CONSTRUCTE STATICE: EVOLUȚIE ȘI IMPACT ASUPRA ASPECTELOR CALITATIVE ALE SISTEMELOR SOFTWARE

Teză de doctorat – Rezumat

pentru obținerea titlului științific de doctor la
Universitatea Politehnica Timișoara
în domeniul de doctorat Calculatoare și Tehnologia Informației

autor ing. Cosmin MARȘAVINA
conducător științific Prof.univ.dr.ing. Mihai V. MICEA

luna Ianuarie anul 2022

Prezenta teză de doctorat cuprinde 7 capitole, 173 de pagini, 56 de tabele, 79 de figuri și diagrame, și 102 titluri bibliografice. Scopul acesteia este de a studia 1) modul în care sunt utilizate constructele statice în cadrul proiectelor software complexe, 2) cum au evoluat acestea pe durata dezvoltării unui sistem, și 3) gradul în care ele afectează o serie de aspecte calitative ale sistemelor software.

Capitolul 1 face o introducere a conceptelor care vor fi prezentate în cadrul tezei. Mai întâi este discutată problema care urmează să fie adresată. Apoi sunt expuse întrebările de cercetare care au fost formulate și se explică importanța fiecăreia. De asemenea, este prezentată relevanța acestui studiu și sunt menționate principalele contribuții aduse. Obiectivele stabilite și modul în care plănuim să le realizăm sunt și ele discutate. Ultima secțiune a acestui capitol descrie structura tezei și conținutul fiecărui capitol.

Se începe prin a explica importanța testării; aceasta reprezintă o parte vitală a ciclului de viață al dezvoltării sistemelor software. Studiile au arătat că mai mult de jumătate din efortul necesar pentru implementarea proiectelor software complexe este dedicat testării [1]. În cadrul acestei teze ne vom concentra pe testarea unitară într-un context orientat pe obiecte deoarece acest tip de teste au o legătură directă cu anumite părți ale codului sursă. Există 3 aspecte calitative ale sistemelor software care sunt strâns legate de procesul de testare: testabilitatea, susceptibilitatea la modificări și susceptibilitatea la erori. Începem prin a le defini; testabilitatea este „capacitatea produsului software de a putea fi validat atunci când suferă modificări” [2]. Susceptibilitatea la modificări / erori sunt caracteristici ale artefactelor software care reprezintă cât de probabil este ca acestea să trebuiască să fie modificate / reparate.

Carențele de proiectare sunt încălcări ale practicilor și principiilor de design care fac sistemele software mai greu de înțeles, întreținut și evoluat. Pentru câteva dintre ele s-a dovedit deja că au un impact negativ asupra anumitor aspecte calitative. Totuși, există încă multe alte carențe de proiectare / aspecte calitative care nu au fost investigate până în acest moment. Constructele statice sunt o categorie de entități din codul sursă în cadrul cărora este folosit cuvântul cheie *static*. Pentru unele dintre ele am reușit deja să demonstrăm că au un efect negativ asupra testabilității [3] sau a susceptibilității la erori [4]. În această teză vrem să investigăm fiecare tip de construct static atât din punct de vedere al prezenței / utilizării, cât și al impactului pe care acestea îl au asupra celor 3 aspecte calitative prezentate anterior. Pentru a studia prezența / utilizarea acestora nu ne vom rezuma doar la cea mai recentă versiune a unui sistem; întregul istoric al proiectului va fi luat în considerare.

Pentru a investiga toate aceste aspecte, au fost formulate următoarele întrebări de cercetare:

- Sunt constructele statice folosite în cadrul sistemelor software complexe? Prin această întrebare încercăm să stabilim: 1) dacă instanțe ale constructelor statice apar într-adevăr în codul de producție; 2) cum sunt acestea utilizate de alte clase; 3) dacă clienții lor sunt localizați într-un număr relativ redus de pachete sau împrăștiați prin tot codul.
- Cum au evoluat constructele statice de-a lungul dezvoltării unui proiect? Dorim să observăm 1) dacă instanțe de anumite tipuri sunt create în continuare deși au fost dovedite ca fiind problematice și 2) dacă numărul de clase client crește odată cu dimensiunea sistemului.
- Au constructele statice un impact negativ asupra aspectelor calitative ale sistemelor software? Vrem să studiem efectele folosirii constructelor statice asupra celor 3 aspecte calitative de interes: 1) testabilitate, 2) susceptibilitate la modificări și 3) susceptibilitate la erori.

