Nevýhoda jeho prístupu je, že v niektorých prípadoch môže jeho prístup viesť k výsledku, ktorý je buď nepochopiteľný, alebo neprináša žiaden pozitívny efekt z pohľadu softvérového inžinierstva.

Lieberherr, Bergstein a Silva-Lepe popisujú algoritmus, ktorý zoberie sadu tried a minimalizuje počet agregácií a dedičnosť v tejto sade pri zachovaní objektov definovaných na začiatku [20, 54]. Táto práca je založená na filozofii, že abstrakcie sú objavované a nie vymýšľané [49], to potom umožňuje návrhárovi definovať konkrétne objekty, ktoré chce použiť, a potom vyriešiť hierarchiu týchto objektov ako celku. Novšie práce Hurscha a Seitera v rovnakej oblasti opisujú súbor transformácii zachovávajúcich správanie, ktoré možno použiť nad sadou tried [79]. Táto práca nikdy nedosiahla veľkej popularity, pravdepodobne kvôli tomu, že je pevne viazaná k zriedkakedy používanému adaptívnemu modelu softvéru, kde štruktúra tried je vytvorená oddelene od správania. Toto silne kontrastuje s prácou Opdyka a Robertsa, ktorá predpokladá, že knižnice tried budú implementované pomocou bežných programovacích jazykov a modelov vývoja softvéru.

Ivan Moore vyvinul nástroj nazvaný Guru, ktorý môže analyzovať a reštrukturalizovať hierarchiu dedičnosti [59]. Hierarchiu dedičnosti optimalizuje tak, aby sa zachovávalo správanie programu. Optimálne podľa Moora znamená, že duplicitné metódy sú odstránené, zdieľanie metód je maximálne a redefinícia metód minimálna. Moore zistil, že vo všeobecnosti je v niektorých prípadoch potrebný ručný zásah, aby bolo možné vytvoriť dobrý výsledok a že úplne nesprávne vytvorená hierarchia na vstupe sa pomocou reštrukturalizácie nezlepší. Taktiež je tu riziko tohto druhu automatických úprav, že kľúčové abstrakcie a vzťahy, ktoré vývojár definoval budú v procese reštrukturalizácie hierarchie tried odstránené, pokiaľ ešte neboli v skutočnosti využité. V [60] Moore rozširuje jeho algoritmus o refaktoring metód pomocou vkladania bežných výrazov do oddelených metód. Kým táto úroveň refaktoringu môže znížiť množstvo kódu v programe a zvýšiť znovupoužitie kódu, nemusia nové metódy vyzerať koherentne pre programátora.

Tip a Snelting navrhujú reštrukturalizáciu hierarchie tried s použitím analýzy konceptu [72]. Vytvorením hierarchie tried vývojár popisuje jeho vnímanie kľúčových tried a ich vzťahov v oblasti, ktorú modeluje. Programátor, ktorý používa túto hierarchiu môže nájsť nezrovnalosti v návrhu a to spôsobí anomálie jeho kódu. Napríklad trieda nemôže používať všetky funkcie jej nadradenej triedy, alebo aplikácia môže vytvoriť niekoľko objektov rovnakej triedy, ale používa rôzne podskupiny funkcií danej triedy v rôznych kontextoch. V obidvoch týchto príkladoch programátor vyžaduje hierarchiu tried odlišnú od tej stanovenej vývojárom. V tejto práci je definovaná štruktúra konceptu tak, aby zdôrazňovala pojmy, ktoré programátor skutočne využil. To poskytuje cenné vodítko pri úpravách hierarchie tried. Táto analýza poskytuje taký typ transformácii, že hierarchia tried sa postupne približuje názoru programátora a jeho pohľadu na doménu problému.

Ducasse, Rieger a Demeyer sa vo svojom výskume venujú technike pre detekciu duplicitného kódu založenej na jednoduchom porovnávaní reťazcov. Po odhalení zhodných riadkov kódu vizualizujú výsledky porovnania pomocou diagramu rozptylu [33]. Pri programe s n riadkami kódu vizualizujú nájdené zhody do matice s rozmerom NxN tak, že bodka sa nachádza na mieste (i, j) len vtedy, keď riadok i je zhodný s riadkom j. Táto práca slú-

žila ako základ pre [34], kde sa nachádza predbežný návrh nástroja slúžiaceho na podporu refaktoringu odstraňujúceho duplicitný kód. Autori upozorňujú, že plnú automatizáciu je možné využiť len v jednoduchých prípadoch, vo väčšine prípadov však bude potrebný zásah programátora.

Sweeney a Tip vyvinuli automatizovaný prístup k detekcii nevyužívaných tried v C++ aplikáciách [77]. Trieda t je definovaná ako nevyužívaná ak v aplikácii nie je objekt, ktorý obsahuje t také, že hodnota t môže mať vplyv na vonkajšie správanie aplikácie. Samozrejme, že zisťovanie týchto údajov o nevyužívaných triedach dláždi cestu jednoduchému refaktoringu, ktorý ich odstraňuje. Tento typ refaktoringu sa zdá byť jednoduchý, ale dosiahnuté výsledky sú netriviálne. Z testov vyplynulo, že v priemere 12,5% tried bolo nájdených ako nevyužívaných a priemerná obsadenosť pamäti nevyužívanými triedami bola 4,4%. To naznačuje, že výskum refaktoringu je stále ešte v plienkach, a že je možné dosiahnuť ešte veľmi veľa s použitím pomerne jednoduchých techník.

Maruyama a Shima prezentujú prístup k metóde refaktoringu založený na spôsoboch využívania knižníc (frameworkov) [55]. Základom je, že metóda v knižnici má závislosť na iných metódach z knižnice, ktoré vo väčšej či menšej miere zodpovedá využívaniu metód programátormi. Ak je metóda prepísaná (overriden) spôsobom, ktorý zachováva tieto závislosti naznačuje to, že interakcia s inými metódami sa nemení a môže byť zachytená v šablóne metódy. Naopak, ak je metóda prepísaná tak, že sa menia tieto závislosti, tak to naznačuje, že metóda predstavuje "prístupový bod" a je lepšie ju modelovať iným spôsobom. Experimentálne výsledky prezentované v [55] dokázali znížiť počet príkazov písaných programátorom až o 22%. Vzhľadom k tomu, že v prezentovanej metóde refaktoring pracuje v automatickom režime, môžu sa novo generované metódy javiť programátorovi ako nezmyselné alebo zbytočné. Výsledky tohto prístupu môžu byť cenné, lebo história prevedených zmien dáva programátorovi nepriamu kontrolu nad tým, čo sa pri refaktoringu menilo.

Je potrebné sa zamyslieť, aký typ reštrukturalizácie môže vývojár chcieť vykonať, aby spravil systém pružnejším a schopným implementovať novú požiadavku. Vývojár zvyčajne má architektonický pohľad na to, ako si želá, aby sa program vyvíjal. Je na vyššej úrovni: napríklad jednoducho vytvorí novú triedu alebo poupravuje existujúce metódy. Medzi najzaujímavejšie a najnáročnejšie kategórie transformácií vyššej úrovne ktoré vývojár môže chcieť, patrí napríklad zavedenie návrhového vzoru [80]. Návrhové vzory zvyčajne uvoľňujú závislosti medzi súčasťami aplikácie, čo umožňuje určité typy úprav programu s minimálnou zmenou samotného programu. Napríklad inštancia triedy Product v rámci triedy Creator by mohli byť nahradené aplikáciou vzoru Factory. To by umožnilo triede Creator, aby bola rozšírená o inštanciáciu podtriedy triedy Product bez významnej zmeny existujúceho kódu.

Účelom tejto kapitoly bolo priniesť zjednodušený pohľad do problematiky refaktoringu, ktorá je v tejto práci preberaná okrajovo ako pomocný problém. Kapitola ponúkla úvod do problematiky a základný prehľad existujúcich prístupov a výsledkov výskumov, ktoré boli ďalej využité pri riešení témy.

Kapitola 5

Teoretické východiská práce

Táto kapitola prináša detailnejší opis vybraných článkov, na ktorých kombinácii je postavená navrhnutá metodika. Podkapitola: Analýza mechanizmov pre vzájomnú výlučnosť prináša pohľad na analýzy nutné k zisteniu informácií o bežiacich vláknach programu a o zámkoch zabezpečujúcich vzájomnú výlučnosť a opisuje článok [66]. Druhá podkapitola prezentuje článok [61] prinášajúci pohľad na metriky kódu určené k diagnostike existujúcich zdrojových kódov za účelom odporúčania vhodných návrhových vzorov. Tento článok ilustruje techniku odporúčania vhodných návrhových vzorov, ktorá bola taktiež inšpiráciou pre navrhnutú metodiku opísanú neskôr.

5.1 Analýza mechanizmov pre vzájomnú výlučnosť

Na to aby bolo možné pracovať s paralelnými návrhovými vzormi, je nutné byť schopný detekovať prístup jednotlivých vlákien k výrazom zdrojového kódu. Riešenie tohoto problému je veľmi blízke analýze zámkov pomocou statickej analýzy. Nasledujúci text vychádza z článku [66] a opisuje použitý prístup, ktorý slúžil ako inšpirácia k riešeniu. Ottom navrhnutý prístup k statickej analýze je založený na nasledujúcich krokoch.

1. Algoritmus analýzy zámkov počíta množinu držaných zámkov pre každý príkaz programu.
2. Výsledky algoritmu z bodu 1 sú použité na (i) získanie informácií o poradí v akom sú zámky získavané a (ii) na zoskupenie príkazov do blokov chránených rovnakým zámkom.
3. Sú extrahované presné informácie o blokoch a obmedzeniach pomocou points-to a MHP analýzy [51].
4. Na základe dát z predošlých krokov je možné vyhľadávať vzory kódu, ktoré môžu viezť k synchronizačným problémom.

Analýza zámkov

Použitý algoritmus prechádza celý zdrojový kód od prvého príkazu funkcie main. V prípade volania inej funkcie je toto volanie analyzované s ohľadom na stav zámkov v čase volania. Pretože každá metóda môže byť volaná v rôznych kontextoch (tj. s rôznymi stavmi zámkov pri volaní), je potrebné vykonať analýzu zámkov pre každé z volaní samostatne. Obrázok 5.1 zobrazuje príklad analýzy pre jednoduchý Java program. Kedykoľvek je vytvorený zámok, názov premennej tohto zámku je pridaný do množiny zámkov. Pri uvoľnení zámku je príslušné meno premennej odstránené z množiny zámkov. Analýza zámkov je klasický problém analýzy toku dát (dataflow problem). Účelom analýzy toku dát je získanie potrebných informácii o každom príkaze zdrojového kódu. [62] To býva väčšinou riešené na základe grafu toku riadenia (control flow graph) s pomocou výpočtu množín GEN(s), KILL(s), IN(s) a OUT(s) pre všetky príkazy s. Množina GEN(s) obsahuje zoznam príkazov ktoré vytvárajú zámky, množina KILL(s) obsahuje príkazy ktoré uvoľňujú zámky. Množiny IN(s) a OUT(s) obsahujú zámky ktoré sú držané pred a po príkaze `s`. Analýza zámkov slúži na dva účely: na výpočet obmedzení a blokov programu, ktoré sú popísane v nasledujúcich dvoch odstavcoch.

```
                                          GEN  KILL  lockset
static void main (String[] args) {       {}   {}    {}
  foo();          /* context c1 */        {}   {}    {}
  lock(o)  {                              {o}  {}    {o}
    foo();       /* context c2 */         {}   {}    {o}
  }                                       {}   {o}   {}
}                                         {}   {}    {}

                                                     c1   c2
static void foo() {                       {}   {}    {}   {o}
  lock(p) {                               {p}  {}    {p}  {o, p}
    /* ... */                             {}   {}    {p}  {o, p}
  }                                       {}   {p}   {}   {o}
}                                         {}   {}    {}   {o}
```

Obrázok 5.1: Analýza zámkov jednoduchého programu.

Obmedzenie je výraz $a \rightarrow b$, kde a a b sú zámky. $a \rightarrow b$ znamená, že v nejakom bode programu je zamykaný zámok b pokým zámok a je zamknutý. Inak povedané, obmedzenia opisujú poradie v akom sú zámky zamykané. Obmedzenia sú nevyhnutné na výpočet potenciálnych zámkových cyklov, ktoré môžu viesť k zablokovaniu programu (deadlock).

Obmedzenia môžu byť spočítané z množín zámkov vypočítaných pre analyzovaný zdrojový kód nasledovne: Ak v nejakom bode programu je množina zámkov $\{a\}$ nasledovaná množinou zámkov $\{a, b\}$, tak je vytvorené obmedzenie $a \rightarrow b$. Pre program z obrázku 5.1, máme iba jedno obmedzenie a to $o \rightarrow p$, ktoré platí v tele funkcie foo() pri volaní v kontexte c2.

Blok je sekvencia príkazov, ktoré sú časťou funkcie a sú synchronizované rovnakou sadou zámkov, alebo nie sú vôbec synchronizované. Bloky sú základnou entitou kódu vo

vzťahu k synchronizácii a môžu byť ľahko vypočítané z množín zámkov: sekvencia príkazov, ktoré držia rovnakú sadu zámkov tvorí jeden blok. Použitie blokov umožňuje ďalšie pohľady na kód, ako napríklad: ktoré bloky kódu sú synchronizované rovnakým zámkom, alebo či dva bloky kódu môžu byť vykonávané paralelne.

Analýza ukazovateľov je potrebná na určenie, či dve premenné v_1 a v_2 ukazujú na rovnaký objekt v pamäti. Toto nastane v prípade, ak prienik príslušných points-to množín je neprázdny $PS(v_1) \cap PS(v_2) \neq \emptyset$. Pomocou analýzy ukazovateľov dokážeme určiť či: (i) rovnaký objekt je používaný dvoma rôznymy blokmi a či (ii) dva rôzne zámky môžu v skutočnosti byť jeden a ten istý objekt v pamäti.

MHP analýza [51] zisťuje či dva kusy kódu môžu byť vykonávané paralelne. MHP analýza dvoh funkcií m_1 a m_2 vracia $true$ ak m_1 a m_2 môžu byť vykonávané paralelne, čo však neznamená že sa tak niekedy stane a vracia $false$ ak sa nikdy nemôže stať, že m_1 a m_2 pobežia paralelne.

5.1.1 Využitie analýzy zámkov

Pomocou výsledkov statickej analýzy zámkov ktoré nám poskytujú podrobné informácie o zdrojovom kóde dokážeme definovať detekčné algoritmy ktoré dokážu detekovať synchronizačné chyby v analyzovanom zdrojovom kóde. Princípy z nižšie uvedených detekcií, boli využité pri návrhu inovatívneho spôsobu zápisu paralelných návrhových vzorov. Nasledujúci text opisuje najčastejšie synchronizačné chyby a ich detekciu.

Cyklická závislosť medzi zámkami

Fragmenty kódu zobrazené na obrázku 5.2 ukazujú časti kódu ktoré sú vykonávané paralelne. Cyklická závislosť medzi zámkami je nutná podmienka zablokovania (deadlock). Cyklické závislosti medzi zamykaním zámkov môžu byť detekované pomocou analýzy obmedzení zdrojového kódu.

```
void X() {
  lock(a1) {
    lock(b1) {
      // ...
    }
  }
}
```

```
void Y() {
   lock(b2) {
     lock(a2) {
      // ...
    }
  }
}
```

Obrázok 5.2: Cyklická závislosť medzi zámkami.

Vyššie uvedený príklad obsahuje dve obmedzenia $a1 \rightarrow b1$ a $b2 \rightarrow a2$. Ak by point-to analýza ukázala, že premenné a1 a a2 ukazujú na rovnaký objekt a súčasne b1 a b2 ukazujú na rovnaký objekt, tak vyššie uvedený kód obsahuje cyklickú závislosť medzi zámkami, a tým pádom môže nasatať situácia, kedy sa skompilovaný program zasekne (deadlock).

Detekcia:

1. Definujme C ako množinu všetkých obmedzení programu.
2. Vytvorme orientovaný graf $G = (V, E)$, v ktorom $V := CaE := \{(c_1 \rightarrow c_2, C'_1 \rightarrow C'_2) \in C \times C : PS(C_2) \cap PS(C'_1) \neq \emptyset\ \}$. Inak povedané, každé obmedzenie tvorí vrchol grafu, hrana existuje medzi obmedzeniami v prípade, že druhý zámok prvého obmedzenia ukazuje na ten istý objekt v pamäti ako prvý zámok druhého obmedzenia.
3. Nájdime všetky cykly v grafe G.
4. Pre každý cyklus, určíme množinu zúčastnených blokov B.
5. Ak pre $MHP(b_1, b_2) = true$ pre každé $b_1, b_2 \in B$, tak označ cyklus ako cyklickú závislosť medzi zámkami.

Chýbajúci, alebo nesprávny zámok

Fragmenty kódu zobrazené na obrázku 5.3 ukazujú časti kódu ktoré paralelne pristupujú k zdieľanej premennej, avšak ich prístup nieje chránený rovnakým zámkom. Inak povedané, majme dva fragmenty kódu X a Y a zdielanú premennú x. Nech prístup k premennej x z fragmentu X je zabezpečený zámkom l_1 a prístup k premennej x z fragmentu Y je zabezpečený zámkom l_2 alebo žiadnym zámkom. Ak X a Y sú vykonávané paralelne, tak môže nastať problém súbežnosti (race condition), pretože prístup k zdieľanej premennej x nie je exkluzívny. Preto by fragmenty X a Y mali byť chránené rovnakým zámkom.

```
void X() {                  void Y() {
  lock(a) {
      x++;                     x++;
  }
}                           }
```

Obrázok 5.3: Chýbajúci zámok.

Detekcia: Pre každý blok b

1. Definujme B ako množinu blokov programu ktoré môžu byť vykonávané paralelne s blokom b: $B := \{b' : MHP(b, b') = true\}$.
2. Pre každý blok $b' \in B$, skontrolujme či bloky b a b' môžu pristupovať k zdieľanej premennej tak, že môže nastať konflikt zápisu alebo čítania.
3. Ak bloky b a b' sú chránené rozličnými zámkami, prípadne nie sú chránené vôbec tak označ dvojicu (b, b').

Veľmi všeobecná synchronizácia

Fragment kódu zobrazený na obrázku 5.4 ukazuje dve funkcie ktoré sú synchronizované aj keď nepristupujú k žiadnemu spoločnej zdieľanej premennej. Ak dve a viac funkcií

je synchronizovaných pomocou zdieľaného zámku a súčasne tieto funkcie nepristupujú k zdieľaným premenným, tak to môže spôsobovať zbytočné blokovanie vlákien vykonávajúcich tieto funkcie paralelne. Ak je potrebná synchronizácia na úrovni funkcií, tak každá funkcia by mala byť synchronizovaná pomocou vlastného zámku.

```
class Trieda {
  [MethodImpl(MethodImplOptions.Synchronized)]
  void X() { x++; }

  [MethodImpl(MethodImplOptions.Synchronized)]
  void Y() { y++; }
}
```

Obrázok 5.4: Veľmi všeobecná synchronizácia.

Detekcia: Pre každú triedu c

1. Definujme S ako množinu synchronizovaných funkcií triedy c.
2. Vytvorme orientovaný graf $G = (V, E)$, v ktorom $V := S$ a $E \subset S \times S$, pričom dvojice funkcií $(m_1, m_2) \in E$, ak m_1 a m_2 môžu pristupovať ku zdieľaným premenným. Inak povedané points-to analýza všetkých premenných z m_1 a m_2 má neprázdny prienik.
3. Ak graf G nieje spojitý, tak triedu c označ.

Uvedený text priniesol bližší pohľad na tématiku statickej analýzy mechanizmov pre vzájomnú výlučnosť, ktorá ukazuje možnosť zistenia informácií o bežiacich vláknach pomocou statickej analýzy. Výsledok tejto analýzy je využitý na detekciu nesprávneho použitia synchronizačných mechanizmov v analyzovanom kóde.

