УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

Унапређење рефакторисања у динамички типизираним објектно-оријентисаним језицима

Докторска дисертација

Improving Refactorings in Dynamically Typed Object-Oriented Languages

Doctoral Dissertation

Ментори:
проф. др Гордана Милосављевић
др Stéphane Ducasse

Кандидат:
Балша Шаренац

Нови Сад, 2026. године

УНИВЕРЗИТЕТ У НОВОМ САДУ **ОБРАЗАЦ – 5а**
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА[1]

Врста рада:	Докторска дисертација
Име и презиме аутора:	Балша Шаренац
Ментор (титула, име, презиме, звање, институција):	др Гордана Милосављевић, редовни професор, Факултет техничких наука, Универзитет у Новом Саду др Stéphane Ducasse, научни саветник, Инриа, Лил, Француска
Наслов рада:	Унапређење рефакторисања у динамички типизираним објектно-оријентисаним језицима
Језик и писмо рада:	Енглески језик, латиница
Физички опис рада:	Унети број: Страница: 121 Поглавља: 8 Референци: 104 Табела: 26 Слика: 9 Графикона: 0 Прилога: 4
Научна област:	Електротехничко и рачунарско инжењерство
Ужа научна област (научна дисциплина):	Примењене рачунарске науке и информатика
Кључне речи / предметна одредница:	Рефакторисање, системи за рефакторисање, трансформације програма, динамички типизирани језици, објектно-оријентисани језици, предуслови, метаморфно тестирање, Pharo

[1] Аутор докторске дисертације потписао је и приложио следеће обрасце:
5б – Изјава о ауторству;
5в – Изјава о истоветности штампане и електронске верзије докторског рада и дозвола за објављивање личних података;
5г – Изјава о коришћењу.
Ове Изјаве се чувају у институцији у штампаном и електронском облику и не кориче се са радом.

Резиме:	Системи за рефакторисање аутоматизују структурне измјене изворног кода са циљем очувања споља видљивог понашања. Програмерима су, међутим, потребне и провјерене структурне измјене које не гарантују очување понашања, на примјер при миграцији програмског интерфејса. Постојећи системи често нуде или рефакторисање или операцију нижег нивоа која провјеру примјенљивости препушта позиваоцу. У динамички типизираним језицима синтаксно исправан резултат и даље може да промијени везивање имена, одабир методе при позиву или ток управљања. Дисертација представља референтну архитектуру која раздваја структурну примјенљивост, провјере очувања понашања и политику интеракције. Трансформације носе структурну измјену и предуслове примјенљивости, рефакторисања их декоришу предусловима очувања понашања, а драјвери измјештају прикупљање параметара и одлучивање о упозорењима изван језгра операције. Архитектура је реализована миграцијом система за рефакторисање у окружењу *Pharo*. Удио дуплираних линија у анализираним пакетима смањен је са 6,02% на 1,52%. Доказ концепта на двије репрезентативне операције, у систему *rope* за језик *Python*, показује да се раздвајање, декорација и композиција могу пренијети и на независно развијен систем. Дисертација представља и Refazz, метаморфни метод тестирања система за рефакторисање. За довршене примјене рефакторисања Refazz провјерава да ли одабрани тестови пројекта и даље успјешно пролазе. Четири режима извршавања обухватају исправан улаз и испитивање простора параметара под уобичајеним провјерама, принудно извршавање послије одбијања предуслова примјенљивости и настављено извршавање послије упозорења о очувању понашања. У 255.699 покушаја над седам рефакторисања и пет пројеката Refazz је потврдио 19 дефеката, 12 у трансформацијама и 7 у предусловима, а шест исправки интегрисано је у *Pharo*.
Датум прихватања теме од стране надлежног већа:	20.11.2025.