Efectuând cercetările necesare pentru a putea răspunde la întrebările de mai sus, ne așteptăm să aducem următoarele contribuții:

- o metodologie pentru a studia diferite tipuri de carențe de proiectare, evoluția lor și impactul pe care îl au asupra calității sistemelor software;
- un model care permite evaluarea testabilității unei clase pe baza cantității și calității testelor unitare corespunzătoare acesteia;
- un proces pentru 1) a determina modificările granulare care au loc asupra codului sursă în cadrul unui commit și 2) a stabili dacă în commit-ul respectiv au fost reparate erori;
- un tool care incorporează toate aceste aspecte;
- un studiu empiric cu ajutorul căruia răspundem la întrebările de cercetare propuse.

Pentru a reuși să aducem contribuțiile prezentate anterior, ne-am setat mai multe obiective ce trebuie îndeplinite. Principalele obiective ar fi:

1. studierea stadiului actual al literaturii de specialitate pentru temele de interes, și anume: detecția carențelor de proiectare (cu accent pe constructele statice) și evoluția acestora, modele pentru cuantificarea aspectelor calitative ale sistemelor software, și carențe de proiectare care au un impact negativ asupra aspectelor investigate;
2. categorisirea constructelor statice și definirea unor strategii de detecție prin care putem identifica instanțe din fiecare tip. În plus, analizarea prezenței și utilizării acestor instanțe atât pentru cea mai recentă versiune a unui proiect cât și pentru o serie de versiuni premergătoare;
3. dezvoltarea unor proceduri prin care aspectele calitative investigate (testabilitate, susceptibilitate la modificări și susceptibilitate la erori) pot fi evaluate. Stabilirea dacă constructele statice din fiecare categorie au un efect negativ asupra acestora.

Ultima secțiune a acestui capitol explică modul în care a fost structurată teză. Pentru capitolele rămase sunt discutate pe scurt împărțirea pe secțiuni și conținutul fiecărei secțiuni. De asemenea, la finalul fiecărui capitol există un fragment în care sunt sintetizate toate lucrurile discutate în capitolul respectiv.

Capitolul 2 prezintă stadiul actual al literaturii de specialitate cu privire la detecția și evoluția carențelor de proiectare, modele pentru evaluarea aspectelor calitative ale sistemelor software, și carențe care afectează aceste aspecte. În prima secțiune sunt discutate diferitele strategii de detecție propuse, precum și tool-uri dezvoltate în acest scop. Se insistă asupra strategiilor bazate pe metrici, ca și cele propuse în [5], deoarece vom folosi o abordare similară pentru a identifica diferitele tipuri de constructe statice. O altă lucrare de interes este [6], unde pe lângă strategii de detecție sunt menționate și tehnici de reparare cu ajutorul cărora se pot înlătura carențele de proiectare găsite. În a doua parte a aceste secțiuni sunt prezentate o serie

de smell-uri ce pot apărea în clasele de test; acestea reprezintă deviații de la directivele propuse pentru a ajuta dezvoltatorii să creeze suite de teste adecvate. Ca și în cazul carențelor de proiectare, au fost identificate diferite categorii de test smell-uri și s-au propus strategii pentru detecția lor (de exemplu, în [7]). Prezența acestor probleme în codul de testare a fost corelată cu existența carențelor de proiectare în clasele de producție [8]; vom avea în vedere acest aspect atunci când vom evalua calitatea testării efectuate asupra unei clase.

A doua secțiune privește modul în care au evoluat anumite carențe de proiectare. Există un număr considerabil de articole care investighează aspectul acesta (cum ar fi [9]), dar niciunul dintre ele nu analizează vreun tip de constructe statice. Vom studia evoluția acestor constructe într-un mod similar celui prezentat în [10], unde am cercetat co-evoluția dintre codul de producție și cel de test.

În următoarea secțiune sunt descrise modelele propuse pentru a evalua aspectele calitative de interes. Majoritatea modelelor pentru testabilitate (de exemplu, cel expus în [11]) pornesc de la design-ul claselor / întregului sistem, nu de la codul sursă. Într-un articol care compară diferite modele pentru evaluarea acestui aspect calitativ s-a conchis 1) că niciun model nu este superior și 2) că modelul folosit ar trebui selectat pe baza particularităților analizei care urmează să fie întreprinsă [12]. Pentru susceptibilitate la modificări / erori, majoritatea publicațiilor încearcă să prezică dacă o anumită clasă urmează să fie modificată / reparată în viitorul apropiat. Există foarte puține articole care să propună modele pentru cuantificarea acestor 2 aspecte calitative pe baza istoriei unui sistem. Un astfel de exemplu ar fi [13], unde clasele Java sunt categorisite ca fiind cu / fără erori folosind 2 seturi de metrici (legate de produs / de proces).