5.2 Metriky pre aplikáciu návrhových vzorov

Skúmaný článok [61] prináša pohľad na metriky kódu určené k diagnostike existujúcich zdrojových kódov, za účelom odporúčania vhodných návrhových vzorov. Definuje ich formálne, čo zjednodušuje a sprehľadňuje význam daných metrík.

Na to aby sme mohli formálne definovať metriky kódu, potrebujeme definovať použitý meta model, tj abstraktnú syntax objektovo orientovaného návrhu ktorá je zobrazená diagramom tried v jazyku UML (Unified Modeling Language). Na obrázku vidíme triedu, ktorá má metódy zložené zo sekvencie príkazov. Asociácia "Ďalší"na triede "Príkaz"označuje poradie príkazov vo funkciách a atribút "Text"označuje text príkazu. Pretože na ďalšie štúdium potrebujeme hlavne podmienené príkazy, konštruktory a volania metód, trieda "Príkaz"má tri zodpovedajúce podtriedy. ConditionalStmt je trieda pre podmienené príkazy a má dve časti; podmienku a telo. Napríklad podmienený príkaz "`If (flag == 1) instancia_auta = new Auto();`"sa skladá z podmienky "`(flag == 1)`"a tela "`instancia_auta = new Auto();`". Telo podmienky obsahuje príkaz alebo postupnosť príkazov, tj bloku príkazov. Príkaz `{instancia_auta = new Auto(); }` patrí do

skupiny ConstrnctorStmt (čo je trieda konštruktorov), pretože obsahuje konštruktor "new Auto()"triedy Auto. Každý konštruktor obsahuje informácie o triede ktorú vytvára, čo je znázornené vzťahom "Odkazuje"medzi Triedou a ClassStmt, triedu ConstructorStmt. Podobne, je namodelovaná trieda MethodInvocation, ktorá definuje odoslanie správy a volanie metódy iného objektu.

Nasledujúci text používa logické výrazy predikátovej logiky na definíciu metrík pomocou vyššie spomenutého meta modelu. Názov atribútu je použitý ako funkcia ktorá vracia hodnotu daného atribútu objektu, názov asociácie je použitý ako predikátový symbol ktorý vracia 1 ak dva objekty majú danú asociáciu. Napríklad text(obj1) = "Auto". Ďalej sú použité štandardné množinové operátory $\in$ (členstvo), $\cap$ (prienik), #(kardinalita: počet prvkov v množine), a tak ďalej and logické operátory ako napríklad $\exists$ (existenčný kvantifikátor), $\neg$ (negácia), $\wedge$ (logické a) a tak ďalej. Zápis množiny $\{x \in A|P(x)\}$ značí množinu ktorej každý prvok x je A a spĺňa podmienku P(x).

Základné definície matematického modelu

Na to aby sme mohli definovať metriky kódu pomocou matematickej logiky nad meta modelom, potrebujeme definovať základné definície.

Def.1 triedy a strom dedičnosti

$$\begin{aligned}
&Subclass(c, c') = inheritance(c, c') \vee \\
&\quad \exists c1 \in Class \bullet (inheritance(c, c1) \wedge inheritance(c1, c') \\
&Subclasses(c) = \{c' \in Class|Subclass(c, c')\} \\
&Superclass(c, c') = Subclass(c', c) \\
&Superclasses(c) = \{c' \in Class|Superclass(c, c')\} \\
&ITree(c) = Subclasses(c) \cup Superclasses(c) \cup \{c\} \\
&Included(c, t) = (ITree(c) = t)
\end{aligned}$$

$$\begin{aligned}
Distance(c, c') &= 0 && if\ c = c' \\
&= Distance(c1, c') + 1 && if\ inheritance(c, c1) \\
&= Distance(c, c1) + 1 && if\ inheritance(c1, c') \\
&= \infty && otherwise
\end{aligned}$$

Definícia triedy a stromu dedičnosti: Subclass(c) a Superclass(c) značia množiny pod tried a nadtried triedy c a ITree(c) značí celú hierarchiu dedičnosti do ktorej patrí trieda c = strom všetkých tried z ktorých dedí trieda c spolu s množinou všetkých treid dediacich od tried c. Predikát Included(c,t) je platný vtedy a len vtedy keď trieda c je zahrnutá do stromu dedičnosti t. Distance (c,c’) počíta kolko hrán sa nachádza medzi vrcholmi c a c’ v danom strome t.

Nasledujúca funkcia počíta minimálnu vzdialenosť medzi množinou tried cs v strome dedičnosti. Táto funkcia platí iba ak každý prvok množiny cs patrí do rovnakého stromu dedičnosti.

$$Diversity(cs) = min(\{Distance(c1, c2) \mid c1 \in cs \wedge c2 \in cs\})$$

Def.2 nahrádzanie metód

$$IsOverriding(m) = \exists c, c' \in Class, \exists m, m' \in Method \bullet$$
$$(aggregation(c,m) \wedge Subclass(c,c') \wedge aggregation(c',m') \wedge label(m) = label(m'))$$

$$InvokingMethod(m,m') = \exists c' \in Class, \exists mi \in MethodInvocation \bullet$$
$$(aggregation(m,mi) \wedge refer(mi,c) \wedge aggregation(c,m') \wedge label(mi) = label(m'))$$

$$NonOverridenMethod(c) = \{m \in Method \mid aggregation(c,m) \wedge$$
$$\neg\exists c' \in Subclasses(c) \exists m' \in Method \bullet (aggregation(c',m') \wedge label(m) = label(m'))\}$$

$$OverridingMethod(m) = \{m' \in Method \mid \exists c, c' \in Class \bullet$$
$$(aggregation(c,m) \wedge Subclass(c',c) \wedge aggregation(c',m') \wedge label(m) = label(m'))\}$$

Hodnota funkcie lsOverriding(m) je pravdivá vtedy a len vtedy, ak metóda m prepísala metódu super triedy, zatiaľ čo funkcia NonOverridenMethod(c) vracia metódy triedy c, ktoré sú definované v triede c a súčasne nie sú prepísané v žiadnej z podtried triedy c. Predikát lnvokingMethod(m, m') platí, keď je metóda m volá metódu m'. OverriddenMethod(m) je súbor metód, ktoré prepisujú metódu m. V príklade na obrázku 3, lsOverriding(C2.M2) = true, NonOverriddenMethod(CO) = {}, InvokingMethod (C2.M2, CO.M2) = True a OverridingMethod(CO.M2) = {C2.M2, C5.M2}.

Def.3 tranzitívny uzáver "agregácie"

$$aggregation*(x,y) =$$
$$aggregation(x,y) \wedge \exists z \bullet (aggregation(x,z) \wedge aggregation*(z,y))$$

Keďže podmienený príkaz môže vo svojom tele obsahovať ďalšie podmienené príkazy je vhodné definovať operáciu agregácie*

5.2.1 Metriky

Na to aby sme vedeli definovať kritéria na vkladanie vybraných návrhových vzorov, musíme definovať nasledovných 11 metrík pre podmienené príkazy a 9 metrík pre hierarchiu dedičnosti.

Metriky pre podmienené príkazy

Nasledujúce metriky vyjadrujú zložitosť vetvenia, výkon v podmienených príkazoch a silu závislosti. To umožňuje rozpoznať, kedy, kde a ktoré by mali byť použité návrhové vzory. Nech cst je inštancia triedy ConditionalStmt v nasledujúcich logických výrazoch, definujme metriky pre podmienené príkazy nasledovne.

CP.1 počet konštruktorov v tele podmieneného výrazu cst.

$$CP1(cst) = \#\{cs \in ConstructorStmt \mid \exists cb \in ConditionalBody \bullet$$
$$(aggregation(cst,cb) \wedge aggregation*(cb,cs))\}$$

CP.2 počet tiel podmieneného príkazu ktoré obsahujú aspoň jeden konštruktor.

$$CP2(cst) = \#\{cb \in ConditionalBody \mid \exists cs \in ConstructorStmt \bullet$$
$$(aggregation*(cst,cb) \wedge aggregation(cb,cs))\}$$

CP.3 počet inštancovaných tried v tele podmieneného príkazu.

$$CP3(cst) = \#\{c \in Class \mid \exists cb \in ConditionalBody \exists cs \in ConstructorStmt \bullet \\ (aggregation*(cst, cb) \land aggregation(cb, cs) \land aggregation(cs, c))\}$$

CP.4 počet stromov dedičnosti ktoré obsahujú triedy ktoré môžu byť inštancované v tele podmieneného príkazu.

$$CP4(cst) = \#\{Reachable(c) \in setofClass \mid \\ \exists cb \in ConditionalBody \exists cs \in ConstructorStmt \bullet \\ (aggregation*(cst, cb) \land aggregation(cb, cs) \land aggregation(cs, c))\}$$

CP.5 minimálna vzdialenosť tried ktoré sú inštancované v tele podmieneného príkazu.

$$CP5(cst) = Diversity(\{c \in Class \\ \exists cb \in ConditionalBody \exists cs \in ConstructorStmt \bullet \\ (aggregation*(cst, cb) \land aggregation(cb, cs) \land aggregation(cs, c))\})$$

CP.6 maximálny počet príkazov zahrnutých v tele podmieneného príkazu.

$$CP6(cst) = max(\{NumStmt(cb) \mid aggregation*(cst, cb)\}) \\ whereNumStmt(cb) = \\ \#\{stmt \in Statement \mid aggregation(cb, stmt)\}$$

Nasledujúci kód 5.5 prakticky ukazuje hodnoty metrík CP1, CP2 a CP3. CP1 = 5, keďže počet konštruktorov v príklade je 5. CP2 = 4, pretože prvý podmienený príkaz má vnorené ďalšie tri podmienené príkazy, ktoré rozširujú prvý podmienený príkaz o ďalšie tri telá. CP3 = 3, pretože sa inštancujú triedy A, B a C.

```
if(flag == 0) {
  var a = new A();
}else if(flag == 1){
  var b = new B();
}else if(flag == 2){
  var b = new B();
  var c = new C();
}else if(flag == 3){
  var c = new C();
}
```

Obrázok 5.5: Príklad podmieneného príkazu

Metriky CP1 až CP5 je možné aplikovať nielen na volania konštruktorov, ale aj na volania funkcií. Formálne definície týchto metrík, môžeme získať nahradením MethodInvocation za ConstructorStmt. Podobne ako je nižšie uvedená metrika CM1, dokážeme vytvoriť metriky CM2 až CM5 z metrík CP2 až CP5.

CM.1 počet volaní funkcií zahrnutých v tele podmieneného príkazu.

$$CM1(cst) = \#\{mi \in MethodInvocation \mid \exists cb \in ConditionalBody \bullet \\ (aggregation(cst, cb) \wedge aggregation*(cb, mi))\}$$

Metriky pre hierarchiu dedičnosti

Metriky pre hierarchiu dedičnosti je možné rozdeliť do dvoch skupín: prvá ktorá pokrýva metriky nad hierarchiou dedičnosti (skupina ICx), druhá pokrýva metriky nad funkciami v kóde (skupina IMx).

IC.1 a IC.2 počet funkcií ktoré sú nanovo zavedené do tried ktoré sú súčasťou hierarchie dedičnosti t, a ich pomer ku celkovému počtu tried v hierarchii dedičnosti t.

$$IC1(t) = \#\{m \in Method \mid \exists c \in Class \bullet \\ (m \in NonOverridenMethod(c) \wedge Included(c, t))\}$$

$$IC2(t) = \frac{IC1(t)}{\#\{c \in Class \mid Included(c, t)\}}$$

IC.3 počet funkcií ktoré prepisujú svojho predka a ktoré volajú svojho predka vo svojom tele.

$$IC3(t) = \#\{m \in Method \mid \exists c, c' \in Class, \exists m' \in Method \bullet \\ (IsOverriding(m) \wedge InvokingMethod(m, m') \wedge aggregation(c, m) \wedge \\ aggregation(c', m) \wedge Subclass(c', c) \wedge label(m) = lable(m') \wedge Included(c, t))\}$$

IC.4 pomer počtu tried ktoré obsahujú funkcie ktoré prepisujú svojich predkov a ktoré volajú svojho predka vo svojom tele, voči počtu tried v celej hierarchií dedičnosti t.

$$ICt4(c) = \frac{\#ClassWithOlMethods(t)}{\#\{c \in Class \mid Included(c, t)\}} \\ where \\ ClassWithOlMethods(t) = \{c \in Class \mid \exists c' \in Class, \exists m, m' \in Method \bullet \\ (IsOverriding(m) \wedge InvokingMethod(m, m') \wedge aggregation(c, m) \wedge \\ aggregation(c', m) \wedge Subclass(c', c) \wedge label(m) = lable(m') \wedge Included(c, t))\}$$

Nasledujúce metriky môžu byť definované nad funkciami, ktoré sú prepísané v niektorej triede dediacej od aktuálnej triedy. Matematicky vyjadrené, doména nasledovných metrík je $\{m \in Method \mid OverridingMethod(m) \neq \{\}\}$

IM.1 počet volaní konštruktorov z funkcii ktoré prepisujú funkciu m.

$$IM1(m) = \#\{cs \in ConstructorStmt \mid \exists stmt \in Statement, \\ \exists m1 \in OverridingMethod(m) \bullet (aggregation(m1, stmt) \wedge aggregation*(stmt, cs))\}$$

IM.2 pomer počtu tried ktoré obsahujú funkcie ktoré prepisujú funkciu m, voči počtu tried v podstrome hierarchie dedičnosti s koreňom v triede c.

$$IM2(m) = \frac{\#\{c \in Class \mid \exists m1 \in OverridingMethod(m) \bullet aggregation(c, m1)\}}{\#Subclasses(c1)}$$

$$whereaggregation(c1, m)$$

IM.3 počet tried ktorých inštancie sú vytvorené funkciami ktoré prepisujú funkciu m.

$IM3(m) = \#InstantiatedClass(m)$

$where$

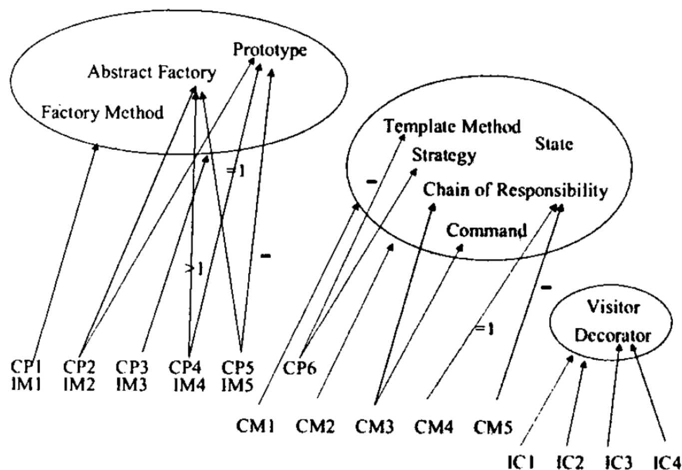
$InstantiatedClass(m) = \{c \in Class \mid \exists stmt \in Statement, \exists m1 \in OverridingMethod(m),$
$\exists cs \in ConstructorStmt \bullet (aggregation(m1, stmt) \wedge aggregation * (stmt, cs) \wedge refer(cs, c)\}$

IM.4 počet hierarchií dedičnosti do ktorých patria triedy ktorých inštancie sú vytvorené funkciami ktoré prepisujú funkciu m.

$$IM4(m) = \#\{lTree(c) \in SetofClass \mid c \in InstantiatedClass(m)\}$$

IM.5 minimálna vzdialenosť tried ktorých inštancie sú vytvorené funkciami ktoré prepisujú funkciu m.

$$IM5(m) = Diversity(InstantiatedClass(m))$$



Obrázok 5.6: Vzťahy medzi metrikami a návrhovými vzormi.

Obrázok 5.6 ukazuje súvislosti medzi metrikami a GOF návrhovými vzormi. Šípky bez popisu značia metriky, pri ktorých hodnota metriky je priamo úmerná váhe odporúčania použiť daný návrhový vzor. Napríklad, ak CP.2 a IM.2 majú vyššiu hodnotu, tak použijeme návrhový vzor Abstract Factory, alebo Prototype. Šípka označená znamienkom -, značí metriky, pri ktorých je hodnota metriky nepriamo úmerná váhe odporúčania. Boolove výrazy ako napríklad =1 a >1 značia prípady pri ktorých je odporúčané použitie zvoleného návrhového vzoru. Napríklad ak CP.4 = 1 a súčasne IM.4 = 1, tak je odporúčané použitie návrhového vzoru Prototype. Pri rozhodovaní o použitých návrhových vzoroch sa pozeráme na hodnoty všetkých metrík, nielen na vybrané metriky jednotlivo.

5.3 Zhrnutie

Táto kapitola priniesla detailnejší opis vybraných článkov, na ktorých kombinácii je postavená navrhnutá metodika opísaná v nasledujúcej kapitole. Podkapitola s názvom: Analýza mechanizmov pre vzájomnú výlučnosť priniesla pohľad na analýzy nutné k zisteniu informácií o bežiacich vláknach programu a o zámkoch zabezpečujúcich vzájomnú výlučnosť. Analýzy popísané v danom článku slúžili ako inšpirácia pre navrhovaný systém zápisu paralelných návrhových vzorov. Druhý prezentovaný článok priniesol pohľad na metriky kódu určené k diagnostike existujúcich zdrojových kódov, za účelom odporúčania vhodných návrhových vzorov. Tento článok ilustroval techniku odporúčania vhodných návrhových vzorov, ktorá bola taktiež inšpiráciou pre navrhnutú metodiku opísanú nižšie.

Kapitola 6

Návrh spôsobu zápisu návrhových vzorov

Ako už bolo spomenuté v úvode, hlavným prínosom práce je návrh spôsobu zápisu návrhových vzorov, ktorý umožní asistované vkladanie paralelných návrhových vzorov do existujúcich paralelných zdrojových kódov. Zdrojový kód vytvorený s pomocou návrhových vzorov je efektívnejší, čitateľnejší, jednoduchšie spravovateľný a tým pádom aj spoľahlivejší.

Na to aby bolo možné splniť požiadavky uvedené vyššie je potrebné byť schopný pracovať s návrhovým vzorom automaticky, čo znamená že návrhový vzor musí byť definovaný formálne. Druhá požiadavka je, že návrhový vzor musí obsahovať špecifikáciu miesta, kam sa má vložiť. Tretia požiadavka je, že špecifikácia návrhového vzoru musí umožňovať jeho automatické vkladanie do existujúcich zdrojových kódov. Na základe týchto požiadaviek definujme návrhový vzor ako dvojicu (P, d) kde P je množina preconditions, určujúca vhodné umiestnenie vzoru, d je samotný popis vzoru. Množina P vychádza z analýzy kódu opísanej v nasledujúcej kapitole, ktorá umožňuje vytvárať dopyty nad existujúcim zdrojovým kódom, ktorých výsledok je miesto v zdrojovom kóde. Špecifikácia vzoru definuje jeho štruktúru a správanie a je opísaná v kapitole nižšie.

6.1 Analýza kódu

Na začiatok špecifikácie použitého algoritmu, je potrené zhrnúť použitú notáciu. Použitý algoritmus berie ako vstup uzavretý program s hlavnou metódou označenou m_{main}. Definujme $\mathbb{M}$ ako označenie množiny všetkých funkcií, ktoré môžu byť dosiahnuteľné z m_{main}. $\mathbb{M}$ môže byť jednoduchou nadaproximáciou, napríklad vypočítaný analýzy hierarchie tried. Definujme $m_{start} \in \mathbb{M}$ ako označenie pre funckiu Start() triedy `System.Threading.Thread`, ktorá slúži na explicitné vytvorenie nového vlákna. Definujme $\mathbb{I}$ množinu všetkých volaní funkcií v tele každej z metód $m \in \mathbb{M}$. Definujme $\mathbb{H}$ množinu všetkých alokácií objektov v tele každej z metód $m \in \mathbb{M}$. Definujme $\mathbb{V}$ ako označenie množiny všetkých lokálnych premenných referencovanými metódami $m \in \mathbb{M}$. Definujme $\mathbb{F}$ ako označenie množiny všetkých premenných aktuálneho objektu referencovanými metódami $m \in \mathbb{M}$. V prípade polí, sa nahradzujú objekty na indexe samostatnými virtuálnymi premennými s indexom v názve.