Impactul pe care anumite carențe de proiectare îl au asupra celor 3 aspecte calitative menționate anterior este discutat în ultima secțiune a acestui capitol. Nu am găsit niciun articol care să investigheze acest lucru pentru diferite tipuri de constructe statice. Una dintre categoriile studiate în [14] este starea globală (mai exact atribute statice non-finale și singletoni); se concluzionează că aceasta poate avea un efect asupra testabilității sistemelor software. Într-o altă lucrare [15], autorul propune un tool care poate fi folosit pentru a investiga acest aspect; tool-ul nu pleacă însă de la codul sursă Java, ci de la bytecode.

În **Capitolul 3** se discută abordarea adoptată în vederea: 1) categorisirii și detecției constructelor statice, 2) studierii evoluției acestora, 3) cuantificării celor 3 aspecte calitative, și 4) implementării întregului proces de colectare a datelor. Luând în considerare granularitatea variată a constructelor care folosesc cuvântul cheie *static*, am decis să realizăm o categorisire pe mai multe niveluri. La nivel de clasă distingem 3 tipuri de constructe statice: singletoni (atât cu stare cât și fără), clase utilitare și restul claselor care conțin doar instanțe statice mai mici. Pentru cele din urmă categorisirea s-a făcut în funcție de tipul instanței prezente, și anume: metode statice care accesează / modifică starea clasei din care fac parte, metode statice care folosesc doar parametrii, atribute statice non-finale, constante, și blocuri de inițializare statice. Pentru fiecare dintre categoriile menționate anterior au fost propuse strategii de detecție cu ajutorul cărora să poată fi identificate instanțele respective. De exemplu, 3 condiții trebuie să fie îndeplinite pentru ca o clasă să fie categorisită ca singleton:

1. nu există niciun constructor public în clasă;
2. clasa are un atribut privat static („instanța singleton”) și o metodă accesor declarată public static care instanțiază în mod „leneș” acel atribut și apoi îl returnează;
3. metoda menționată anterior reprezintă singura cale prin care atributul respectiv poate fi accesat.

Această strategie de detecție corespunde formei generale a tiparului; ea a fost extinsă pentru a putea detecta o serie de variații ale acestuia (cele discutate în [16]). De asemenea,

singletonii au fost împărțiți în 2 categorii, cu și fără stare.

A doua secțiune a acestui capitol explică procesul prin care este studiată evoluția constructelor statice. Am utilizat sistemul Git pentru a obține datele necesare efectuării acestei analize. Pentru fiecare sistem analizat, commit-urile sunt eșantionate cu o frecvență de 1 commit pe lună. Apoi iterăm peste commit-urilor rămase și determinăm diferențele dintre fiecare commit și cel corespunzător lunii precedente; următoarele diferențe sunt consemnate: numărul total de instanțe per categorie, numărul de clase client pentru fiecare instanță și numărul mediu de clienți pentru întregul proiect. Date suplimentare privind fiecare construct static / toți clienții acestuia din commit-ul respectiv sunt de asemenea înregistrate, alături de alte informații utile (cum ar fi faptul că o clasă a fost marcată cu adnotarea @Deprecated).