Definujme $\mathbb{P}$ ako označenie množiny všetkých príkazov použitých v metódach $m \in \mathbb{M}$, do tejto množiny patria volania konštruktorov objektov, volania funkcií, čítanie a zápis premenných z a do pamäte. Predpokladáme, že každá metóda $m \in \mathbb{M}$ môžu byť synchronizované pomocou cez niektorý zo svojich svojich argumentov, čo zapíšeme pomocou sync(o, m), a súčasne neobsahuje žiadne iné synchronizované bloky kódu v tele. Obrázok 6.1 ukazuje aj relácie použité počas statickej analýzy: (cg) analýza grafu volaní, (pt) analýza ukazatelov, (esc) analýza vlákien a (mhp) analýza súbežnosti akcií.

$$
\begin{array}{lr}
(funkcia) & m \in M = m_{main}, m_{start}, ... \\
(lokalna\ premenna) & v \in V \\
(miesto\ alokacie) & h \in H \\
(mnozina\ miest\ alokacii) & [h_1 :: ... :: h_n] \in H^n \\
(prikaz) & p \in P \\
\\
(abstraktny\ objekt) & o \in \mathbb{O} = \mathbb{H}^0 \cup \mathbb{H}^1 \cup \mathbb{H}^2 \cup ... \\
(abstraktny\ kontext) & c, t, l \in \mathbb{C} = \mathbb{O} \times \mathbb{M} \\
\\
(graf\ volani) & \mathrm{cg} \subseteq (\mathbb{C} \times \mathbb{C}) \\
(analyza\ ukazatelov) & \mathrm{pt} \subseteq (\mathbb{C} \times \mathbb{V} \times \mathbb{O}) \\
(analyza\ vlakien) & \mathrm{ta} \subseteq (\mathbb{C} \times \mathbb{P} \times \mathbb{O}) \\
(analyza\ zamkov) & \mathrm{la} \subseteq (\mathbb{C} \times \mathbb{P} \times \mathbb{O})
\end{array}
$$

Obrázok 6.1: Notácia použitá v popise algoritmu analýzy kódu.

Na vytvorenie grafu volaní a analýzu ukazovateľov slúži k-object-sensitive analysis [x14]. Táto analýza je objektovo citlivá *(object sensitive)* čo znamená, že dokáže reprezentovať oddelené inštancie objektov, vytvorené rovnakým kódom, pomocou volaní z potencionálne rôznych inštancií voajúcich objektov. Inštancia objektu, alebo abstraktný objekt zapísaný pomocou $o \in \mathbb{O}$, je konečná množina miest alokácii objektu, ktoré sú zapísané pomocou $[h_1 :: ... :: h_n]$. Prvé miesto alokácie h_1, špecifikuje miesto v kóde ktoré vytvorilo aktuálnu inštanciu v kontexte objektu ktorý bol alokovaný pomocou inštancie objektu $[h_2 :: ... :: h_n]$ a tak ďalej až po prvý alokovaný objekt. V prípade statických metód sa použije zápis [ ] ktorý znamená žiaden alokovaný objekt nad ktorým sa vykonáva zvolená statická funkcia.

Použitá analýza je aj kontextová *(context sensitive)* čo znamená, že dokáže analyzovať implementáciu metódy v rôznych abstraktných kontextoch. Abstraktný kontext $c \in \mathbb{C}$ je pár (o, m), kde o je inštancia objektu a m je metóda v ktorej použite kľúčového slova `this` vráti objekt m. Pre statické metódy $o = [\,]$. Táto analýza umožňuje definovať nasledovné relácie:

- $\mathrm{cg} \subseteq (\mathbb{C} \times \mathbb{C})$ kontextový graf volaní, obsahuje každú dvojicu $((o_1, m_1), (o_2, m_2))$ takú, že metóda m_1 inštancie objektu o_1 volá metódu m_2 nad inštanciou objektu o_2.
- $\mathrm{pt} \subseteq (\mathbb{C} \times \mathbb{V} \times \mathbb{O})$ výsledok analýzy ukazovateľov obsahuje množinu trojíc (c, v, o)

takých, že lokálna premenná v môže ukazovať na inštanciu objektu o v kontexte c.

- ta $\subseteq (\mathbb{C} \times \mathbb{P} \times \mathbb{O})$ výsledok analýzy vlákien obsahuje množinu trojíc (c, p, o) takých, že príkaz p inštancie objektu o je vykonávaný v kontexte c.
- la $\subseteq (\mathbb{C} \times \mathbb{P} \times \mathbb{O})$ výsledok analýzy zámkov obsahuje množinu trojíc (c, p, o) takých, že príkaz p inštancie objektu o je vykonávaný v kontexte c.

Relácie grafu volaní a analýzy ukazovateľov sa naplnia nasledovným spôsobom. Analýza začína naplnením kontextu statickej metódy `Main` ([ ], Main) a statických konštruktorov použitých tried ([ ], Class.ctor). Analýza predpokladá kladné celé číslo priradené ku každému miestu alokácie objektu, nazvané k-hodnota daného miesta. Uvažujme akékoľvek miesto alokácie $v = \mathbf{new}^h ...$ kde $h \in \mathbb{H}$ a $v \in \mathbb{V}$, v metóde m ktorá je dosiahnuteľná v kontexte (o, m). Potom analýza pridá trojicu $((o, m), v, h \oplus_k o)$ do relácie pt, kde $h \oplus_k o$ značí konečnú neprázdnu usporiadanú množinu miest alokácii objektu, ktorej prvý prvok je h a ďalej obsahuje zoradených $k-1$ miest alokácii v poradí v akom boli alokované. Pre príklad z obrázku 7.5, ktorého kontext ([ ], A) obsahuje alokáciu dvoch premenných t1 = $\mathbf{new}^{h_1}$ a t2 = $\mathbf{new}^{h_2}$ pribudnú nasledovné relácie do pt ((([ ], A),v1,[h_1]) a ((([ ], A),v2,[h_2]).

Ak $n(...)$ je volanie statickej metódy v dosiahnuteľnom kontexte (o, m), tak analýza prída dvojicu $((o, m), ([\,], n))$ do cg, za predpokladu, že kontext $([\,], n)$ je dosiahnuteľný. Ak $v.n(...)$ je inštančné volanie metódy, tak volaná metóda závisí od aktuálneho typu objektu v. Každá trojica $((o, m), v, [h_1 :: ... :: h_n]) \in$ pt môže označovať rôzne metódy v rôznych kontextoch. Analýza preto pridáva $((o, m), [h_1 :: ... :: h_n], n')$ do cg, kde n' označuje cieľ volania n nad objektom alokovaným pomocou h_1. Analýza predpokladá, že kontext $([h_1 :: ... :: h_n], n')$ je dosiahnuteľný. V našom príklade z obrázku 7.5, ktorého kontext ([ ], A) obsahuje volanie dvoch funkcií, `t1.Start()` a `t2.Start()`, analýza pridá nasledové dvojice do cg ((([ ],A),([h_1],m_{start})) a ((([ ],A),([h_2],m_{start})).

Výstupom analýzy vlákien je relácia, definovaná pomocou trojice (c, p, o), v ktorej príkaz p abstraktného objektu o je dosiahnuteľný v kontexte c. Kedže každé vlákno je v .Net vykonávané nad inštanciou objektu typu `System.Threading.Thread`, môžeme každé unikátne vlákno označiť pomocou inštancie objektu t.j. kontextu v ktorom je vykonávané.

Majme metódu m s unikátnym počiatočným príkazom p v abstraktnom kontexte c', volanú z metódy n kontextu c. Relácia pt, neobsahuje ani jednu lokálnu premennú v z metódy m. Ak má metóda neprázdnu množinu vstupných parametrov, tak relácia pt, obsahuje zjednotenie relácii pt pre všetky argumenty analyzovanej metódy. Abstrakcia haldy a relácia ta, obsahuje zjednotenie haldy a relácie ta zo všetkých príkazov volajúcich metódu m. Ak $m = m_{start}$ čo značí, že m je metóda `Start()` objektu typu `System.Threading.Thread`, ktorá je zodpovedná za vytvorenie nového vlákna, tak všetky príkazy nasledujúce po volaní metódy m v metóde n a kontexte c je označený ako dosiahnuteľný z novo vytvoreného vlákna.

Analýza zámkov je naplnená rovnakým algoritmom ako analýza vlákien. S tým rozdielom, že sa sleduje držanie a uvoľňovanie zámkov. V našom výskume, pre zjedodušenie uvažujeme iba zámok typu monitor, ktorého zamykanie je definované na obrázku 6.2.

Na základe takto získaných dát, dokážeme definovať miesta v kóde s nesprávnou synchronizáciou medzi vláknami, prípadne s nesynchronizovaným prístupom ku zdielaným premenným. Tieto dva prípady sú najčastejšími dôvodmi na vloženie návrhového vzoru a preto

```
MONITORENTER(mon) =
  if mon = Null then FAILUP(ArgumentNullException)
  elseif lockOwner(mon) = self then
    lockCount(mon, self) := lockCount(mon, self) + 1
    YIELDUP(Norm)
  elseif lockOwner(mon) = None ∧ Empty(readyQueue(mon)) then
    LOCK(self, mon)
    lockCount(mon, self) := 1
    YIELDUP(Norm)
  elseif interruptRequested(self) then THROWINTERRUPTEDEXCEPTION
  else
    readyQueue(mon) := readyQueue(mon) · [self]
    monObj(self) := mon
    execState(self) := Syncing
    YIELDUP(Norm)
```

Obrázok 6.2: Definícia zamykania použitého zámku.

sú popísané v tejto práci. Samozrejme existujú aj iné dôvody na vloženie paralelných návrhových vzorov do existujúcich zdrojových kódov, ale špecifikácia príslušných anti-vzorov sa dá získať jednoduchým rozšírením vyššie popísaného algoritmu. Špecfikáciou miesta v kóde na ktoré by sa mal aplikovať návrhový vzor je $p \in \mathbb{P}$ ktoré vieme jednoznačne definovať pomocou výrokov predikátovej logiky nad reláciami cg, pt, ta a la.

6.2 Špecifikácia vzoru

Špecifikácia vzoru využíva údaje zistené počas analýzy na definíciu miesta kam sa má vložiť vzor. Na to aby sme dokázali vložiť vzor potrebujeme minimálne jeden bod ktorého sa môžeme chytiť. Týmto bodom je miesto s vadným kódom a je označené značkou l. Nasledujúci text uvádza použitú formálnu špecifikáciu vzoru, ktorá je rozšírením existujúceho jazyka BPSL [78]. Rozšírenie spočíva v pridaní možnosti označovať metódy, atribúty, premenné a objekty značkami l, ktoré umožňujú priradiť k daným bodom zo vzoru precondition, ktorá určuje kam je vhodé daný vzor vložiť. Ďalej je jazyk rozšírený o množinu aktívnych vlákien TA a zámkov LA použitých v danom vzore.

Formálna špecifikácia návrhového vzoru používa existujúci formálny jazyk BPSL (Balanced Pattern Specification Language) [78], ktorý umožňuje špecifikáciu návrhových vzorov jak z pohľadu štruktúry, tak aj z pohľadu správania. Tento formálny jazyk sa radí do kategórie formálnych systémov, ktoré vhodným spôsobom kombinujú už existujúce formálne mechanizmy a prispôsobujú ich tak, aby ich bolo možné použiť na opis návrhových vzorov. K definícii štruktúry používa predikátovú logiku 1. rádu, k definícii správania temporálnu logiku akcie. na to aby mohol byť tento jazyk použitý pre potreby tejto práce, bolo nutné ho rozšíriť o informácie o vláknach a zámkoch. Toto rozšírenie je opísané v špecifikácii štruktúry.

6.2.1 Špecifikácia štruktúry

BPSL využíva k špecifikácii štruktúry vzoru podmnožinu predikátovej logiky 1. rádu (najmä premenné a predikáty), pretože vzťahy medzi účastníkmi vzoru môžu byť ľahko vyjadrené ako predikáty. Temporálne relácie medzi účastníkmi vzoru a ich správanie umožňuje definovať pomocou podmnožiny temporálnej logiky akcie (používa najmä akcie a premenné popisujúce stav). BPSL tak vhodným spôsobom kombinuje dva formálne systémy a vďaka tomu môžeme jedným jazykom definovať správanie i štruktúru vzoru. BPSL uvažuje ako hlavný stavebné prvky, ktoré odrážajú entity a relácie:

1. Triedy, atribúty, metódy, objekty, netypované hodnoty, ktoré nazýva primárnymi entitami.
2. Trvalé a dočasné relácie.
3. Akcie.
4. Všetky ďalšie prvky musia byť odvodené z trvalých relácií alebo primárnych entít.

Primárne entity jazyk chápe ako základné prvky štruktúry návrhových vzorov, pričom uvažuje možnosť objektov a netypovaných hodnôt vyskytovať sa ako parametre metód. Trvalé relácie definuje jazyk ako relácie, ktoré sa po vytvorení už nemôžu meniť, zatiaľ čo dočasné relácie sa môžu meniť v priebehu správania vzoru. Akcie jazyk opisuje ako atomické jednotky činnosti, kde zostavenie viac takýchto jednotiek definuje správanie vzoru.

K definícii štruktúry sú využité prvky predikátovej logiky 1. rádu, najmä premenné, logické spojky, existenčný kvantifikátor a predikáty. Premenné reprezentujú primárne entity, zatiaľ čo predikáty reprezentujú trvalé relácie medzi nimi. Nech triedy sú označené premennou C, atribúty A, metódy M, objekty O, netypované hodnoty V, označené body programe L, vlákna programu TA a zámky programu LA, potom tabuľka 6.1 popisuje možné trvalé relácie medzi účastníkmi vzťahu realizované ako predikáty.

6.2.2 Špecifikácia správania

U niektorých návrhových vzorov nestačí samotná definícia ich štruktúry, ale tiež je veľmi dôležitá definícia ich správania, tj. popis, ako účastníci vzoru spolu spolupracujú. BPSL opisuje správanie vzoru pomocou podmnožiny temporálnej logiky akcie. Jazyk opisuje správanie vzoru ako nekonečnú sekvenciu stavov, ktorými entity vzoru prechádzajú. Každý stav možno chápať ako kolekciu hodnôt stavových premenných (napr. hodnoty atribútov tried) a dočasných vzťahov medzi objektami.

Dvojica po sebe idúcich stavov sa nazýva prechod. Systém sa na začiatku nachádza v počiatočnom stave a postupne, ako sa vykonávajú jednotlivé akcie, prechádza jednotlivými stavmi. Akcia sú vyberané nedeterministicky a každá má stanovenú vstupnú podmienku, ktorá musí byť splnená, aby sa akcia uskutočnila. BPSL naviac oproti definícii temporálnej logiky akcie rozširuje sémantiku akcií. V BPSL sa počas vykonávania akcií nielen menia hodnoty stavových premenných, ale aj dočasné vzťahy medzi objektmi (môžu vznikať nové, zanikať existujúce).

Názov	**Doména**	**Význam**
DefinedIn	M × C	Metóda je definovaná v konkrétnej triede
	A × C	Atribút je definovaný v konkrétnej triede
ReferenceTo one (many)	C × C	Trieda obsahuje referenciu na jednu (viacej) inštancií triedy
Inheritance	C × C	Prvá trieda dedí z druhej
Creation	M × C	Metoda vytvára nové inštancie triedy
	C × C	Jedna z metód prvej triedy vytvára inštanciu druhej triedy
Invocation	M × M	Prvá metóda volá druhú metódu
Argument	C × M	Argumentom metódy je referencia na triedu
	V × M	Argumentom metódy je hodnota
Instance	O × C	Objekt je inštanciou danej triedy
Label	M × L	Metóda je označená značkou
	V × L	Hodnota je označená značkou
	O × L	Objekt je označený značkou
	A × L	Atribút je označený značkou

Tabuľka 6.1: Možné trvalé relácie medzi účastníkmi vzťahu realizované ako predikáty BPSL.

Dočasné relácie (TR) BPSL definuje ako dvojicu $TR(C1[cardinality1], C2[cardinality2])$, kde TR je názov relácie, $C1$, $C2$ sú triedy a kardinality reprezentujú počet inštancií daných tried, ktoré sú spolu vo vzťahu. Kardinalitou môže byť konkrétny rozsah nezáporných celých čísel (zapísaný v tvare [m ... n]) alebo [*], čo znamená akýkoľvek počet. V akciách sa potom môžu vyskytovať zápisy v tvare napr. $TR(o1, o2)$, čo znamená, že objekt o1 je vo vzťahu s objektom o2, $\neg TR(o1, o2)$, čo znamená, že objekt o1 nie už ďalej vo vzťahu s objektom o2, $TR(o1, C2)$ znamená, že objekt o1 je vo vzťahu so všetkými inštanciami triedy C2.

Akcie v BPSL sú definované ako zoznam vstupných parametrov, vstupné podmienky a tela (ako sa má zmeniť stav systému vykonaním akcie). Akcia A môže byť napr. Definovaná nasledovne: $Ao1, o2, p) : TR(o1, o2) \wedge o1.x \neq p \rightarrow \neg TR(o1, o2) \wedge o1.x' = p$, kde $o1$ je objekt triedy C1, $o2$ je objekt triedy C2, p je vstupný parameter akcie a x je atribút triedy C1. Je teda zrejmé, že výraz $TR(o1, o2) \wedge o1.x \neq p$ je vstupná podmienka akcie (ak nie je splnená, akcia sa nevykoná) a zvyšok výrazu je samotné telo akcie, x' znamená hodnotu atribútu x po vykonaní akcie. Ako už bolo povedané vyššie, objekty a hodnoty, ktoré sa akcie zúčastňujú, sú vybrané nedeterministicky. Preto sa akcia A vykoná pre všetky objekty tried C1 a C2, ktoré sú spolu v relácii.

BPSL nepoužíva vzájomne disjunktné množiny premenných v trvalých a temporálnych reláciách a akciách. Naopak kombinuje tieto množiny dohromady, tj. niektoré premenné, ktoré sa vyskytujú v definícii trvalých relácií, sa vyskytujú aj v definícii dočasných relácií a akcií. BPSL teda pozerá na návrhový vzor ako na kolekciu entít (tried, atribútov, metód, objektov, hodnôt), relácií (dočasných a trvalých) a akcií medzi nimi. Z formálneho hľadiska BPSL definuje návrhový vzor ako model $M =< E, R >$, kde E je univerzum entít a R je množina relácií (temporálnej / permanentných a akcií).

6.3 Príklad zápisu paralelného návrhového vzoru

V nasledujúcom texte bude zapísaný návrhový vzor Thread Safe Interface, popísaný vyššie v kapitole 3.3.2 pomocou navrhnutého spôsobu zápisu. Ako už bolo spomenuté na začiatku tejto kapitoly, tak bolo potrebné rozšíriť špecifikačný jazyk BPSL o novú množinu L ktorá obsahuje značky (labels) použité pri popise miest kam sa má daný vzor vložiť. Ďalej boj jazyk rozšírený o množinu preconditions, ktoré určujú miesto v analyzovanom zdrojovom kóde, kam je vhodné daný návrhový vzor vložiť. Prepojenie medzi definíciou miesta kam sa má vzor vložiť a popisom vzoru, umožňuje automatické vkladanie návrhových vzorov do existujúcich paralelných zdrojových kódov.

Tanuľka 6.2 obsahuje formálnu špecifikáciu návrhového vzoru Thread Safe Interface. V prvej časti tabuľky sú definované entity vystupujúce v systéme. V tomto prípade sa jedná o dve triedy *Client* a *Server*, ďalej atribut *serviceLock* ktorý slúži na zabezpečenie vzájomnej výlučnosti. V množine metód M musia existovať prvky *client_thread*, *service*, *service_imp*. Množina konkrétnych objektov O musí obsahovať dva objekty s a c. Množina premenných V je prázdna. A množina označení kódu obsahuje jeden prvok *label*.