Următoarea secțiune descrie modelul dezvoltat pentru cuantificarea testabilității unei clase. Spre deosebire de modelele propuse până în momentul de față, acesta evaluează testabilitatea unei clase de producție pe baza cantității și calității testelor unitare corespunzătoare acesteia. Așadar, acest aspect este estimat atât dintr-o perspectivă cantitativă cât și dintr-una calitativă. Metricile folosite pentru măsurarea cantității sunt 1) acoperirea la nivel de linie și 2) procentul de metode de producție testate. Datele referitoare la acoperire sunt obținute cu ajutorul lui *JaCoCo* [17], un tool care poate fi utilizat pe orice tip de proiect (Maven, Gradle, etc.) și oferă un raport detaliat care include de asemenea o serie de măsurători privind complexitatea claselor / metodelor. Pentru a evalua calitatea testării efectuate asupra unei clase luăm în considerare anumite smell-uri ce ar putea să apară în clasa de test corespunzătoare. Test smell-urile sunt identificate folosind tsDetect [18]; acest tool a fost extins pentru a putea specifica care dintre cele 19 smell-uri detectabile sunt prezente într-un anumit test unitar. Sunt calculate din nou 2 metrici: 1) procentul de teste unitare în care apar smell-uri și 2) numărul de tipuri diferite de smell-uri care există într-o clasă de test. Pe baza acestor 4 metrici calculăm 2 scoruri (cantitativ și calitativ) care sunt agregate apoi pentru a obține scorul general de testabilitate. Acest scor poate fi folosit pentru a compara o clasă de producție cu alta (care este similară cu ea din punct de vedere al dimensiunii și complexității); dacă prima clasă are un scor general mai mare, atunci înseamnă că aceasta este mai ușor de testat. Pentru a determina cele 2 scoruri menționate anterior, sunt utilizate o serie de intervale corespunzătoare valorilor de prag (explicate pe larg în secțiunea respectivă).

Pentru a identifica clasele susceptibile la modificări / erori folosim procedurile descrise în penultima secțiune a acestui capitol. Singura diferență dintre cele 2 este că în cadrul procedurii corespunzătoare susceptibilității la erori sunt luate în considerare doar commit-urile care au fost categorisite ca fiind reparatoare de defecte. Pentru această categorisire sunt utilizate 2 tipuri de informații: 1) cele disponibile în mesajul commit-ului și 2) date suplimentare colectate de pe instanța Jira asociată proiectului. Procedura propusă în acest sens este explicată în detaliu în cadrul tezei; cu ajutorul acesteia putem determina cu o acuratețe ridicată commit-urile în care sunt reparate erori. Pentru a evalua cele 2 aspecte calitative trebuie să putem stabili cu exactitate ce modificări au fost făcute în cadrul unui commit. În acest scop extragem modificările granulare care au loc în codul sursă folosind *ChangeDistiller* [19]; acestea specifică: entitatea care a fost modificată (clasă, metodă sau atribut), tipul acelei modificări (de exemplu, modificarea unei instrucțiuni condiționale într-o metodă), precum și alte detalii referitoare la aceasta (cum ar fi severitatea). Întregul istoric al modificărilor pentru o anumită clasă este analizat și comparat cu istoricele claselor similare. Dacă acea clasă 1) a fost modificată mai des sau 2) mai multe modificări au fost efectuate asupra acesteia, atunci ea poate fi considerată ca având o susceptibilitate mai mare la modificări. Așa cum am menționat anterior, susceptibilitatea la erori este evaluată pe baza acelorași metrici, doar că atunci când acestea sunt calculate se iau în considerare doar commit-urile reparatoare de defecte.

În ultima secțiune este prezentat întregul proces de colectare a datelor, alături de tool-ul

care a fost dezvoltat în acest sens. Acest tool se numește *DFAnalizer* și este implementat ca o extensie a lui *Patrools* [20], un plugin de Eclipse care era deja capabil să calculeze câteva dintre metricile necesare. Am proiectat tool-ul astfel încât să aibă o structură modulară; mai multe module pot fi combinate pentru ca acesta să efectueze analiza dorită. Unul dintre module conține strategiile de detecție pentru carența de proiectare studiată (de exemplu, singleton). Un alt modul este responsabil de cuantificarea aspectului calitativ investigat. Dacă vrem să analizăm și evoluția carenței respective, atunci trebuie doar să adăugăm modulul corespunzător. Modulele sunt configurabile și pot fi extinse cu ușurință; de exemplu, am putea studia o altă carență de proiectare creând un modul nou cu strategiile de detecție aferente.

Capitolul 4 explică modul în care a fost realizat studiul empiric. Acesta începe prin a prezenta scopul principal al studiului, precum și ipotezele care au fost formulate. După sunt descrise variabilele independente și dependente corespunzătoarei fiecărei ipoteze împreuna cu procedurile prin care acestea pot fi măsurate. Criteriile pe baza cărora au fost selectate proiectele incluse în studiu sunt, de asemenea, discutate. Ultima secțiune a acestui capitol prezintă analizele care au fost efectuate ca parte a studiului empiric.