$\exists\ Client, Server \in C$; $\exists\ serviceLock \in A$; $\exists\ client_thread, service, service_imp \in M$; $\exists\ c, s \in O$; $V = \emptyset$ $\exists\ label \in L$
$DefinedIn(serviceLock, Server) \wedge$ $DefinedIn(client_thread, Client) \wedge$ $DefinedIn(service, Server) \wedge$ $DefinedIn(service_imp, Server) \wedge$ $ReferenceToOne(Client, Server) \wedge$ $Invocation(service, service_imp) \wedge$ $Instance(c, Client) \wedge$ $Instance(s, Server)$ $Label(service_imp, label)$
$Locked(Server[1], serviceLock[1])$ $Waiting(service[0..*], serviceLock[1])$
$Init : \neg Locked(s, serviceLock) \wedge \neg Waiting(s, serviceLock)$ $Service(s) : \neg Locked(s, serviceLock) \rightarrow Locked'(s, serviceLock) \vee$ $Locked(s, serviceLock) \rightarrow Waiting'(s, serviceLock)$
$\exists\ p \in service_imp \wedge p = label$; $\exists\ t_1, t_2 \in TA\ \bullet\ \{t_1, t_2\} \subseteq (\mathbb{C} \times p \times \mathbb{O})$; $\nexists\ lock \in LA\ \bullet\ \{lock\} \subseteq (\mathbb{C} \times p \times \mathbb{O})$;

Tabuľka 6.2: Príklad formálneho zápisu paralelného návrhového vzoru pomocou navrhnutého spôsobu zápisu.

Duhá časť tabuľky obsahuje množinu trvalých relácii. Trieda *Server*, musí obsahovať atribút *serviceLock*. Trieda *Client* musí obsahovať funkciu *client_thread*. Treida *Server*, ďalej obsahuje funkcie *service* a *service_imp*. Trieda *Client* si drží referenciu na tredu *Server*. Inštancia treidy *Client* je pomenovaná c a inštancia triedy *Server* je pomenovaná

s. Nakoniec metóda *service_imp* je označená značkou *label*. Táto značka nám umožňuje priradiť tejto metóde precondition (výsledok analýzy kódu) ktorá určuje, kam by bolo vhodné aplikovať opisovaný návrhový vzor.

Tretia a štvrtá časť tabuľky definujú pomocou dočasných relácii správanie vzoru. V opisovanom vzore existujú dve dočasné relácie pomenované *Locked* a $Waiting$. Relácia *Locked* umožňuje definovať počiatočný stav, kedy prístup k implementácii funkcie *service_imp* nieje blokovaný zámkom a súčasne relácia $Waiting$ popisuje, že žiadne volanie funcke *service_imp* nieje zablokované. V prípade volania funkcie $Service(s)$ sa zmení stav objektu na *Locked* alebo volajúce vlákno prejde do stavu $Waiting$.

Posledná časť tabuľky opisuje miesto kam je vhodné vložiť návrhový vzor pomocou výrazov predikátovej logiky nasledovne. Existuje príkaz p funkcie *service_imp*, a tento príkaz vystupuje v predošlej špecifikácii pod názvom *label*. V relácii výsledkov analýzy vlákien programu TA existujú relácie t_1 a t_2 také, že príkaz p je v nich obsiahnutý. Neexistuje žiaden záznam v relácii výsledkov analýzy zámkov LA taký, že príkaz p je v ňom obsiahnutý.

6.4 Zhrnutie

Ako už bolo spomenuté v úvode, hlavným prínosom práce je návrh spôsobu zápisu paralelných návrhových vzorov, ktorý umožní asistované vkladanie paralelných návrhových vzorov do existujúcich paralelných zdrojových kódov. Na to aby to bolo možné splniť je potrebné byť schopný pracovať s návrhovým vzorom automaticky, čo znamená že návrhový vzor musí byť definovaný formálne. Druhá požiadavka je, že návrhový vzor musí obsahovať špecifikáciu miesta, kam sa má vložiť. Tretia požiadavka je, že špecifikácia návrhového vzoru musí umožňovať jeho automatické vkladanie do existujúcich zdrojových kódov. Na základe týchto požiadaviek je definovaný návrhový vzor ako dvojica (P, d) kde P je množina preconditions, určujúca vhodné umiestnenie vzoru a d je samotný popis vzoru. Množina P vychádza z analýzy kódu opísanej na začiatku tejto kapitoly, ktorá umožňuje vytvárať dopyty nad existujúcim zdrojovým kódom, ktorých výsledok je miesto v zdrojovom kóde. Špecifikácia vzoru definuje jeho štruktúru a správanie a je opísaná v druhej časti kapitoly.

Na základe dát získaných z analýzy v prvej časti kapitoly, je možné definovať miesta v kóde s nesprávnou synchronizáciou medzi vláknami, prípadne s nesynchronizovaným prístupom ku zdielaným premenným. Tieto dva prípady sú najčastejšími dôvodmi na vloženie návrhového vzoru a preto sú popísané v tejto práci. Samozrejme existujú aj iné dôvody na vloženie paralelných návrhových vzorov do existujúcich zdrojových kódov, ale špecifikácia príslušných anti-vzorov sa dá získať jednoduchým rozšírením vyššie popísaného algoritmu. Špecifikáciou miesta v kóde, na ktoré by sa mal aplikovať návrhový vzor je $p \in \mathbb{P}$, ktoré vieme jednoznačne definovať pomocou výrokov predikátovej logiky nad reláciami cg, pt, ta a la.

Špecifikácia vzoru využíva údaje zistené počas analýzy na definíciu miesta, kam sa má vložiť vzor. Na to aby bolo možné vložiť vzor je potrebný minimálne jeden bod v kóde. Týmto bodom je miesto s nevhodným kódom. K definícii štruktúry sú využité prvky predikátovej logiky 1. rádu, najmä premenné, logické spojky, existenčný kvantifikátor a predikáty. Premenné reprezentujú primárne entity, zatiaľ čo predikáty reprezentujú trvalé relácie medzi nimi. U niektorých návrhových vzorov nestačí samotná definícia ich štruktúry,

ale tiež je veľmi dôležitá definícia ich správania, tj. popis, ako účastníci vzoru navzájom spolupracujú. Použitý jazyk opisuje správanie vzoru pomocou podmnožiny temporálnej logiky akcie. Jazyk opisuje správanie vzoru ako nekonečnú sekvenciu stavov, ktorými entity vzoru prechádzajú.

Na ukázanie spôsobu zápisu paralelných návrhových vzorov bol rozšírený existujúci jazyk BPSL [78]. Rozšírenie spočíva v pridaní množiny označení L, ktorou je možné označiť metódy, triedy, premenné a atribúty. Toto označenie umožňuje druhé rozšírenie jazyka BPSL a to o vyjadrenie výrazu *precondition* pomocou výrazov predikátovej logiky. Na vyjadrenie výrazu *precondition* slúžia relácie TA a LA, ktoré obsahujú informácie o vláknach a zámkoch z analyzovaného kódu.

Nasledujúca kapitola opisuje návrh systému na vkladanie návrhových vzorov do existujúcich paralelných zdrojových kódov, ktorý overuje realizovateľnosť navrhnutého spôsobu zápisu.

Kapitola 7

Návrh systému na vkladanie návrhových vzorov do existujúcich paralelných zdrojových kódov

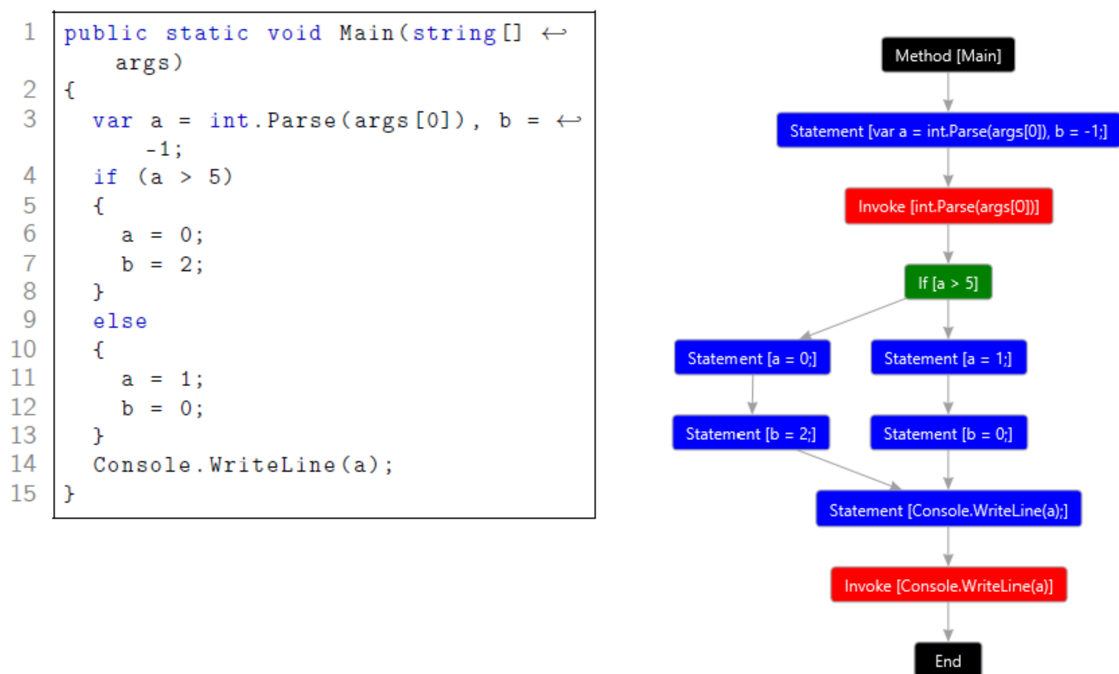
Navrhovaný systém na vkladanie návrhových vzorov do existujúcich paralelných zdrojových kódov sa skladá z troch častí. Analyzátora kódu, ktorý je zodpovedný za spracovanie zdrojového kódu a jeho reprezentáciu pomocou štruktúry Searchable Code Model, ktorá je výsledkom tejto práce. Druhým modulom je modul vyhodnocovania kvality kódu, ktorý má na vstupe Searchable Code Model analyzovaného zdrojového kódu a množinu špecifikácii návrhových vzorov, zapísaných vo formáte XML, ktorý vychádza z metodiky popísanej v predošlej kapitole. Tento modul vyhodnotí predložený kód oproti definovaným návrhovým vzorom a navrhne používateľovi vhodné návrhové vzory na zvýšenie kvality spracovávaného zdrojového kódu. Posledným modulom systému je modul refaktoringu, ktorý na základe špecifikácií návrhových vzorov vykoná príslušný refaktoring zdrojového kódu. Jednotlivé časti navrhovaného systému sú opísané v nasledujúcich kapitolách.

7.1 Analýza kódu

Navrhovaný Code Search Model poskytuje rozhranie na jednoduché vyhľadávanie v zdrojovom kóde. Je jednou zo základných častí navrhovaného systému na vkladanie návrhových vzorov do existujúcich paralelných zdrojových kódov. Code Search Model je opísaný v kapitole 7.1.2, táto kapitola opisuje algoritmus, ktorým sa plní. Ako bolo spomenuté vyššie, Code Search Model poskytuje rozhranie na dopytovanie nad štruktúrou existujúceho zdrojového kódu. Dopyty nad týmto modelom, slúžia na definíciu miest, kam sa majú vkladať návrhové vzory. Každý návrhový vzor je potom definovaný ako dvojica (*prec, spec*), v ktorej prvok prec slúži na špecifikáciu miesta kam sa má vložiť návrhový vzor popísaný prvkom spec. Analýza kódu je založená na statickej analýze a vo výsledku poskytuje informácie o všetkých triedach, metódach a premenných v analyzovanom zdrojovom kóde rozšírené o informáciach o vláknach pristupujúcim k daným objektom a ich premenným. Toto rozšírenie štrukturálnych informácii o informácie o vláknach poskytuje možnosť definovať miesta v kóde s nesprávnou, alebo neúplnou synchronizáciou, alebo prácou s vláknami.

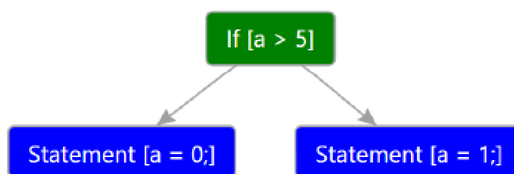
7.1.1 Analýza toku dát

Analýza toku dát je založená na grafe toku riadenia (control flow graph CFG), ktorý je definovaný ako orientovaný graf modelujúci všetky cesty, ktorými by mohol program prejsť počas jeho vykonávania [11], a je vytvorený pre každú funckiu, atribút objektu, konštruktor a lambda výraz. Každý uzol v grafe predstavuje príkaz, smerujúce hrany reprezentujú kroky v toku riadenia. Graf toku riadenia definuje dva špeicálne uzly: vstupný uzol v ktorom začína každý program a výstupný uzol, ktorý definuje ukončenie programu. Volania funkcií sú vždy reprezentované samostatným uzlom.



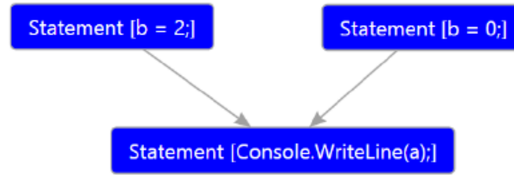
Obrázok 7.1: Príklad grafu toku riadenia s príslušným zdrojovým kódom.

Ak z uzlu grafu vychádzajú dve hrany ako na obrázku 7.2, tak to znamená, že počas behu programu bude vybraná vždy len jedna z hrán. Tento prípad nastáva pri podmienených príkazoch, akými sú napríklad príkaz `if`, alebo `switch`.



Obrázok 7.2: Viacero hrán vychádzajúcich z vrcholu obsahujúceho podmienený príkaz `if`.

Do uzlu grafu môže smerovať viacero hrán ako na obrázku 7.3, tento prípad nastáva napríklad za podmienenými príkazmi, akými sú napríklad príkaz `if`, alebo `switch`.



Obrázok 7.3: Viacero hrán smerujúcich do vrcholu grafu.

Intro-procedurálna analýza toku dát

Intro-procedurálna analýza toku dát sa zameriava na zhromažďovanie informácií o možných stavoch v každom bode programu v rámci jednotlivých funkcií (metód, atribútov objektov alebo lambda výrazov), zobrazených pomocou uzlov v grafe toku riadenia (CFG). Vygenerovaný graf toku riadenia umožňuje analýzu, ktorá berie do úvahy poradie príkazov v akom sa vykonávajú.

Použitá analýza toku dát zobrazená na obrázku 7.9 je založená na opakovanom výpočte výstupného stavu zo vstupného stavu pre každý uzol. Výpočet sa opakuje, dokým sa systém nestabilizuje. Algoritmus je založený na fronte, z ktorej sa po každom prechode odstráni spracovaný vrchol. Ak sa počas výpočtu zmení výstupný stav, tak všetci nasledovníci daného vrcholu, sú znova pridaný do fronty, pretože vyžadujú prepočet svojho stavu.

Keďže analýza toku dát je založená na behu programu, výstupný stav bloku b je funkciou vstupného stavu. Výstupný stav je spočítaný pomocou **transfer** funkcie nasledovne: $out_b = transfer_b(in_b)$. **Join** funkcia kombinuje výstupné stavy predchodcov do vstupného stavu aktuálneho vrcholu v grafe $in_b = join_{p \in predchodcovia_b}(out_p)$.

```
Vstup  : Množina príkazov (vrcholov z CFG)
Vystup: Stabilizovaná intro-procedurálna anaýza toku dát

for each príkaz S do
  OUT[S] = init(S); WORKQUEUE.Enqueue(S);
end
while WORKLIST not EMPTY do
  S = WORKLIST.Dequeue(); IN[S] = join(OUT[predchodca príkazu]);
  OUT[S] = transfer (IN[S]);
  if OUT[S] zmenené then
    WORLIST.Enqueue([nasledovník príkazu]);
  end
end
```

Obrázok 7.4: Algoritmus intro-procedurálnej analýzy toku dát

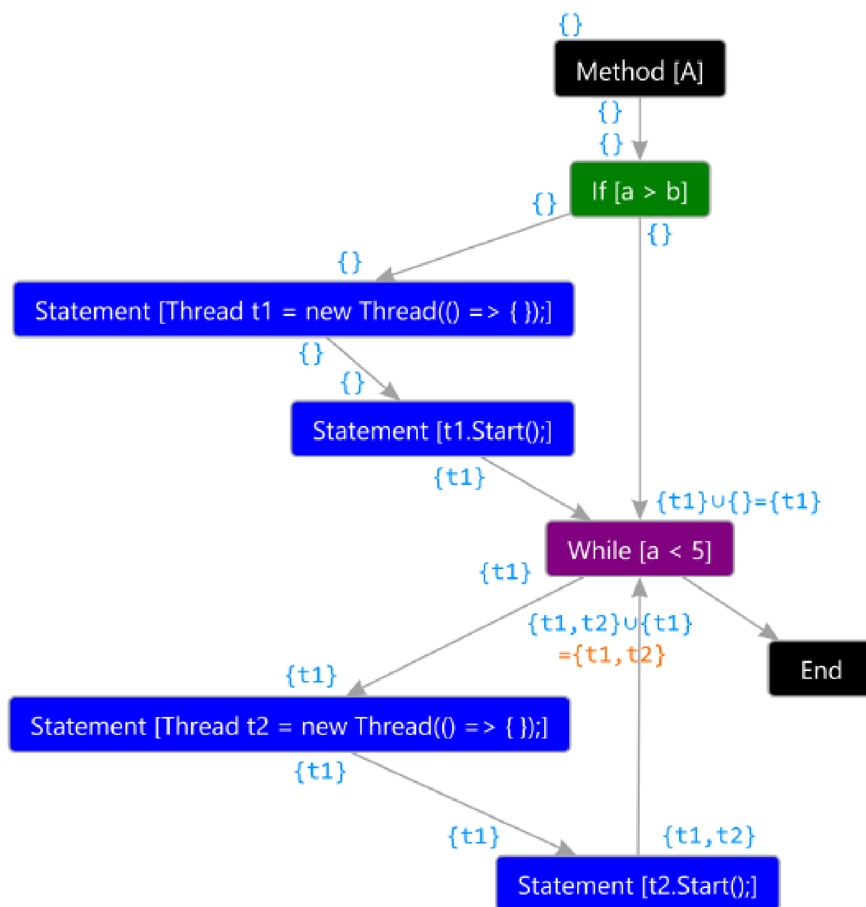
Každá analýza toku dát definuje konfiguráciu, **transfer** a **join** funkcie a počiatočný stav. Keďže použitá analýza toku dát je generická, tak umožňuje spočítať rôzne množiny stavov ako napríklad množinu bežiacich vlákien v každom bode programu, alebo množinu držaných zámkov pre každý bod programu.

Ak chceme napríklad získať množinu aktívnych vlákien pre každý vrchol grafu tak **stav** vrcholu bude obsahovať množinu aktívnych vlákien, počiatočný stav pre každý vrchol je prázdna množina. Funkcia **transfer** vracia vstupný stav, a ak daný príkaz vytvára nové vlákno, tak toto vlákno je pridané k vstupnému stavu. **Join** funkcia vracia množinu vlákien ktoré sú na výstupe všetkých predchodcov akutálneho vrcholu v grafe.