Așa cum a fost explicat in primul capitol, scopul principal al acestei tezei este de a obține o mai bună înțelegere a 1) constructelor statice, 2) evoluției acestora și 3) aspectelor calitative asupra cărora ele au un impact negativ. Pentru a ne atinge scopul, am formulat 3 întrebări de cercetare corespunzătoare fiecăruia dintre aceste aspecte. Obiectivul principal al studiului empiric este de a obține răspunsuri la cele 3 întrebări. De aceea am analizat în mod izolat aspectele. Pentru fiecare dintre întrebările de cercetare am formulat 2 ipoteze corespunzătoare variantelor nulă și alternativă. Ca și exemplu, pentru prima întrebare varianta nulă ar fi „constructele statice apar rareori în sistemele software complexe”, iar cea alternativă este „constructele statice sunt prezente în codul de producție și există alte clase care le utilizează”.

Fiecare dintre ipoteze este discutată în detaliu și sunt prezentate variabilele independente și dependente. Pentru perechea de ipoteze corespunzătoare primei întrebări variabilele independente sunt caracteristicile specifice ale sistemului studiat, iar cele dependente sunt diferitele tipuri de constructe statice care apar / sunt utilizate în cadrul acestuia. Pe lângă caracteristicile generale, cum ar fi dimensiunea sau complexitatea, mai sunt investigate o serie de alte caracteristici (de exemplu, funcționalități cheie sau faptul că proiectul respectiv e structurat ca o librărie).

Atunci când am ales sistemele incluse în studiu empiric am avut în vedere mai multe criterii, inclusiv pe cele discutate în [21]. Pentru a fi selectate, proiectele trebuiau să fie:

1. relevante din punct de vedere al dimensiunii și complexității;
2. disponibile pe Git și având un număr considerabil de versiuni;
3. acoperite corespunzător prin testare unitară;
4. asociate cu instanțe de Jira care conțin toate problemele întâmpinate pe parcursul dezvoltării lor.

Pe baza acestor criterii, am selectat 11 sisteme open-source care să fie incluse în cadrul studiului. Am încercat să alegem proiecte care diferă din punct de vedere 1) al dimensiunii și complexității, 2) al practicilor de dezvoltare, sau 3) a efortului depus pentru testarea lor, dar care să respecte totuși criteriile. O analiză preliminară a acestor aspecte este realizată în cadrul aceleiași secțiuni.

În ultima secțiune sunt discutate analizele efectuate asupra sistemelor prezentate anterior, și anume:

- o analiză preliminară cu privire la dimensiunea și structura sistemelor, istoricul lor, și efortul depus pentru a le testa;
- o analiză a prezenței / utilizării diferitelor tipuri de constructe statice în cea mai recentă

versiune a fiecărui proiect;

- o analiză a evoluției fiecărei categorii de constructe statice de-a lungul dezvoltării sistemelor;
- 3 analize privind impactul constructelor statice asupra aspectelor calitative studiate (testabilitate, susceptibilitate la modificări și susceptibilitate la erori).

În **Capitolul 5** sunt prezentate rezultatele obținute pentru analizele enumerate anterior. Acest capitol include doar rezultate brute, interpretarea lor are loc în capitolul următor. În prima secțiune sunt analizate constructele statice identificate în cea mai recentă versiune a fiecăruia dintre cele 11 proiecte studiate. Pe lângă numărul de instanțe, am mai vrut să investigăm modul în care acestea sunt folosite și să observăm dacă clienții lor sunt localizați într-un număr limitat de pachete sau împrăștiați prin întreg sistemul. Pentru fiecare categorie de constructe statice am comparat clienții și localizarea acestora cu aspectele similare pentru celelalte entități de același tip; de exemplu, metodele statice care accesează atributele claselor în care sunt declarate / care utilizează doar parametrii primiți sunt comparate cu metodele non-statice din cadrul sistemului cu privire la aceste 2 aspecte. Pentru fiecare dintre cele 11 proiecte s-a realizat câte un tabel care conține toate datele menționate anterior.

A doua secțiune a acestui capitol analizează evoluția diferitelor tipuri de constructe statice. Ca și în analiza precedentă, nu ne-am concentrat doar pe numărul de instanțe de un anumit tip prezente într-o versiune a sistemului; am încercat să înțelegem motivele pentru care unele instanțe (sau clienții lor) sunt adăugate / șterse. În plus, pentru constructele la nivel de clasă (singletoni și clase utilitare) am investigat și cum au fost folosite de-a lungul dezvoltării unui proiect. Pentru fiecare tip de construct static am creat un grafic care ilustrează numărul total de instanțe / procentul de clase de producție care utilizează instanțe de acel tip (axa Y) în timp (axa X). Atunci când apar situații neașteptate, cum ar fi o scădere semnificativă a numărului de instanțe de interes sau o clasă care își pierde majoritatea clienților, am căutat motivele din spatele acestor decizii.