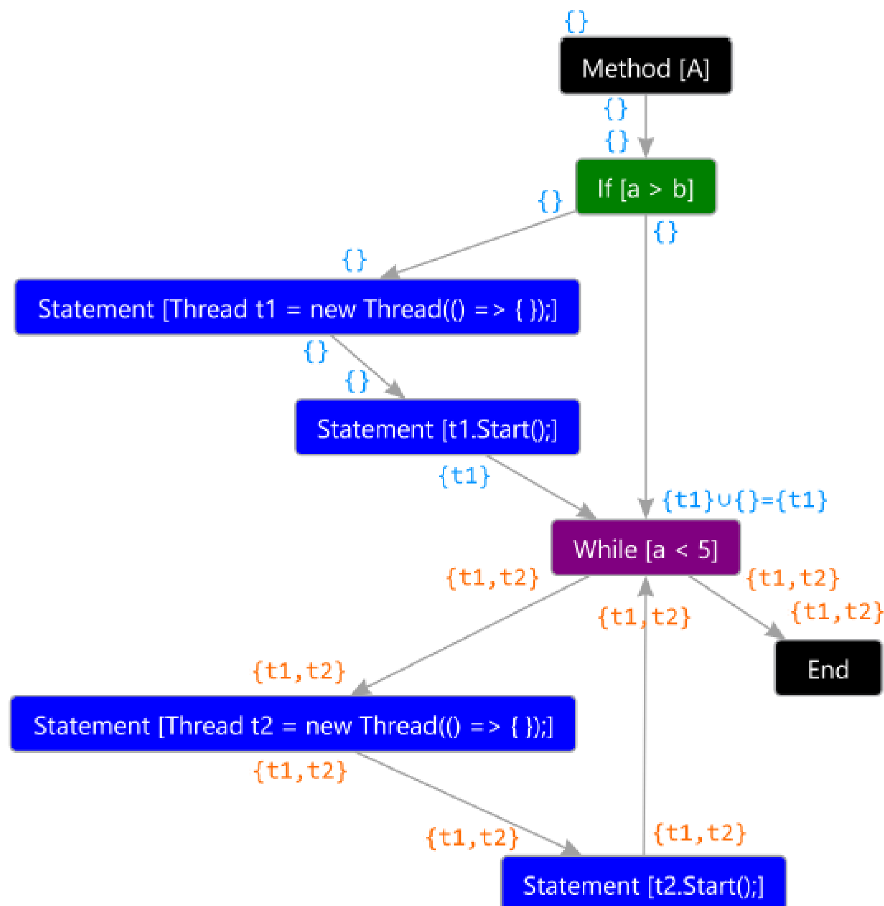
```
static void A(int a, int b)
{
    if (a > b)
    {
        Thread t1 = new Thread(() => { });
        t1.Start();
        }
    }

    while (a < 5)
    {
        Thread t2 = new Thread(() => { });
        t2.Start();
    }
}
```

Obrázok 7.5: Príklad analyzovaného kódu.



Obrázok 7.6: Graf toku riadenia analyzovaného kódu z obrázka 7.5 po prvom prechode algoritmu.

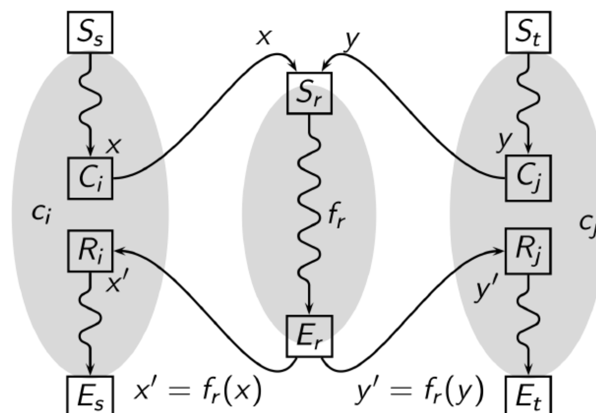


Obrázok 7.7: Graf toku riadenia analyzovaného kódu z obrázka 7.5 po druhom prechode algoritmu.

Na obrázku 7.5 je možné vidieť príklad analyzovaného kódu, ktorý spúšťa vlákna. Stav po prvom prechode algoritmu vidíme na obrázku 7.6. Ako vidíme, tak systém je nestabilný, pretože výstupný stav po volaní funkcie t1.Start() je zmenený oproti počiatočnému stavu ($\neq t1$), takže všetky nasledujúce príkazy, musia byť spracované ešte raz. Stav systému po druhom prechode algoritmu vidíme na obrázku 7.7. Po druhom prechode algoritmu sa nám zmenil stav u príkazov v tele príkazu while, ktoré sa rozšírili o informáciu o vlákne t2. Po treťom prechode algoritmu sa už nič nemení a systém je stabilizovaný.

Inter-procedurálna analýza toku dát

Inter-procedurálna analýza toku dát [52] rozširuje pôsobnosť analýzy za hranice funkcií a započítava vplyvy volania procedúr do ich stavu v rôznych kontextoch. Informácie o tokoch dát sú dedené z rôznych kontextov s cieľom získať kompletnú inter-procedurálnu analýzu tokov dát nad oddelenými grafmi. Táto analýza je obmedzená na vlákna, to znamená, že vytvorenia nových vlákien nie sú považované za volania funkcií.



Obrázok 7.8: Inter-procedurálna analýza toku dát.

Na obrázku 7.8 sú šedým pozadím vyznačené bloky kódu, c_i a c_j na stranách označujú volajúce funkcie s ich kontextami x a y. Šedá elipsa v strede označuje blok funkcie f_r s jej počiatočným uzlom S_r a koncovým uzlom E_r. Uzly C a R reprezentujú volanie (C = call) a návrat z funkcie (R = return) s príslušnými kontextami x a y. Označenie x' respektíve y' znamená kontext x prípadne y v čase návratu z funkcie f_r.

Inter-procedurálna analýza používa intro-procedurálnu analýzu viacerých grafov toku riadenia opísanú vyššie. Počas výpočtu algoritmu inter-procedurálnej analýzy, vrcholy grafu ktoré referencujú funkcie popísané samostatnými grafmi prenášajú vstupný stav volajúceho vrchola so vstupným stavom referencovanej funkcie. Výstupný stav referencovanej funkcie je prenesený do výstupného stavu volajúceho vrcholu v grafe. Rovnako ako v algoritme intro-procedurálnej analýzy, je výpočet ukončený po stabilizácii celého systému.

```
Vstup  : Množina grafov toku riadenia
Vystup: Stabilizovaná inter-procedurálna anaýza toku dát

while nastala zmena lubovolného OUT do
  for each graf C do
    for each príkaz S do
      if S is 'Invoke' then
        IN[ C[S] ] = join(OUT[predchodca príkazu]);
      end
      if S is 'Exit' then
        OUT[ REF[S] ] = join(OUT[predchodca príkazu]);
      end
    end
  end
end
```

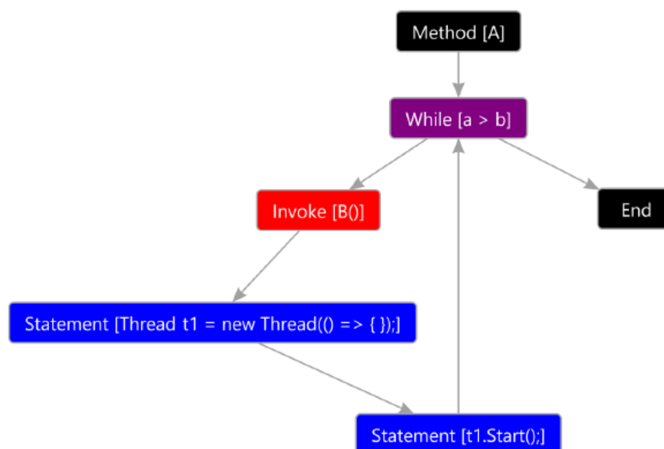
Obrázok 7.9: Algoritmus inter-procedurálnej analýzy toku dát

Na lepšie pochopenie poslúži príklad s nasledovnými parametrami. Ak sa výpočet do-

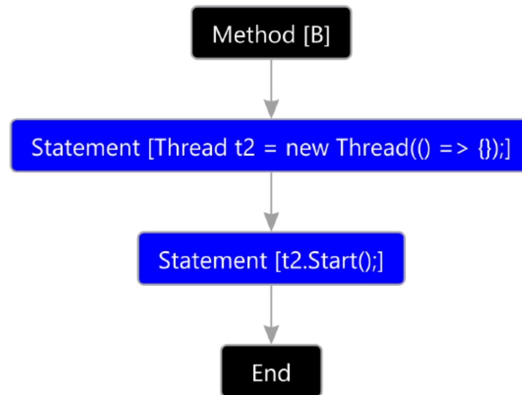
stane na vrchol *Invoke*, ktorý volá funkciu popísanú samostatným grafom, tak aktuálny vstupný stav je zlúčený so vstupným stavom volanej funkcie. Ak sa výpočet dostane na vrchol *End*, ktorý vracia výsledok z volanej funkcie, tak aktuálny výstupný stav je zlúčený s výstupným stavom volajúcej funkcie vo vrchole *Invoke*.

```
static void A(int a, int b)
{
    if (a > b)
    {
        B();
        Thread t1 = new Thread(() => { });
        t1.Start();
        }
    }
}
static void B()
{
    Thread t2 = new Thread(() => { });
    t2.Start();
}
```

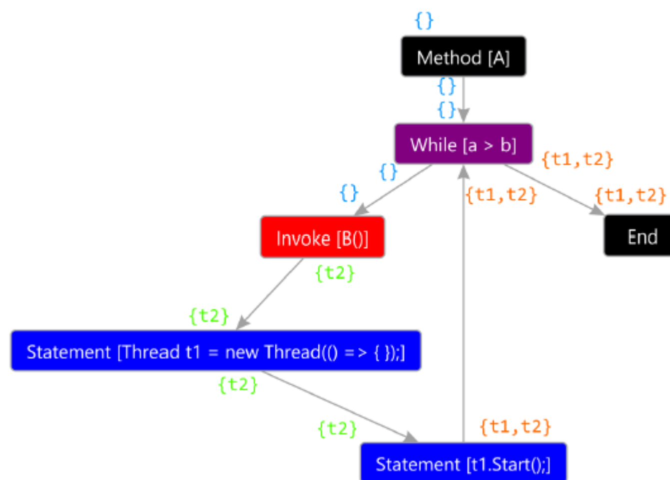
Obrázok 7.10: Príklad analyzovaného kódu.



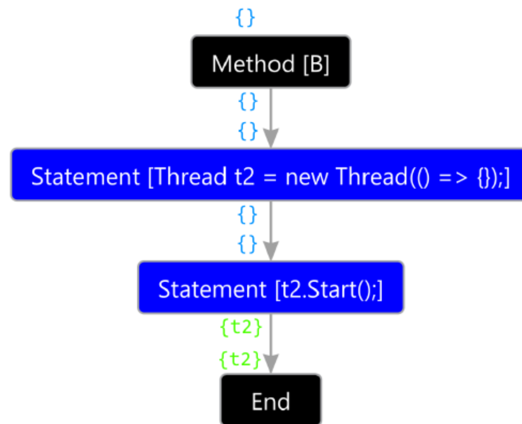
Obrázok 7.11: Graf toku riadenia analyzovaného kódu z obrázku 7.10.



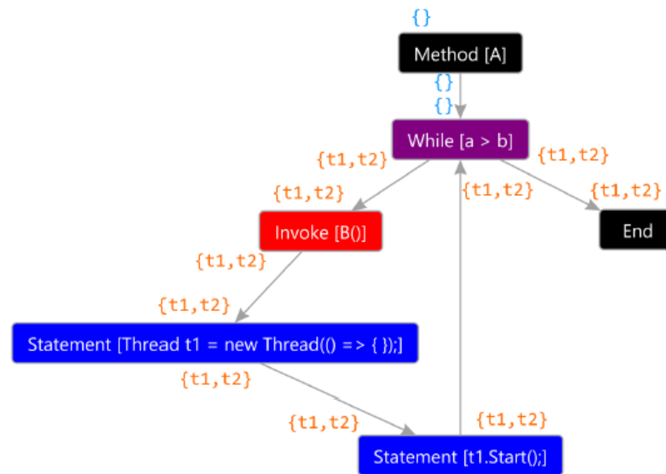
Obrázok 7.12: Graf toku riadenia analyzovaného kódu z obrázku 7.10.



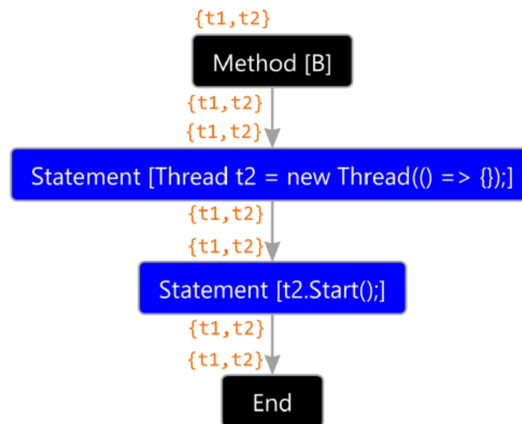
Obrázok 7.13: Graf toku riadenia analyzovaného kódu z obrázku 7.10, po prvom prechode inter-procedurálnej analýzy.



Obrázok 7.14: Graf toku riadenia analyzovaného kódu z obrázku 7.10, po prvom prechode inter-procedurálnej analýzy.



Obrázok 7.15: Graf toku riadenia analyzovaného kódu z obrázku 7.10, po druhom prechode inter-procedurálnej analýzy.



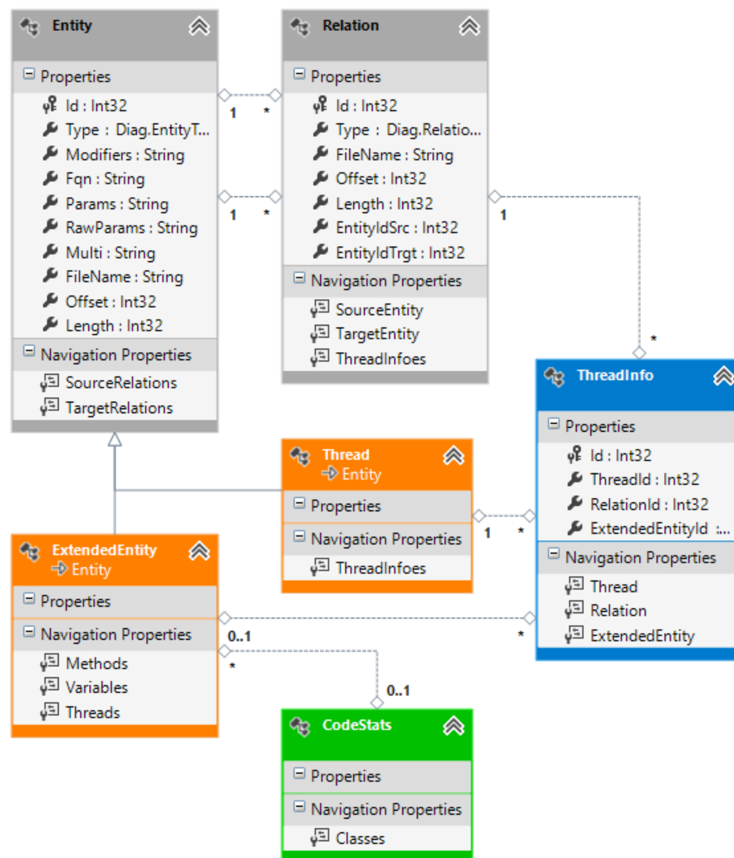
Obrázok 7.16: Graf toku riadenia analyzovaného kódu z obrázku 7.10, po druhom prechode inter-procedurálnej analýzy.

Ako vidíme na obrázku 7.16, systém sa stabilizoval po druhom prechode algoritmu. Takto získané informácie o vláknach sú uložené v modeli opísanom v nasledujúcej kapitole.

7.1.2 Použitý model

Existuje niekoľko prístupov k ukladaniu zdrojového kódu v dopytovateľných úložiskách, niektoré z nich sú založené na syntaktických stromoch (AST) a ďalšie na relačných databázach. Pri tvorbe našeho modelu sme nechceli začínať od nuly, ale skôr sme hľadali cestu rozšírenia nejakého existujúceho modelu, ktorý by spĺňal naše požiadavky. Základnou požiadavkou na použitý model je dostatočná expresívnosť aby použitý model dokázal popísať štruktúru analyzovaného kódu. Druhou požiadavkou je efektivita uloženia a škálovateľnosť, aby použitý model dokázal jednoducho a efektívne spracovať tisícky riadkov kódu. Navrhované riešenie je postavené na upravenej verzií úložiska SourcererDB [65], ktoré je založené na Chenovom [28] C++ entity-relationship (ER) metamodeli. Na rozdiel od modelu SourcererDB, ktorý je uložený v databázi, náš model je uložený iba vo forme objektov, avšak pri zachovaní pôvodnej expresívnosti, ktorá je dokonca rozšírená o informácie o vláknach. Náš model pracuje na úrovni jednotlivých príkazov zdrojového kódu, čo tím autorov SourcererDB nazýva "top level declaration granularity"[65]. Tento zápis poskytuje dobrý kompromis medzi nadmernou veľkosťou modelu pri jemnejšej granularite a obmedzenia analýzy ktoré by spôsobila hrubšia granularita zápisu. Nižšie popísaný model je rozšírená a upravená verzia Sourcerer metamodelu [65]. Ako je zobrazené na obrázku 7.17. Nami revidovaný model pridáva podporu pre informácie o vláknach a odstraňuje Java väzby, ktoré nie sú použiteľné v našom prostredí. Použitý model slúži na modelovanie informácií o štruktúre kódu a ďalších informácii extrahovaných z .Net C# projektov. Každý analyzovaný súbor obsahuje zdrojový kód entít definovaných v ňom, vzťahy medzi týmito entitami, a príslušné komentáre ktoré sú s nimi spojené.

Základ použitého modelu je prebraný z modelu databázy používaného nástrojom Sourcerer, konkrétne sa jedná o tabuľky entity a relácie. Naše objekty založené na týchto tabuľkách sú naplnené pomocou algoritmov používaných v nástroji Sourcerer. Na tomto základe



Obrázok 7.17: Diagram objektov syntaktického stromu rozšíreného o informácie o vláknach a typoch vytvorený pomocou analýzy kódu.

je vytvorená rozšírená dátová štruktúra, ktorá obsahuje triedy: ExtendedEntity, Thread, ThreadInfo. Trieda CodeStats slúži ako spoločný jedinečný vstupný bod pre všetky dopyty a umožňuje jednoduchý a efektívny prístup k tvorbe vyhľadávacích dopytov nad nižšie popísanou dátovou štruktúrou.

Trieda Entity Trieda Entity je prebraná z návrhu nástroja Sourcerer. Aj keď v našom modeli sme odstránili Java-špecifické závislosti nepoužiteľné v našom návrhu a taktiež sme zjednodušili systém referencií medzi entitami a súbormi ktoré ich obsahujú. Všetky typy entít použitých v našom modeli korešpondujú s explicitnými deklaráciami jazyka .net C# a mali by byť dostatočné na vymodelovanie všetkých objektovo orientovaných jazykov. Typy entít sú: NAMESPACE, CLASS, INTERFACE, ANNOTATION, FIELD, INITIALIZER, CONSTRUCTOR , METHOD, PARAMETER, LOCAL VARIABLE, ARRAY, TYPE, PRIMITIVE, ENUM, ENUM CONSTANT, INSTRUCTION. Každá entita je unikátne identifikovaná je plne kvalifikovaným menom (Fqn: string), názvom súboru v ktorom sa nachádza jej kód, a jej pozíciou v danom súbore. Každá entita je ďalej oanotovaná jej modifikátormi ako napríklad `public static` (ak ich má).

Trieda Relation Trieda Relation je taktiež prebraná z návrhu Sourcerera, avšak v našom modeli sme odstránili typy relácií, ktoré nie sú použiteľné v našom kontexte. Tabuľka

Relácia	Popis
CONTAINS	Fyzicky obsiahnuté
IMPLEMENTS	Implementácia rozhrania / rozšírenie
TYPEOF	Typ premenntej
RETURNS	Typ návratovej hodnoty z funkcie
READS	Čítanie premennej
WRITES	Zápis premennej
CALLS	Volanie funkcie
INSTANTIATES	Volanie konštruktora
THROWS	Deklarácia explicitného vyvolania výnimky
ANNOTATED BY	Anotácia
USES	Referencia
PARAMETRIZED BY	Typy parametrov
OVERRIDES	Preťaženie funkcie

Tabuľka 7.1: Typy relácií medzi entitami.

7.1 obsahuje použité typy relácii v našom modeli. Všetky uvedené relácie sú binárne, spájajúce dve entity navzájom (zdrojovú a cieľovú). Každá relácia je unikátne identifikovaná jej typom, plne kvalifikovanými menami jej zdrojovej a cieľovej entity a lokáciou v zdrojovom kóde kde sa nachádza. Výsledok tohto zápisu je, že ak nejaká metóda volá inú metódu v cykle, tak viaceré volania rovnakej metódy sú zapísané pomocou maximálne dvoch relácií. Keďže nás zaujímajú tri skupiny relácií: žiadne, jednoduché a dva a viac násobné, tak nám stačí si poznačiť maximálne dve relácie pri každom násobnom vzťahu.

Trieda Thread Trieda Thread definuje špeciálny typ entity, slúžiaci na uloženie informácií o vláknach v aplikácii. Na každé vlákno sa dá pozerať ako na špeciálny typ funkcie ktorá je vykonávaná paralelne s metódou `main` analyzovaného programu. Ako bolo spomenuté vyššie, počas tvorby AST a grafu volaní sú pomocou Intro a Inter-procedurálnej analýzy zistené informácie o vláknach pristupujúcim k jednotlivým entitám kódu a informácie o týchto vláknach sú uložené v objektoch typu Thread.

Pre naše potreby dopytov za účelom automatického doporučovania návrhových vzorov potrebujeme rozdeliť entity do troch skupín: entity ku ktorým nepristupuje žiadne vlákno, entity ku ktorým pristupuje práve jedno vlákno a entity ku ktorým pristupuje dve a viac vlákien. Toto rozdelenie zjednodušuje proces detekcie vlákien. Ak nejaké vlákno je tvorené v cykle alebo pomocou rekurzie, tak po pridaní druhej inštancie triedy Thread môžeme ukončiť detekciu ďalších vlákien. Ako bolo spomenuté vyššie, trieda Thread dedí od triedy Entity ktorá je jedinečne identifikovaná jej plne kvalifikovaným menom, lokáciou v zdrojovom kóde. Ak počas prechodu grafom volaní narazíme na vlákno ktoré je už definované v modeli, tak ho pridávame do modelu maximálne dva krát. V inom prípade ďalšie inštancie rovnakého vlákna ignorujeme, pretože tieto už nie sú potrebné.

Trieda ThreadInfo Trieda ThreadInfo slúži na definíciu špeciálneho typu relácie medzi triedami Thread a Entity. Táto relácia nám hovorí, že inštancia vlákna t, pracuje s príslušnými premennými na ktoré ukazuje daná relácia. Relácia ThreadInfo označuje všetky relácie spojené s príslušnými riadkami kódu ku ktorým pristupuje. Inštancie triedy ThreadInfo sú vytvorené počas tvorby triedy Thread tak, že sa zoberú všetky relácie spojené s hlavnou funkciou vlákna. Ak hlavná funkcia vlákna volá iné funkcie, tak relácie týchto funkcií, sú

tiež označené.

Triedy ExtendedEntity and CodeStats Triedy ExtendedEntity a CodeStats slúžia na zjednodušenie dotazov do modelu. Trieda CodeStats je jednotný vstupný bod pre všetky dopyty. Triedy ExtendedEntity vytvárajú stromovú štruktúru a sú naplnené počas tvorby entít počas prvého prechodu zdrojovými kódmi. Ak typ Entity je trieda, metóda alebo premenná tak je vytvorená príslušná inštancia objektu ExtendedEntity. Trieda ExtendedEntity rozširuje triedu Entity pomocou dodatočných navigačných premenných, ktorými sú: Methods, Variables a Threads. Premenná Methods je použitá iba ak ExtendedEntity je Trieda, a obsahuje referencie na všetky metódy danej triedy. Premenná Variables obsahuje všetky premenné použité v danej ExtendedEntity a premenná Threads obsahuje informácie o vláknach a jej účelom je zjednodušiť dopytovanie sa v špecifikácii vzorov.

7.2 Špecifikácia návrhových vzorov

Jedným z hlavných vstupov aplikácie je opis vzoru. Tento opis by mal byť nezávislý na platforme, ľahko rozšíriteľný, prenositeľný medzi aplikáciami, zrozumiteľný používateľom, pretože budú v tejto forme tvoriť definície ďalších vzorov, ale tiež ľahko validovateľný, aby aplikácia mohla spracovávať len správne definície. Preto by bolo vhodné zapisovať definície návrhových vzorov vo formáte XML, ktorý spĺňa všetky vyššie uvedené požiadavky. Validita XML dokumentov je zabezpečená pomocou XML schémy (napr. XSD), ktorú je možné použiť na kontrolu, či je definícia vzoru validná, alebo nie.

Popis vzoru sa skladá z dvoch hlavných častí: entít (značené pomocou <entities>) a vzťahov medzi nimi (značené pomocou <relations>). Za popisom vzoru nasleduje časť dopytov (značená pomocou <preconditions>). Časť <entities> obsahuje definíciu entít, ktoré sa vyskytujú v návrhových vzoroch. Sú tu najmä uložené informácie o atribútoch, metódach, vlastnostiach, konštruktore. Pri každej časti možno definovať viac modifikátorov. Pri každom elemente v množine <entities> je možné definovať atribut label, ktorý slúži ako väzobný bod pre množinu preconditions. Časť <relations> obsahuje definície relácií medzi entitami. Sú tu najmä uvedené informácie o asociáciách, dedičnosti, realizácii a pod. Každá táto relácia "spája"entity, ktoré sú popísané v predchádzajúcej časti, a umožňuje určiť ich kardinalitu. Časť <precoditions> umožňuje definovať množinu vyhľadávacích dotazov do Searchable Code Model, ktoré umožnia definovať miesta v analyzovanom zdrojovom kóde, kam je možné vkladať popisovaný návrhový vzor.

Aby opis vzoru spĺňal požiadavky na aplikáciu opísanú vyššie, je nutné, aby bol dostatočne abstraktný. Len vďaka tomu je potom možné vyhľadať podobné štruktúry vzoru v kóde, ktoré nie sú úplne rovnaké so vzorom, ale zároveň dostatočne relevantné. Toho je možné dosiahnuť tak, že v popise vzoru sú názvy entít (tried, rozhrania, abstraktných tried a pod.) A dátové typy metód, atribútov, vlastností uvedené len v abstraktnom poňatí.

Dátové typy sú uvedené všeobecným označením napr. Typ 1. Sú navyše očíslované, aby bolo možné viaceré typov v rámci entity (napr. návratové typy metód v triede), ale zároveň pri zachovaní možnosti definície príslušnosti vlastnosti (atribútu alebo metódy) k triede. Algoritmus nájdenia vhodných entít (popísaný nižšie) potom neberie do úvahy konkrétny "typ"triedy alebo metódy, ale jeho abstraktnú variantu. Algoritmus je toto schopný zohľadniť pri vyhľadávaní podobných entít v zdrojovom kóde. Vďaka tomu je popis návrhového

vzoru dostatočne všeobecný, ale zároveň umožňuje plne špecifikovať všetky vlastnosti vzoru.

7.3 Vkladanie návrhových vzorov

Kedže vkladanie návrhových vzorov je riešené len ako dodatočný problém, tak vkladanie vzorov je riešené pomocou jednoduchého mapovania existujúcej štruktúry kódu na návrhový vzor. Druhým možným riešením, ktoré nieje podrobnejšie študované je vytvorenie jednoúčelových transformácii / operácii refaktoringu ktoré priamo zavedú príslušný vzor. Nasledujúci text opisuje metódu navrhovania vkladanie vzorov pomocou mapovania štruktúry kódu a vzoru.

Aby bolo možné efektívne a správne vyhľadávať podobné entity vzoru v zdrojových kódoch, porovnávať ich a hodnotiť ich z hľadiska relevantnosti, je nutné, aby zdrojové kódy aj definícia návrhových vzorov boli v rámci algoritmu reprezentované rovnakou uniformnou štruktúrou. Zdrojové kódy sú načítané pomocou parsera použitého programovacieho jazyka, ktorého výstupom je abstraktný syntaktický strom (AST) kódu. Parsovanie kódov má aj ďalšiu výhodu. Súčasne s načítaním kódu parser vykonáva aj validáciu, lebo ak načítaný kód nie je validný, nemožno zostaviť korektne AST pre daný jazyk. Vďaka tomu sa po korektnom načítaní zdrojových kódov (a teda aj po úspešnom zostavení AST) prevedie už načítaná definícia vzoru taktiež na AST, ktorý je reprezentovaný zhodnou dátovou štruktúrou ako AST zdrojových kódov.

Ďalšou významnou časťou logiky aplikácie je algoritmus nájdenia vhodných entít. Jeho vstupom je popis vzoru a zdrojový kód, obaja sú reprezentované rovnakou štruktúrou. Cieľom je potom vyhľadať pre každú entitu vzoru zodpovedajúce entitu v zdrojovom kóde a jej ohodnotenie. Výsledkom algoritmu je zoznam podobných štruktúr zdrojového kódu k jednotlivým entitám vzoru. Tento algoritmus začína z bodov definovaných v množine preconditions, t.j. entít kódu, ktoré vyhľadávací algoritmus opísaný v predošlej kapitole analýza kódu označil ako kandidátov na vloženie daného vzoru.

7.4 Príklad zápisu vzoru pomocou navrhnutého systému

Majme popis návrhového vzoru Thread-Safe Interface z obrázku 7.18.

```xml
<?xml version="1.0" encoding="windows-1250"?>
<pattern name="Thread-Safe␣Interface" xmlns="http://www.w3.org">
    <entities>
        <class>
            <name>Class1</name>
            <attributes>
                <attribute>
                    <modifiers>
                        <modifier>private</modifier>
                        <modifier>readonly</modifier>
                    </modifiers>
                    <name>attribute1</name>
                    <type>object</type>
                </attribute>
            </attributes>
            <methods>
                <method label="label" originalName="method1">
                    <modifiers>
                        <modifier>private</modifier>
                    </modifiers>
                    <name>method_impl1</name>
                </method>
                <method>
                    <modifiers>
                        <modifier>public</modifier>
                    </modifiers>
                    <name>method1</name>
                    <code>
                    <![CDATA[lock(this) {this.method_impl1}]]>
                    </code>
                </method>
            </methods>
        </class>
    </entities>
    <relations></relations>
    <preconditions>
        <precondition label="label">
        <![CDATA[CodeStats.Types
            .SelectMany(type => type.Functions)
            .Where(func => ((func.Threads.Count > 1)
            && (GetLocks(func).Count() == 0));]]>
        </precondition>
    </preconditions>
</pattern>
```

Obrázok 7.18: Príklad špecifikácie vzoru.

Zápis vzoru uvedený na obrázku 7.18 ukazuje návrhový vzor Thread-Safe Interface, v ktorom na zabezpečenie správnej synchronizácie, klientské vlákna volajú iba funkcie verejného rozhrania. Funkcia rozhrania získa potrebný zámok a zavolá príslušnú implementačnú funkciu ktorá sa už nestará o zamykanie a môže slobodne volať iné implementačné funkcie.

Uviaznutie (self-deadlock) nemôže nastať pretože zámok sa získava iba raz na začiatku v rozhraní a pri rekurzívnom volaní funkcií je zámok získavaný iba raz, čo taktiež zvyšuje výkon programu.

Najdôležitejšou časťou použitého zápisu je element `<preconditions>` spoločne s atribútom `label` ktoré umožňujú definovať miesto v existujúcom zdrojovom kóde kam je vhodné daný vzor vložiť, súčasne s návrhom riešenia pomocou návrhového vzoru. V prípade vzoru Thread-Safe Interface, uvdený dopyt vracia všetky funkcie objektu, ku ktorým pristupuje viac ako jedno vlákno a súčasne daná funkcia nieje chránená žiadnym zámkom. Tento dopyt je zapísaný pomocou notácie Microsoft LINQ.

```
class client {
        server s = new server();
    void main(){
        thread t1 = new thread(() => thread_main);
        t1.start();
        thread t2 = new thread(() => thread_main);
        t2.start();

        thread.joinall(t1, t2);
    }
    void thread_main(){
        s.service();
    }
} // end client
class server {
    int x = 0;
    void service() {
        x++;
    }
} // end server
```

Obrázok 7.19: príklad analyzovaného kódu.

Ak tento vzor aplikujeme na príklad z obrázku 7.19, tak navrhnutý systém označí funkciu `service` značkou "label". V kroku 2 modul refaktoringu zistí, že funkcia označená značkou "label"má byť premenovaná na `service_imp` a namiesto nej má byť vytvorená funkcia `service` obsahujúca kód zabezpečujúci vzájomnú výlučnosť. V poslednom kroku je zmenený názov funkcie `service` zmenený na pôvodný názov funkcie označenej značkou "label".

7.5 Zhrnutie

Táto kapitola priniesla opis navrhovaného systému na vkladanie návrhových vzorov do existujúcich paralelných zdrojových kódov skladajúceho sa z troch častí. Analyzátora kódu, ktorý je zodpovedný za spracovanie zdrojového kódu a jeho reprezentáciu pomocou štruktúry Searchable Code Model. Druhým modulom je modul vyhodnocovania kvality kódu,

ktorý má na vstupe Searchable Code Model analyzovaného zdrojového kódu a množinu špecifikácii návrhových vzorov, zapísaných vo formáte XML, ktorý vychádza z metodiky popísanej v predošlej kapitole. Tento modul vyhodnotí predložený kód oproti definovaným návrhovým vzorom a navrhne používateľovi vhodné návrhové vzory na zvýšenie kvality spracovávaného zdrojového kódu. Posledným modulom systému je modul refaktoringu, ktorý na základe špecifikácií návrhových vzorov vykoná príslušný refaktoring zdrojového kódu. Takto navrhnutý systém bol využitý pri experimentálnom overení pomocou vybraných návrhových vzorov opísanom v nasledujúcej kapitole.

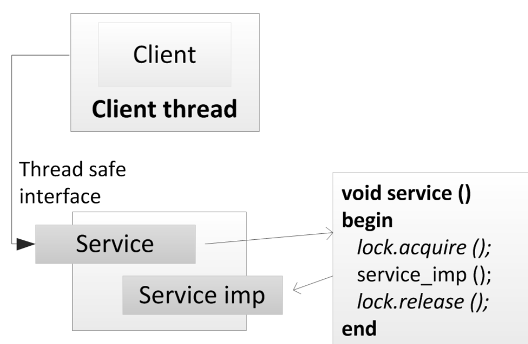
Kapitola 8

Experimentálne overenie pomocou vybraných návrhových vzorov

Táto kapitola opisuje experimentélne overenie navrhnutej metodiky pomocou vybranej sady návrhových vzorov. Návrhové vzory boli vybrané z katalógu POSA [23] a sú opísané v kapitole 3.3. Predmetom experimentu bol zápis návrhového vzoru pomocou navrhnutého spôsobu zápisu. V skoro všetkých prípadch bolo možné nájsť rozumné dopyty špecifikujúce miesto vloženia daného návrhového vzoru. Nasledujúce kapitoly prinášajú krátky popis každého vzoru, UML diagram a nájdené dopyty do navrhnutého Searchable Code Modelu, ktorý je výsledkom tejto práce. Dopyty sú zapísané pomocou Fluent API notácie Microsoft LINQ [1] a pomocou písaného textu. Všetky nižšie spomenuté návrhové vzory slúžia na predchádzanie problémom so synchronizáciou vlákien.

8.1 Thread safe interface

Ako je ukázané na obrázku 8.1 návrhový vzor Thread - Safe Interface rozdeľuje funkcie komponenty na verejne prístupné rozhranie a privátnu implementáciu. Verejne prístupné rozhranie získa zámok, zavolá príslušnú privátnu funkciu a potom uvoľní zámok.



Obrázok 8.1: Návrhový vzor Thread - Safe Interface.

Na zabezpečenie správnej synchronizácie, klientské vlákna volajú iba funkcie rozhrania. Funkcia rozhrania získa potrebný zámok a zavolá príslušnú implementačnú funkciu ktorá sa už nestará o zamykanie a môže slobodne volať iné implementačné funkcie. Uviaznutie (self-deadlock) nemôže nastať pretože zámok sa získava iba raz na začiatku v rozhraní a pri rekurzívnom volaní funkcií je zámok získavaný iba raz, čo taktiež zvyšuje výkon programu.

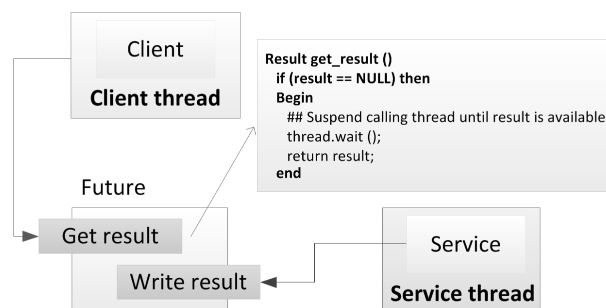
Z popisu vzoru dokážeme definovať množinu vyhľadávacích dopytov pre tento návrhový vzor takto: Existuje objekt s jednou alebo viacerými funkciami, ku ktorým pristupuje viac ako jedno vlákno a daná funkcia nieje chránená žiadnym zámkom. Na obrázku 8.2 môžeme vidieť zápis dopytu pomocou Microsoft LINQ notácie. Bolo by možné polemizovať o úplnosti zvoleného dopytu, pretože daný dopyt v podstate skúma iba existenciu nejakého zámku, ale keďže použité algoritmy analýzy kódu poskytujú jak informáciu o vláknach, tak informáciu o držaných zámkoch pre každý riadok programu, tak je možné vymyslieť aj iné dopyty ktoré by určovali miesto v zdrojovom kóde kam by bolo vhodné vložiť návrhový vzor Thread - Safe Interface.

```
CodeStats.Types
.SelectMany(type => type.Functions)
.Where(func => ((func.Threads.Count > 1)
&& (GetLocks(func).Count() == 0));
```

Obrázok 8.2: Kód dopytu pre návrhový vzor Thread - Safe Interface.

8.2 Future

Na obrázku 8.3 je možné vidieť návrhový vzor Future, ktorý okamžite po zavolaní funkcie vracia "virtuálny"dátový objekt nazvaný Future. Objekt Future, obsahuje informácie o stave výpočtu výsledku volanej funkcie, pričom výpočet výsledku je spustený v samostatnom vlákne. Objekt Future, vráti výsledok iba po dokončení algoritmu výpočtu.



Obrázok 8.3: Návrhový vzor Future.

Ak chce klientské vlákno prečítať hodnotu z Future objektu pred tým než je výsledok platný, tak Future objekt uspí klientske vlákno do doby než je platný výsledok zapísaný vo Future objekte. Future objekt taktiež môže obsahovať neblokujúcu funkciu slúžiacu na neblokujúce skontrolovanie platnosti uloženej hodnoty.