În următoarea secțiune se analizează impactul constructelor statice asupra testabilității claselor. Ne bazăm pe scorul de testabilitate pentru a compara clasele care conțin constructe statice cu alte clase care sunt similare cu ele în ceea ce privește dimensiunea și complexitatea. Așa cum am explicat în Capitolul 3, această comparație este făcută atât dintr-o perspectivă cantitativă, cât și dintr-una calitativă. Cantitatea este evaluată pe baza 1) acoperirii la nivel de linie și 2) a procentului de metode testate dintr-o anumită clasă. Pentru calitate sunt determinate 1) procentul de teste unitare în care există smell-uri și 2) numărul de tipuri diferite de smell-uri dintr-o clasă de test. Pe baza acestor metrici calculăm 2 scoruri (unul cantitativ și altul calitativ) care sunt agregate într-un scorul general de testabilitate. Pentru fiecare sistem am realizat un tabel care conține cele 3 scoruri pentru clasele cu diferite tipuri de constructe statice / clasele similare acestora.

Ultima secțiune a acestui capitol prezintă rezultatele corespunzătoare impactului acestora asupra susceptibilității la modificări / erori. În ambele evaluări se compară numărul mediu de modificări care au avut loc asupra claselor care conțin un anumit tip de construct static / clasele care au fost categorisite ca fiind similare cu acestea. De asemenea, a fost calculat numărul mediu de modificări per commit pentru a determina dacă clasele cu instanțe de interes au fost modificate în mai multe commit-uri decât clasele similare cu ele. În plus, am vrut să observăm ce tipuri de modificări granulare au loc cel mai frecvent (top 5 tipuri de modificări) și să stabilim dacă clasamentele sunt diferite pentru clasele cu diverse categorii de constructe statice / clasele similare acestora. Pentru fiecare sistem am creat 2 tabele, unul corespunzător susceptibilității la modificări și altul pentru susceptibilitatea la erori. Așa cum am explicat în Capitolul 3, pentru cel de-al doilea aspect luăm în considerare doar commit-urile care au fost categorisite ca fiind reparatoare de erori.

Capitolul 6 discută: 1) interpretarea rezultatelor în contextul întrebărilor de cercetare care au fost formulate și 2) factorii care ar putea fi considerați amenințări la adresa validității studiului empiric și a rezultatelor obținute. Începem prin a privi rezultatele în ansamblu, reușind astfel să tragem niște concluzii pertinente. Pentru prima întrebare de cercetare am observat că: 1) constructele statice sunt prezente în toate cele 11 sisteme analizate; 2) constantele sunt de departe cel mai utilizat tip de constructe statice, urmate de metodele statice (de ambele tipuri) și de clasele utilitare; atributele statice non-finale, singletonii și blocurile de inițializare statice apar mai rar, în special în proiectele de dimensiuni mai mici; 3) numărul de clienți al constructelor statice și localizarea lor nu sunt foarte diferite în comparație cu cele ale constructelor similare (în cele mai multe dintre cazuri).

Legat de a doua întrebare de cercetare, principalele observații ar fi: 1) că există câteva categorii de constructe statice (cum sunt atributele statice non-finale sau singletonii) pentru care apar din ce în ce mai puține instanțe în versiunile recente (comparativ cu cele de la începutul / mijlocul procesului de dezvoltare) și 2) că am identificat multe cazuri în care aceste instanțe își pierd majoritatea clienților (sau chiar toți) înainte de a fi la rândul lor șterse.

Pentru ultima întrebare de cercetare am stabilit că: 1) atributele statice non-finale, singletonii cu stare și metodele statice care accesează starea au cel mai mare impact negativ asupra testabilității; 2) toate categoriile de constructe statice cu excepția constantelor și a metodelor statice care folosesc doar parametrii afectează susceptibilitatea la modificări, deși pentru singletonii fără stare și clasele utilitare rezultatele sunt contradictorii pentru tipuri diferite de sisteme; 3) în cazul susceptibilității la erori, impactul constructelor statice nu este la fel de semnificativ ca și pentru susceptibilitatea la modificări.

În cea de-a doua secțiune a acestui capitol sunt prezentate cele 3 categorii de factori care amenință validitatea studiului empiric și a rezultatelor; de asemenea, sunt discutate modurile în care am încercat să atenuăm efectele acestora. Factorii sunt grupați în 3 categorii (pe baza categorisirii propuse în [21]): de construcție, interni și externi. Cei din prima categorie pot apărea din cauza unor probleme în codul care a fost dezvoltat în vederea colectării datelor necesare pentru analizele efectuate. Pentru a evita astfel de probleme, abordarea propusă a fost testată cu atenție folosind mai multe sisteme de mici dimensiuni create special în acest scop. De exemplu, pentru strategiile de detecție definite am adăugat instanțe din fiecare categorie în diferite combinații pentru a ne asigura că aceste sunt identificate corect. De asemenea, am verificat manual toate datele obținute; din câte știm, acestea sunt corecte și complete. Amenințările din a doua categorie, cele interne, apar atunci când modificări ale variabilelor dependente nu pot fi corelate cu modificări în cele dependente. Principalele amenințări din această categorie sunt factorii perturbatori, mai exact alte variabile care pot masca o asociere existentă sau sugera în mod fals o asociere aparentă între variabilele independente și cele dependente. Sunt foarte greu de identificat toți factorii care au un impact asupra acestor variabile, dar am încercat să discutăm cât mai mulți posibil. Ca și exemplu, pentru prima ipoteză ar mai putea fi și alte caracteristici ale unui sistem care afectează prezența / utilizarea diferitelor tipuri de constructe statice.

Am identificat o serie de amenințări externe care au legătură cu rezultatele studiului empiric, mai exact că acestea nu sunt generalizabile în alte contexte. O primă limitare a acestui studiu este că toate cele 11 sisteme studiate sunt open-source. Sperăm să obținem acces la cel puțin 2 proiecte comerciale în viitorul apropiat, putând astfel înlătura această amenințare. O altă limitare ar fi că toate sistemele sunt dezvoltate în Java urmând o paradigmă orientată pe obiecte. Plănuim să reimplementăm tool-ul astfel încât să putem analiza proiecte realizate în alte limbaje de programare (de exemplu, C# și C++) sau după alte paradigme (cum ar fi programarea funcțională). Un alt factor de luat în considerarea este granularitatea ridicată la care sunt efectuate analizele. Studiul ar putea fi mai detaliat dacă am analiza totul la nivel de metodă;

aceasta reprezintă una dintre direcțiile asupra cărora ne vom concentra eforturile de acum încolo.

Capitolul 7 conține concluziile și direcțiile viitoare de cercetare. În prima secțiune sunt reiterate contribuțiile aduse, și anume:

1) metodologia pentru studierea evoluției și impactului asupra calității sistemelor software a oricărei carențe de proiectare;

2) modelul pentru cuantificarea testabilității claselor;

3) procesul pentru identificarea commit-urilor în care sunt rezolvate erori și determinarea modificărilor granulare care au loc între anumite commit-uri;

4) tool-ul pentru investigarea aspectelor de interes;

5) studiul empiric prin care sunt obținute răspunsuri la întrebările de cercetare pentru diferite tipuri de constructe statice.

Per total, principalul scop al tezei a fost atins; am reușit să obținem o mai bună înțelegere a prezenței / utilizării constructelor statice, a modului în care acestea au evoluat și a impactului pe care ele îl au asupra celor 3 aspecte calitative studiate (testabilitate, susceptibilitate la modificări și susceptibilitate la erori).

Analizând toate aspectele prezentate anterior, am reușit să tragem mai multe concluzii relevante. Principalele concluzii ar fi:

- că constructele statice 1) sunt prezente într-un număr foarte mare în codul de producție și 2) sunt utilizate frecvent de alte clase;
- că acestea încep să fie folosite din ce în ce mai puțin odată ce un proiect a ajuns la maturitate (comparativ cu versiunile din stadiul incipient al dezvoltării acestuia);
- că anumite tipuri de constructe statice, cum ar fi instanțe ale stării globale mutabile (singletoni cu stare și atribute statice non-finale) sau metode statice care accesează / modifică starea clasei din care fac parte, au un impact negativ asupra celor 3 aspecte calitative investigate.