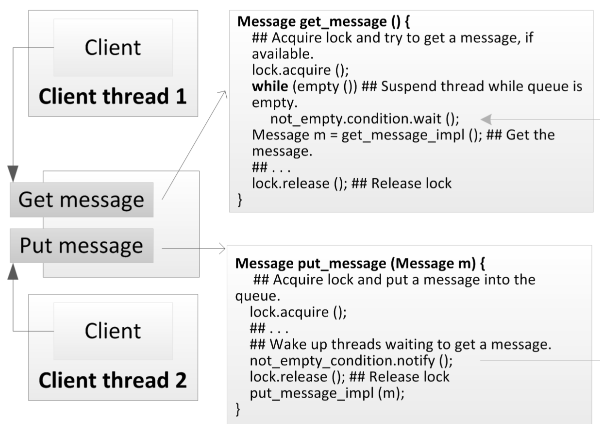
Z popisu vzoru dokážeme definovať dopyty pre návrhový vzor, takto: Existuje objekt X, ktorý obsahuje funkciu, ktorej počet volaných inštrukcií je väčší než definovaná medza. Alebo existuje objekt X, ktorého konštruktor volá viac inštrukcií, než je definovaná medza a prvé volanie funkcie nad týmto objektom vo volajúcej funkcií je vzdialené od volania konštruktora. Ako môžeme vidieť na ukážke kódu 8.4, dopyt v tomto prípade používa pomocnú funkciu AvgFunctionLen() ktorá vracia odhadnutý počet volaných inštrukcií počas výpočtu zvolenej funkcie alebo konštruktora. Kvôli bezpečnosti a konečnosti výpočtu, funkcia AvgFunctionLen vracia maximálne hodnotu Int32.MaxValue. Na obrázku 8.4 môžeme vidieť zápis dopytu pomocou Microsoft LINQ notácie.

```
CodeStats.Types
.SelectMany(type => type.Functions)
.Where(func =>
 AvgFunctionLen(func) >= treshold);
```

Obrázok 8.4: Kód dopytu pre návrhový vzor Future.

8.3 Guarded Suspension

Ako je ukázané na obrázku 8.5 návrhový vzorGuarded Suspension namiesto ukončenia blokovaného vlákna dané vlákno uspí, takže ostatné vlákna môžu pristupovať ku zdieľaným premenným a tak meniť hodnoty používané zablokovanými vláknami, čím umožňuje dodatočné odblokovanie blokovaných vlákien.



Obrázok 8.5: Návrhový vzor Guarded Suspension.

Hlavným prínosom návrhového vzoru Guarded Suspension je, že minimalizuje náklady spojené s paralelizáciou a taktiež zvyšuje dostupnosť zdieľaných komponentov. Ak je uspanie vlákien riešené na vrstve OS, tak cena daného uspania a príslušnej synchronizácie je minimálna.

Z popisu vzoru dokážeme definovať dopyty pre návrhový vzor, takto: Existuje funkcia ktorá je vykonávaná v samostatnom vlákne, táto funkcia obsahuje podmienku, ktorá ukončuje vlákno funkcie bez žiadneho ďalšieho výpočtu. Ako môžeme vidieť na detaile kódu 8.6,

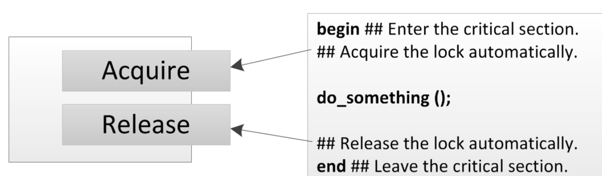
tak dopyt používa funkciu InstrCounter(), ktorá vytvára dvojice (instrID, inštrukcia),ktoré poskytujú informáciu o vzdialenosti každej inštrukcie od začiatku funkcie. Kvôli bezpečnosti a konečnosti výpočtu, funkcia InstrCounter vracia maximálne hodnotu Int32.MaxValue. Na obrázku 8.4 môžeme vidieť zápis dopytu pomocou Microsoft LINQ notácie.

```
CodeStats.Types
.SelectMany(type => type.Functions)
.Select(func => InstrCounter(func))
.Where(instr =>
 (instr.Id <= treshold) && (instr.IsReturn));
```

Obrázok 8.6: Kód dopytu pre návrhový vzor Guarded Suspension

8.4 Scoped Locking

Ako vidíme na obrázku 8.7, návrhový vzor Scoped Locking zaobaluje kritickú sekciu programu pomocou príkazu lock, ktorý automaticky získa zámok typu mutex na svojom začiatku a automaticky uvoľňuje držaný zámok pri každej ceste von z bloku kódu v tele príkazu lock.



Obrázok 8.7: Scoped Locking design pattern.

Návrhový vzor Scoped Locking zvyšuje robustnosť paralelného softvéru tým, že eliminuje časté programátorské chyby spojené so synchronizáciou viacerých vlákien. Zámky sú automaticky získavané keď vlákno vstúpi do kritickej sekcie, a automaticky uvoľňované keď z nej vystúpi. Implementácia tohto návrhového vzoru závisí od použitého programovacieho jazyka. Napríklad programovací jazyk Java obsahuje kľúčové slovo synchronized ktoré prikazuje kompilátoru automaticky vygenerovať príslušné inštrukcie obsluhujúce zamykanie a odomykanie zámkov.

Z popisu vzoru dokážeme definovať dopyty pre návrhový vzor, takto: majme objekt X, ku ktorého premenným pristupuje viac ako jedno vlákno, a prístup k týmto premenným,

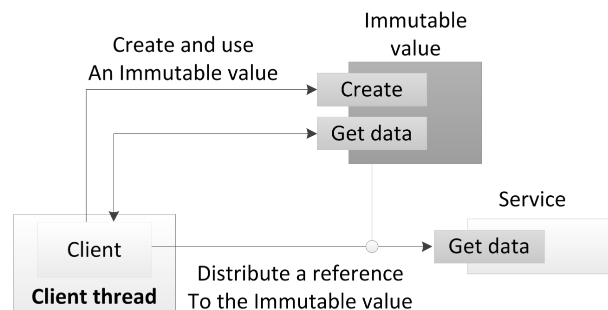
```
CodeStats.Types
.SelectMany(type => type.Variables)
.Where(v => v.Threads.Count() > 1)
.Where(v => GetLocks(v).Count() == 0);
```

Obrázok 8.8: Kód dopytu pre návrhový vzor Scoped Locking.

nieje chránený žiadnym zámkom. Na obrázku 8.8 môžeme vidieť zápis dopytu pomocou Microsoft LINQ notácie.

8.5 Immutable Value

Ako vidíme na obrázku 8.9, návrhový vzor Immutable Value definuje objekty, ktorých inštancie sú nemenné. Stav objektu a všetkých jeho premenných je nastavený v konštruktore a žiadne ďalšie zmeny nie sú dovolené. Takýto objekt dokáže rádovo zvýšiť rýchlosť spracovania dát, pretože programový kód vie, že dáta v danom objekte sú nemenné a tým pádom nemusí vykonávať toľko kontrol a validácií ako keby boli dáta objektu premenlivé.



Obrázok 8.9: Návrhový vzor Immutable Value.

V nemennom objekte sa nachádzajú iba read only parametre. Absencia ľubovoľnej možnosti zmeniť daný objekt odstraňuje nutnosť akejkoľvek synchronizácie a tým pádom zjednodušuje návrh systému a zefektívňuje prácu programu. Odstránením nutnosti kopírovania objektov, taktiež zlepšuje výkon programu.

Z popisu vzoru dokážeme definovať dopyty pre návrhový vzor, takto: všetky premenné ktoré sú zapísané iba raz a v ďalšom kóde programu sp už iba čítané. Na obrázku 8.10 môžeme vidieť zápis dopytu pomocou Microsoft LINQ notácie.

```
CodeStats.Types
.SelectMany(type => type.Variables)
.Where(v => GetWritesCount(v) == 1);
```

Obrázok 8.10: Kód dopytu pre návrhový vzor Immutable Value.

8.6 Vyhodnotenie experimentu a zhrnutie

Táto kapitola opisuje experimentélne overenie navrhnutej metodiky pomocou vybranej sady návrhových vzorov. Návrhové vzory boli vybrané z katalógu POSA [23] a sú detailne opísané v kapitole 3.3. Predmetom experimentu bol zápis návrhového vzoru pomocou navrhnutého spôsobu zápisu. V skoro všetkých prípadch bolo možné nájsť rozumné dopyty špecifikujúce miesto vloženia daného návrhového vzoru. Napríklad pri vzore Thread-Safe Interface

je dopyt špecifikujúci miesto v analyzovanom kóde pomerne jednoduchý a jasný, celkovo pri synchronizačných návrhových vzoroch bolo jednoduché nájsť vhodnú precondition. Na druhej strane, pri návrhových vzoroch pre súbežný beh sa ukázalo hľadanie vhodných preconditions ako nemožné a bez výsledku. Dôvodom môže byť pomerne veľká abstrakcia návrhových vzorov pre súbežný beh. To že nie je možné definovať vhodnú precondition neznamená, že daný návrhový vzor nieje opísateľný pomocou navrhovaného spôsobu zápisu. Pri daných návrhových vzoroch je možné popísať ich štruktúru a správanie, ale nateraz nie je možné využívať možnosť automatického navrhovania daných návrhových vzorov počas analýzy existujúcich paralelných zdrojových kódov. Zápis návrhových vzorov pre súbežný beh, vrátane špecifikácie precondition môže byť predmetom ďalšieho výskumu v budúcnosti.

Kapitola 9

Záver

Táto dizertačná práca predstavuje spôsob zápisu paralelných návrhových vzorov, ktorý umožňuje využitie tohoto zápisu pre automatické vkladanie návrhových vzorov do existujúcich zdrojových kódov. Existujú rôzne výskumy, ktoré sa venujú problematike automatického zavádzania návrhových vzorov do existujúcich zdrojových kódov, avšak žiaden z existujúcich výskumov sa nevenuje problematike zavádzania paralelných návrhových vzorov. Prvá časť práce prináša pohľad na použité témy analýzy kódu, návrhových vzorov a refaktoringu v kapitolách 2, 3, 4

Hlavným prínosom práce je návrh spôsobu zápisu návrhových vzorov umožňujúci asistované vkladanie paralelných návrhových vzorov do existujúcich paralelných zdrojových kódov, ktorý je vytvorený pomocou kombinácie existujúcich techník a metodík. Na to je potrebné byť schopný pracovať s návrhovým vzorom automaticky, čo znamená že návrhový vzor musí byť definovaný formálne. Druhá požiadavka je, že návrhový vzor musí obsahovať špecifikáciu miesta, kam sa má vložiť návrhový vzor. Tretia požiadavka je, že špecifikácia návrhového vzoru musí umožňovať jeho automatické vkladanie do existujúcich zdrojových kódov. Na základe týchto požiadaviek definujme návrhový vzor ako dvojicu (P, d) kde P je množina preconditions, určujúca vhodné umiestnenie vzoru a d je samotný popis vzoru. Množina P vychádza z analýzy kódu, ktorá umožňuje vytvárať dopyty nad existujúcim zdrojovým kódom, ktorých výsledok je miesto v zdrojovom kóde. Špecifikácia vzoru definuje jeho štruktúru a správanie. Navrhnutý spôsob zápisu je formálne popísaný v kapitole 6.

Na základe dát získaných z analýzy (ktorá je popísaná v kapitole 6.1), je možné definovať miesta v kóde s nesprávnou synchronizáciou medzi vláknami, prípadne s nesynchronizovaným prístupom ku zdielaným premenným. Tieto dva prípady sú najčastejšími dôvodmi na vloženie návrhového vzoru a preto sú popísané v tejto práci. Samozrejme existujú aj iné dôvody na vloženie paralelných návrhových vzorov do existujúcich zdrojových kódov, ale špecifikácia príslušných anti-vzorov sa dá získať jednoduchým rozšírením vyššie popísaného algoritmu. Špecfikáciou miesta v kóde na ktoré by sa mal aplikovať návrhový vzor je $p \in \mathbb{P}$ ktoré vieme jednoznačne definovať pomocou výrokov predikátovej logiky nad reláciami cg, pt, ta a la.

Špecifikácia vzoru využíva údaje zistené počas analýzy na definíciu miesta kam sa má vložiť vzor. Na to, aby bolo možné vložiť vzor je potrebný minimálne jeden bod v kóde.

Týmto bodom sa rozumie miesto s nevhodným kódom. K definícii štruktúry sú využité prvky predikátovej logiky 1. rádu, najmä premenné, logické spojky, existenčný kvantifikátor a predikáty. Premenné reprezentujú primárne entity, zatiaľ čo predikáty reprezentujú trvalé relácie medzi nimi. U niektorých návrhových vzorov nestačí samotná definícia ich štruktúry, ale tiež je veľmi dôležitá definícia ich správania, tj. popis, ako účastníci vzoru spolu spolupracujú. Použitý jazyk opisuje správanie vzoru pomocou podmnožiny temporálnej logiky akcie. Jazyk opisuje správanie vzoru ako nekonečnú sekvenciu stavov, ktorými entity vzoru prechádzajú.

Spôsob zápisu paralelných návrhových vzorov bol použitý na návrh systému na vkladanie návrhových vzorov do existujúcich paralelných zdrojových kódov ktorý je popísaný v kapitole 7. Navrhnutý systém sa skladá z troch častí. Prvou časťou je analyzátor existujúcich kódov, ktorý pripraví informácie o zdrojových kódoch, vrátane informácie o vláknach a zámkoch. Druhou časťou je modul vyhodnocovania kvality analyzovaných kódov, ktorý dokáže navrhnúť vhodné paralelné návrhové vzory. Treťou časťou je modul refaktoringu, ktorý sa postará o zavedenie návrhového vzoru do existujúcich kódov bez zmeny ich funkčnosti.

Navrhnutý systém bol experimentálne overený na vybranej množine paralelných návrhových vzorov zo zbierky POSA [23], ktoré sa zaoberajú súbežným behom a synchronizáciou súbežne bežiacich vlákien v programe. Vykonaný experiment vrátane výsledkov je popísaný v kapitole 8. Navrhnutý systém je možné zlepšovať, napríklad podrobnejším štúdiom a implementáciou rôznych metód refaktoringu, alebo použitím iných metód analýzy zdrojových kódov. Taktiež je možné preštudovať možné rozšírenia navrhnutého spôsobu zápisu paralelných návrhových vzorov.

Využitie navrhnutého spôsobu zápisu je v oblasti bezpečnostných štandardov, najmä v oblasti letectva, zdravotníckej a vojenskej techniky. Navrhnutý spôsob zápisu je možné použiť aj k zvyšovaniu kvality existujúcich zdrojových kódov, keďže zdrojové kódy založené na návrhových vzoroch sú prehľadnejšie a jednoduchšie spravovateľné.

Tématika spoľahlivosti a bezpečnosti je hlavou témou autorových publikácií a ostatných vedeckých výstupov. Opisovaný spôsob zápisu paralelných návrhových vzorov a návrh systému, ktorý ho využíva, boli prezentované v publikáciach na konferenciách a vedeckých časopisoch: [Pub1, Pub2, Pub3, Pub4]. Využitie paralelných algoritmov na útoky silou bolo prezentované v časopise [Pub7]. Téma spoľahlivosti je spoločnou témou aj dvoch zverejnených úžitkových vzorov [Pat1, Pat2].

Vybrené publikácie autora

Publikace

Pub1 JURNEČKA Peter, HANÁČEK Petr a KAČIC Matej. Code Search API, Base of Parallel Code Refactoring System for Safety Standards Compliance. In: Journal of Cyber Security and Mobility 2014, s. 47-63 ISSN: 2245-1439, DOI: 10.13052/jcsm2245-1439.313

Pub2 HANÁČEK Petr, JURNEČKA Peter a KAČIC Matej. Concept of parallel code generating and refactoring system for safety standards compliance. In: Proceedings of the 2013 IEEE 7th International Conference on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS). Berlin: Institute of Electrical and Electronics Engineers, 2013, s. 630-635. ISBN 978-1-4799-1426-5.

Pub3 JURNEČKA Peter, HANÁČEK Petr, BARABAS Maroš, HENZL Martin a KAČIC Matej. A method for parallel software refactoring for safety standards compliance. In: System Safety 2013 collection of papers. Cardiff: The Institution of Engineering and Technology, 2013, s. 1-6. ISBN 978-1-84919-777-9. ISSN 0537-9989.

Pub4 JURNEČKA Peter, HANÁČEK Petr, BARABAS Maroš, HENZL Martin a KAČIC Matej. A method for parallel software refactoring for safety standards compliance. Resilience, Security & Risk in Transport. London: The Institution of Engineering and Technology, 2013, s. 42-48. ISBN 978-1-84919-787-8.

Pub5 KAČIC Matej, HENZL Martin, JURNEČKA Peter a HANÁČEK Petr. Malware injection in wireless networks. In: Proceedings of the 2013 IEEE 7th International Conference on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS). Berlin: Institute of Electrical and Electronics Engineers, 2013, s. 483-487. ISBN 978-1-4799-1426-5.

Pub6 HENZL Martin, HANÁČEK Petr, JURNEČKA Peter a KAČIC Matej. A Concept of Automated Vulnerability Search in Contactless Communication Applications. In: Proceedings 46th Annual IEEE International Carnahan Conference on Security Technology. Boston: Institute of Electrical and Electronics Engineers, 2012, s. 180-186. ISBN 978-1-4673-4807-2.

Pub7 HANÁČEK Petr a JURNEČKA Peter. Využitie grafických kariet na útoky silou. DSM Data Security Management. 2011, roč. 15, č. 2, s. 10-13. ISSN 1211-8737.

Pub8 JURNEČKA Peter a KAJAN Rudolf. Automatic generation of adaptive, educational and multimedia computer games. Signal, Image and Video Processing. London: Springer London, 2008, roč. 2, č. 4, s. 371-384. ISSN 1863-1703.

Pub9 KAJAN Rudolf a JURNEČKA Peter. Learning with smart multipurpose interactive learning environment. In: Learning to Live in the Knowledge Society. Boston, 2008, s. 101-104. ISBN 978-0-387-09728-2.

Pub10 JURNEČKA Peter a KAJAN Rudolf. Adaptive Educational Gameplay within Smart Multipurpose Interactive Learning Environment. In: Proceedings of the SMAP 2007 - 2nd International Workshop on Semantic Media Adaptation and Personalization. London, 2007, s. 165-170. ISBN 0-7695-3040-0.

Úžitkové vzory / Patenty

Pat1 JURNEČKA, P., A. JURNEČKA a J. BUDAY: Zariadenie na zvýšenie spoľahlivosti (MTBF) elektrotechnických výrobkov. 2011. Slovensko. 126-2010, MPT:G01R 13/28. Přihlášeno 07.09.2010. Zapsáno 29.09.2011.

Pat2 JURNEČKA, P., J. BUDAY, R. HAVRILA, A. CHLAPÍKOVÁ a M. PČOLA: Zariadenie na meranie a vyhodnocovanie obsahu vyšších harmonických v napätiach, resp. prúdoch elektrických zariadení. Slovensko. 169-2010, MPT:G01R 23/16. Přihlášeno 15.11.2010. Zapsáno 21.10.2011

Literatura

[1] Microsoft LINQ.
URL https://msdn.microsoft.com/en-us/library/mt693024.aspx

[2] ASP.NET MVC Overview. 2016.
URL https://msdn.microsoft.com/en-us/library/dd381412(v=vs.108).aspx

[3] checkstyle project page. 2016.
URL http://checkstyle.sourceforge.net/

[4] Jlint project page. 2016.
URL https://sourceforge.net/projects/jlint/

[5] pmd project page. 2016.
URL https://pmd.github.io/

[6] Administration, F. A.: Advisory Circular 20-115B. http://goo.gl/C6d1k, 1993 [cit. 2014-08-23].

[7] Administration, F. D.: Guidance for the Content of Premarket Submissions for Software Contained in Medical Devices. http://goo.gl/JqkYr, 2005-05-11 [cit. 2014-08-23].