În penultima secțiune am reflectat asupra a ceea ce am realizat și am explicat ce am fi putut face mai bine. Am încheiat capitolul cu câteva direcții promițătoare pe care le luăm în calcul în acest moment:

- îmbunătățirea studiului empiric prin adăugarea unor noi sisteme (comerciale și implementate în alte limbaje de programare / urmând alte paradigme / după alte metodologii de dezvoltare);
- investigarea altor carențe de proiectare, cum ar fi instanțieri de obiecte în constructori / metode sau încălcări ale Legii lui Demeter;
- perfecționarea modelelor cu ajutorul cărora se cuantifică cele 3 aspecte cantitative (de exemplu, adăugarea mai multor metrici la scorul de testabilitate);
- studierea tuturor aspectelor menționate anterior la un nivel mai scăzut de granularitate;
- propunerea unor tehnici de reparare pentru părțile problematice atât ale codului de producție cât și ale celui de test.

Referințe bibliografice

[1] Brooks Jr, Frederick P. The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, 2/E. Pearson Education India, 1995.

[2] IEEE Standards Coordinating Committee. "IEEE Standard Glossary of Software

Engineering Terminology (IEEE Std 610.12-1990). Los Alamitos." CA: IEEE Computer Society 169 (1990).

[3] Marsavina, Cosmin. "Studying the Evolution of Static Methods and their Effect on Class Testability." *2020 IEEE 20th International Symposium on Computational Intelligence and Informatics (CINTI)*. IEEE, 2020.

[4] Marsavina, Cosmin. "Understanding the Impact of Mutable Global State on the Defect Proneness of Object-Oriented Systems." *2020 IEEE 14th International Symposium on Applied Computational Intelligence and Informatics (SACI)*. IEEE, 2020.

[5] Marinescu, Radu. "Detection strategies: Metrics-based rules for detecting design flaws." Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on. IEEE, 2004.

[6] Kessentini, Marouane, et al. "Design defects detection and correction by example." Program Comprehension (ICPC), 2011 IEEE 19th International Conference on. IEEE, 2011.

[7] Van Rompaey, Bart, et al. "On the detection of test smells: A metrics-based approach for general fixture and eager test." *IEEE Transactions on Software Engineering* 33.12 (2007): 800-817.

[8] Tahir, Amjed. *A Study on Software Testability and the Quality of Testing In Object-Oriented Systems*. Diss. University of Otago, 2016.

[9] S. Vaucher, F. Khomh, N. Moha, and Y. G. Guéhéneuc, "Tracking design smells: Lessons from a study of god classes." *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009.

[10] Marsavina, Cosmin, Daniele Romano, and Andy Zaidman. "Studying fine-grained co-evolution patterns of production and test code." *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014.

[11] Mouchawrab, Samar, Lionel C. Briand, and Yvan Labiche. "A measurement framework for object-oriented software testability." Information and software technology 47.15 (2005): 979-997.

[12] Nikfard, Pourya, et al. "An Empirical Analysis of a Testability Model." Informatics and Creative Multimedia (ICICM), 2013 International Conference on. IEEE, 2013.

[13] Moser, Raimund, Witold Pedrycz, and Giancarlo Succi. "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction." Proceedings of the 30th international conference on Software engineering. 2008.

[14] Wolter, Jonathan, Russ Ruffer, and Miško Hevery. "Guide: Writing testable code." (2009): 1-38.

[15] Hevery, Misko. "Testability explorer: using byte-code analysis to engineer lasting social changes in an organization's software development process." *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. ACM, 2008.

[16] K. Stencel and P. Węgrzynowicz, "Implementation variants of the singleton design pattern." *OTM Confederated International Conferences* "*On the Move to Meaningful Internet Systems*." Springer, Berlin, Heidelberg, 2008.

[17] M. R. Hoffmann, B. Janiczak, and E. Mandrikov, "Eclemma-jacoco java code coverage library." (2011).

[18] A. Peruma, et al. "tsDetect: an open source test smells detection tool." *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020.

[19] H. C. Gall, B. Fluri, and M. Pinzger, "Change analysis with evolizer and changedistiller."

IEEE software 26.1 (2009): 26-33.

[20] P. F. Mihancea, “Patrools: Visualizing the Polymorphic Usage of Class Hierarchies.” *2010 IEEE 18th International Conference on Program Comprehension*. IEEE, 2010.

[21] L. S. Pinto, S. Sinha, and A. Orso, “Understanding myths and realities of test-suite evolution,” in Symposium on the Foundations of Software Engineering (FSE). ACM, 2012, p. 33.