[8] Al-Ahmad, W.: Object-Oriented Design Patterns for Detailed Design. *Journal of Object Technology*, ročník 5, č. 2, Březen 2006: s. 155–169, ISSN 1660-1769, doi:10.5381/jot.2006.5.2.a3.
URL http://www.jot.fm/contents/issue_2006_03/article3.html

[9] Alexander, C.: *The Timeless Way of Building*. číslo zv. 8 in Center for Environmental Structure Berkeley, Calif: Center for Environmental Structure series, Oxford University Press, 1979, ISBN 9780195024029.
URL https://books.google.cz/books?id=H6CE9hlbO8sC

[10] Alexander, C.; Ishikawa, S.; Silverstein, M.: *A Pattern Language: Towns, Buildings, Construction*. Center for Environmental Structure Berkeley, Calif: Center for Environmental Structure series, OUP USA, 1977, ISBN 9780195019193.
URL https://books.google.cz/books?id=hwAHmktpk5IC

[11] Allen, F. E.: Control Flow Analysis. In *Proceedings of a Symposium on Compiler Optimization*, New York, NY, USA: ACM, 1970, s. 1–19, doi:10.1145/800028.808479.
URL http://doi.acm.org/10.1145/800028.808479

[12] Alur, D.; Crupi, J.; Malks, D.: *Core J2EE Patterns: Best Practices and Design Strategies*. Core Series, Prentice Hall PTR, 2003, ISBN 9780131422469.
URL https://books.google.cz/books?id=1dx34EMVyi8C

[13] Bajracharya, S.; Ngo, T.; Linstead, E.; aj.: Sourcerer: A Search Engine for Open Source Code Supporting Structure-based Search. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, New York, NY, USA: ACM, 2006, ISBN 1-59593-491-X, s. 681–682, doi:10.1145/1176617.1176671.
URL http://doi.acm.org/10.1145/1176617.1176671

[14] Ball, T.: The Concept of Dynamic Analysis. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-7, London, UK, UK: Springer-Verlag, 1999, ISBN 3-540-66538-2, s. 216–234.
URL http://dl.acm.org/citation.cfm?id=318773.318944

[15] Bandi, R. K.; Vaishnavi, V. K.; Turk, D. E.: Predicting Maintenance Performance Using Object-Oriented Design Complexity Metrics. *IEEE Trans. Softw. Eng.*, ročník 29, č. 1, Leden 2003: s. 77–87, ISSN 0098-5589, doi:10.1109/TSE.2003.1166590.
URL http://dx.doi.org/10.1109/TSE.2003.1166590

[16] Bayley, I.: Formalising Design Patterns in Predicate Logic. In *Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods*, SEFM '07, Washington, DC, USA: IEEE Computer Society, 2007, ISBN 0-7695-2884-8, s. 25–36, doi:10.1109/SEFM.2007.22.
URL http://dx.doi.org/10.1109/SEFM.2007.22

[17] Bayley, I.: Formalising Design Patterns in Predicate Logic. In *Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods*, SEFM '07, Washington, DC, USA: IEEE Computer Society, 2007, ISBN 0-7695-2884-8, s. 25–36, doi:10.1109/SEFM.2007.22.
URL http://dx.doi.org/10.1109/SEFM.2007.22

[18] Bayley, I.; Zhu, H.: On the Composition of Design Patterns. In *Proceedings of the 2008 The Eighth International Conference on Quality Software*, QSIC '08, Washington, DC, USA: IEEE Computer Society, 2008, ISBN 978-0-7695-3312-4, s. 27–36, doi:10.1109/QSIC.2008.32.
URL http://dx.doi.org/10.1109/QSIC.2008.32

[19] Bayley, I.; Zhu, H.: Specifying Behavioural Features of Design Patterns in First Order Logic. In *Proceedings of the 2008 32Nd Annual IEEE International Computer Software and Applications Conference*, COMPSAC '08, Washington, DC, USA: IEEE Computer Society, 2008, ISBN 978-0-7695-3262-2, s. 203–210, doi:10.1109/COMPSAC.2008.67.
URL http://dx.doi.org/10.1109/COMPSAC.2008.67

[20] Bergstein, P. L.: Object-preserving Class Transformations. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, New York, NY, USA: ACM, 1991, ISBN 0-201-55417-8, s. 299–313, doi:10.1145/117954.117977.
URL http://doi.acm.org/10.1145/117954.117977

[21] Bessey, A.; Block, K.; Chelf, B.; aj.: A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM*, ročník 53, č. 2, Únor 2010: s. 66–75, ISSN 0001-0782, doi:10.1145/1646353.1646374.
URL http://doi.acm.org/10.1145/1646353.1646374

[22] Briand, L. C.; Wüst, J.; Daly, J. W.; aj.: Exploring the Relationship Between Design Measures and Software Quality in Object-oriented Systems. *J. Syst. Softw.*, ročník 51, č. 3, Květen 2000: s. 245–273, ISSN 0164-1212, doi:10.1016/S0164-1212(99)00102-8.
URL http://dx.doi.org/10.1016/S0164-1212(99)00102-8

[23] Buschmann, F.; Henney, K.; Schmidt, D.: *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*. John Wiley & Sons, 2007, ISBN 0470059028, 9780470059029.

[24] C., H.: DO-178B safety certification and other software security tools drive avionics software designs. http://goo.gl/Z8zniy, 2011-05-19 [cit. 2014-08-23].

[25] Casais, E.: An Incremental Class Reorganization Approach. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '92, London, UK, UK: Springer-Verlag, 1992, ISBN 3-540-55668-0, s. 114–132.
URL http://dl.acm.org/citation.cfm?id=646150.679212

[26] Casais, E.: The Automatic Reorganization of Object Oriented Hierarchies - A Case Study. 1994.

[27] Chatzigeorgiou, A.: Mathematical Assessment of Object-Oriented Design Quality. *IEEE Trans. Softw. Eng.*, ročník 29, č. 11, Listopad 2003: s. 1050–1053, ISSN 0098-5589, doi:10.1109/TSE.2003.1245306.
URL http://dx.doi.org/10.1109/TSE.2003.1245306

[28] Chen, Y.-F. R.; Gansner, E. R.; Koutsofios, E.: A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. In *Proceedings of the 6th European SOFTWARE ENGINEERING Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC '97/FSE-5, New York, NY, USA: Springer-Verlag New York, Inc., 1997, ISBN 3-540-63531-9, s. 414–431, doi:10.1145/267895.267924.
URL http://dx.doi.org/10.1145/267895.267924

[29] Chidamber, S. R.; Kemerer, C. F.: A Metrics Suite for Object Oriented Design. *IEEE Trans. Softw. Eng.*, ročník 20, č. 6, Červen 1994: s. 476–493, ISSN 0098-5589, doi:10.1109/32.295895.
URL http://dx.doi.org/10.1109/32.295895

[30] CUNNINGHAM, W.: Portland pattern repository.
URL http://c2.com/ppr/index.html

[31] Didier, J.-Y.; Mallem, M.: A New Approach to Detect Potential Race Conditions in Component-based Systems. In *Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '14, New York, NY, USA: ACM, 2014, ISBN 978-1-4503-2577-6, s. 97–106, doi:10.1145/2602458.2602470.
URL http://doi.acm.org/10.1145/2602458.2602470

[32] Dietrich, J.; Elgar, C.: A Formal Description of Design Patterns Using OWL. In *Proceedings of the 2005 Australian Conference on Software Engineering*, ASWEC '05, Washington, DC, USA: IEEE Computer Society, 2005, ISBN 0-7695-2257-2, s. 243–250, doi:10.1109/ASWEC.2005.6.
URL http://dx.doi.org/10.1109/ASWEC.2005.6

[33] Ducasse, S.; Rieger, M.; Demeyer, S.: A Language Independent Approach for Detecting Duplicated Code. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, Washington, DC, USA: IEEE Computer Society, 1999, ISBN 0-7695-0016-1, s. 109–.
URL http://dl.acm.org/citation.cfm?id=519621.853389

[34] Ducasse, S.; Rieger, M.; Golomingi, G.; aj.: Tool Support for Refactoring Duplicated OO Code. In *In Object-Oriented Technology (ECOOP'99 Workshop Reader), number 1743 in LNCS (Lecture Notes in Computer Science*, Springer-Verlag, 1999, s. 2–6.

[35] D'ADDERIO, L.; DEWAR, R.: Systems reengineering patterns.
URL http://homepages.inf.ed.ac.uk/perdita/Reengineering/

[36] Espinoza, F. A. C.; Esquer, G. N.; Cansino, J. S.: *Automatic Design Patterns Identification of C++ Programs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, ISBN 978-3-540-36087-2, s. 816–823, doi:10.1007/3-540-36087-5_94.
URL http://dx.doi.org/10.1007/3-540-36087-5_94

[37] Farias, C. M. D.; Li, W.; Delicato, F. C.; aj.: A Systematic Review of Shared Sensor Networks. *ACM Comput. Surv.*, ročník 48, č. 4, Únor 2016: s. 51:1–51:50, ISSN 0360-0300, doi:10.1145/2851510.
URL http://doi.acm.org/10.1145/2851510

[38] Fleury, E.; Point, G.; Vincent, A.: Binary Program Analysis: Theory and Practice.
URL http://www-verimag.imag.fr/async/CCIS/talk13/Fleury.pdf

[39] Food; Administration, D.: General Principles of Software Validation; Final Guidance for Industry and FDA Staff. http://goo.gl/HjIKb, 2002-01-11 [cit. 2014-08-23].

[40] Fowler, M.: Refactoring. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*, TOOLS '00, Washington, DC, USA: IEEE Computer Society, 2000, ISBN 0-7695-0774-3, s. 437–.
URL http://dl.acm.org/citation.cfm?id=832261.833315

[41] Fowler, M.; Beck, K.: *Refactoring: Improving the Design of Existing Code*. Component software series, Addison-Wesley, 1999, ISBN 9780201485677.
URL https://books.google.cz/books?id=1MsETFPD3I0C

[42] France, R. B.; Kim, D.-K.; Ghosh, S.; aj.: A UML-Based Pattern Specification Technique. *IEEE Trans. Softw. Eng.*, ročník 30, č. 3, Březen 2004: s. 193–206, ISSN 0098-5589, doi:10.1109/TSE.2004.1271174.
URL http://dx.doi.org/10.1109/TSE.2004.1271174

[43] Frey, F. J.; Hentrich, C.; Zdun, U.: Pattern-based Process for a Legacy to SOA Modernization Roadmap. In *Proceedings of the 19th European Conference on Pattern Languages of Programs*, EuroPLoP '14, New York, NY, USA: ACM, 2014, ISBN

978-1-4503-3416-7, s. 10:1–10:21, doi:10.1145/2721956.2721969.
URL http://doi.acm.org/10.1145/2721956.2721969

[44] Gamma, E.; Helm, R.; Johnson, R.; aj.: *Design Patterns: Elements of Reusable Object-Oriented Software (Adobe Reader)*. Pearson Education, 1994, ISBN 9780321700698.
URL https://books.google.cz/books?id=6oHuKQe3TjQC

[45] Gosain, A.; Sharma, G.: *Static Analysis: A Survey of Techniques and Tools*. New Delhi: Springer India, 2015, ISBN 978-81-322-2268-2, s. 581–591, doi:10.1007/978-81-322-2268-2_59.
URL http://dx.doi.org/10.1007/978-81-322-2268-2_59

[46] Halstead, M. H.: *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977, ISBN 0444002057.

[47] Hericko, M.; Beloglavec, S.: A Composite Design-Pattern Identification Technique. *Informatica (Slovenia)*, ročník 29, č. 4, 2005: s. 469–476.
URL http://dblp.uni-trier.de/db/journals/informaticaSI/informaticaSI29.html#HerickoB05

[48] Hovemeyer, D.; Pugh, W.: Finding Concurrency Bugs in Java. In *In Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*, 2004.

[49] Johnson, R. E.; Foote, B.: Designing Reusable Classes. *Journal of Object-Oriented Programming*, ročník 1, č. 2, June/July 1988: s. 22–35.
URL http://www.laputan.org/drc.html

[50] Johnson, R. E.; Opdyke, W. F.: Refactoring and Aggregation. In *Proceedings of the First JSSST International Symposium on Object Technologies for Advanced Software*, London, UK, UK: Springer-Verlag, 1993, ISBN 3-540-57342-9, s. 264–278.
URL http://dl.acm.org/citation.cfm?id=646897.709889

[51] Joshi, S.; Shyamasundar, R. K.; Aggarwal, S. K.: A New Method of MHP Analysis for Languages with Dynamic Barriers. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, May 2012, s. 519–528, doi:10.1109/IPDPSW.2012.70.

[52] Khedker, U.; Sanyal, A.; Karkare, B.: *Data Flow Analysis: Theory and Practice*. Boca Raton, FL, USA: CRC Press, Inc., první vydání, 2009, ISBN 0849328802, 9780849328800.

[53] Lazzarini Lemos, O. A.; Bajracharya, S.; Ossher, J.; aj.: Applying Test-driven Code Search to the Reuse of Auxiliary Functionality. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, New York, NY, USA: ACM, 2009, ISBN 978-1-60558-166-8, s. 476–482, doi:10.1145/1529282.1529384.
URL http://doi.acm.org/10.1145/1529282.1529384

[54] Lieberherr, K. J.; Bergstein, P.; Silva-Lepe, I.: From Objects to Classes: Algorithms for Optimal Objection-oriented Design. *Softw. Eng. J.*, ročník 6, č. 4, Červenec 1991: s. 205–228, ISSN 0268-6961, doi:10.1049/sej.1991.0024.
URL http://dx.doi.org/10.1049/sej.1991.0024

[55] Maruyama, K.; Shima, K.-i.: Automatic Method Refactoring Using Weighted Dependence Graphs. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, New York, NY, USA: ACM, 1999, ISBN 1-58113-074-0, s. 236–245, doi:10.1145/302405.302627.
URL http://doi.acm.org/10.1145/302405.302627

[56] McCabe, T. J.: A Complexity Measure. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, s. 407–.
URL http://dl.acm.org/citation.cfm?id=800253.807712

[57] McConnell, S.: *Code Complete*. DV-Professional Series, Microsoft Press, 2004, ISBN 9780735619678.
URL https://books.google.cz/books?id=QnghAQAAIAAJ

[58] MIRA Ltd: MISRA-C:2004 Guidelines for the use of the C language in Critical Systems. Říjen 2004.
URL www.misra.org.uk

[59] Moore, I.: Guru-a tool for automatic restructuring of self inheritance hierarchies. Citeseer.

[60] Moore, I.: Automatic Inheritance Hierarchy Restructuring and Method Refactoring. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, New York, NY, USA: ACM, 1996, ISBN 0-89791-788-X, s. 235–250, doi:10.1145/236337.236361.
URL http://doi.acm.org/10.1145/236337.236361

[61] Muraki, T.; Saeki, M.: Metrics for Applying GOF Design Patterns in Refactoring Processes. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE '01, New York, NY, USA: ACM, 2001, ISBN 1-58113-508-4, s. 27–36, doi:10.1145/602461.602466.
URL http://doi.acm.org/10.1145/602461.602466

[62] Nielson, F.; Nielson, H. R.; Hankin, C.: *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999, ISBN 3540654100.

[63] Opdyke, W. F.: *Refactoring Object-oriented Frameworks*. Dizertační práce, Champaign, IL, USA, 1992, uMI Order No. GAX93-05645.

[64] Opdyke, W. F.: Refactoring Object-oriented Frameworks. Technická zpráva, Champaign, IL, USA, 1992.

[65] Ossher, J.; Bajracharya, S.; Linstead, E.; aj.: SourcererDB: An Aggregated Repository of Statically Analyzed and Cross-linked Open Source Java Projects. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, Washington, DC, USA: IEEE Computer Society, 2009, ISBN 978-1-4244-3493-0, s. 183–186, doi:10.1109/MSR.2009.5069501.
URL http://dx.doi.org/10.1109/MSR.2009.5069501

[66] Otto, F.; Moschny, T.: Finding Synchronization Defects in Java Programs: Extended Static Analyses and Code Patterns. In *Proceedings of the 1st International Workshop*

on Multicore Software Engineering, IWMSE '08, New York, NY, USA: ACM, 2008, ISBN 978-1-60558-031-9, s. 41–46, doi:10.1145/1370082.1370093.
URL http://doi.acm.org/10.1145/1370082.1370093

[67] Pun, W.; Winder, R.: Automating Class Hierarchy Graph Construction.

[68] Reiss, S. P.: Semantics-based Code Search. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, Washington, DC, USA: IEEE Computer Society, 2009, ISBN 978-1-4244-3453-4, s. 243–253, doi:10.1109/ICSE.2009.5070525.
URL http://dx.doi.org/10.1109/ICSE.2009.5070525

[69] Reissing, R.: Towards a Model for Object-Oriented Design Measurement. In *ECOOP'01 Workshop QAOOSE*, 2001.

[70] Ricci, A.: Programming with Event Loops and Control Loops - From Actors to Agents. *Comput. Lang. Syst. Struct.*, ročník 45, č. C, Duben 2016: s. 80–104, ISSN 1477-8424, doi:10.1016/j.cl.2015.12.003.
URL http://dx.doi.org/10.1016/j.cl.2015.12.003

[71] Schmidt, D. C.; Stal, M.; Rohnert, H.; aj.: *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. New York, NY, USA: John Wiley & Sons, Inc., druhé vydání, 2000, ISBN 0471606952, 9780471606956.

[72] Snelting, G.; Tip, F.: Reengineering Class Hierarchies Using Concept Analysis. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '98/FSE-6, New York, NY, USA: ACM, 1998, ISBN 1-58113-108-9, s. 99–110, doi:10.1145/288195.288273.
URL http://doi.acm.org/10.1145/288195.288273

[73] Stamelos, I.; Angelis, L.; Oikonomou, A.; aj.: Code Quality Analysis in Open-Source Software Development,. *Information Systems Journal*, ročník 12, č. 1, 2002: s. 43–60.

[74] Subramanyam, R.; Krishnan, M.: Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. *IEEE Computer Society Trans. Software Engineering*, ročník 29, č. 4, 2003: s. 297–310, ISSN 0098-5589, doi:http://doi.ieeecomputersociety.org/10.1109/TSE.2003.1191795.

[75] Sun, Y.; Gray, J.; White, J.: A Demonstration-based Model Transformation Approach to Automate Model Scalability. *Softw. Syst. Model.*, ročník 14, č. 3, Červenec 2015: s. 1245–1271, ISSN 1619-1366, doi:10.1007/s10270-013-0374-0.
URL http://dx.doi.org/10.1007/s10270-013-0374-0

[76] Sutter, H.; Alexandrescu, A.: *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. C++ In-Depth Series, Pearson Education, 2004, ISBN 9780132654425.
URL https://books.google.cz/books?id=mmjVIC6WolgC

[77] Sweeney, P. F.; Tip, F.: A Study of Dead Data Members in C++ Applications. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, New York, NY, USA: ACM, 1998, ISBN 0-89791-987-4, s. 324–332, doi:10.1145/277650.277750.
URL http://doi.acm.org/10.1145/277650.277750

[78] Taibi, T.; Ngo, D. C. L.: Formal Specification of Design Patterns - A Balanced Approach. *Journal of Object Technology*, ročník 2, č. 4, Červenec 2003: s. 127–140, ISSN 1660-1769, doi:10.5381/jot.2003.2.4.a4.
URL http://www.jot.fm/contents/issue_2003_07/article4.html

[79] Walter L. Hürsch; Seiter, L. M.: Automating the Evolution of Object-Oriented Systems. In *Proc. ISOTAS*, 1996, s. 2–21.

[80] Watson, G. M.; Hessinger, D. A.: Software Complexity. *Crosstalk, Journal of Defense Software Engineering*, 1994: s. 5–9.

[81] Xenos, M.; Stavrinoudis, D.; Zikouli, K.; aj.: Object-oriented metrics - a survey. In *Proceedings of the FESMA Conference (FESMA'2000)*, 2000.

[82] Zeller, A.: Program Analysis: A Hierarchy.
URL
http://www.eecs.yorku.ca/coursearchive/2004-05/F/6431/ZellerErnst.pdf

[83] Zhu, H.; Bayley, I.; Shan, L.; aj.: Tool Support for Design Pattern Recognition at Model Level. In *Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference - Volume 01*, COMPSAC '09, Washington, DC, USA: IEEE Computer Society, 2009, ISBN 978-0-7695-3726-9, s. 228–233, doi:10.1109/COMPSAC.2009.37.
URL http://dx.doi.org/10.1109/COMPSAC.2009.37