Датум одбране: (Попуњава накнадно институција)	
Чланови комисије: (титула, име, презиме, звање, институција)	**Председник:** др Мирослав Зарић, редовни професор, Факултет техничких наука, Универзитет у Новом Саду **Члан:** др Милена Фртунић Глигоријевић, доцент, Електронски факултет, Универзитет у Нишу **Члан:** др Гордана Ракић, доцент, Природно-математички факултет, Универзитет у Новом Саду **Члан:** др Игор Дејановић, редовни професор, Факултет техничких наука, Универзитет у Новом Саду **Ментор:** др Stéphane Ducasse, научни саветник, Инриа, Лил, Француска **Ментор:** др Гордана Милосављевић, редовни професор, Факултет техничких наука, Универзитет у Новом Саду
Напомена:	

UNIVERSITY OF NOVI SAD
FACULTY OF TECHNICAL SCIENCES

KEY WORD DOCUMENTATION[2]

Document type:	Doctoral dissertation
Author:	Balša Šarenac
Supervisor (title, first name, last name, position, institution):	Gordana Milosavljević, PhD, full professor, Faculty of Technical Sciences, University of Novi Sad Stéphane Ducasse, PhD, research director, Inria, Lille, France
Thesis title in English:	Improving Refactorings in Dynamically Typed Object-Oriented Languages
Language and script:	English, Latin script
Physical description:	Number of: Pages: 121 Chapters: 8 References: 104 Tables: 26 Illustrations: 9 Graphs: 0 Appendices: 4
Scientific field:	Electrical and Computer Engineering
Scientific subfield (scientific discipline):	Applied Computer Science and Informatics
Subject, Key words:	Refactoring, refactoring engines, program transformations, dynamically typed languages, object-oriented languages, preconditions, metamorphic testing, Pharo

[2]The author of the doctoral dissertation has signed the following Statements:
5б – Statement on the authorship,
5в – Statement that the printed and e-version of the doctoral dissertation are identical and authorization to use personal data,
5г – Copyright statement.
The paper and e-versions of Statements are held at the institution and are not included into the printed thesis.

Abstract:	Refactoring engines automate structural changes to source code while aiming to preserve observable behavior. Developers also need checked structural changes that do not guarantee behavior preservation, for example when migrating an API. Existing engines often offer either a complete refactoring or a lower-level rewrite that leaves applicability checks to the caller. In dynamically typed languages, a syntactically valid result can still change name binding, dispatch, or control flow. This thesis presents a reference architecture that separates structural applicability, behavior-preserving checks, and interaction policy. Transformations carry the structural change and its applicability preconditions, refactorings decorate them with behavior-preserving preconditions, and interaction drivers keep parameter collection and warning decisions outside the operation core. The architecture was realized by migrating Pharo's refactoring engine. Duplicated lines in the analyzed packages decreased from 6.02% to 1.52%. A proof of concept on two representative operations, carried out in *rope*, a Python refactoring engine, shows that separation, decoration, and composition can transfer to an independently developed engine. The thesis also presents Refazz, a metamorphic testing method for refactoring engines. For completed refactoring applications, Refazz tests whether selected project tests remain passing. Its four execution modes cover valid input and parameter exploration under normal checks, forced execution after an applicability rejection, and continued execution after a behavior-preserving warning. Across 255,699 attempts involving seven refactorings and five projects, Refazz confirmed 19 defects, 12 in transformations and 7 in preconditions, and six fixes were integrated into Pharo.
Date of endorsement by the scientific board:	20.11.2025.
Date of defence: (Filled in by the institution)	

Thesis defence board: (title, first name, last name, position, institution)	**Chair:** Miroslav Zarić, PhD, full professor, Faculty of Technical Sciences, University of Novi Sad **Member:** Milena Frtunić Gligorijević, PhD, assistant professor, Faculty of Electronic Engineering, University of Niš **Member:** Gordana Rakić, PhD, assistant professor, Faculty of Sciences, University of Novi Sad **Member:** Igor Dejanović, PhD, full professor, Faculty of Technical Sciences, University of Novi Sad **Mentor:** Stéphane Ducasse, PhD, research director, Inria, Lille, France **Mentor:** Gordana Milosavljević, PhD, full professor, Faculty of Technical Sciences, University of Novi Sad
Note:	

Милици, мојој вјечној инспирацији.

Contents

Резиме

Увод

Софтверски системи се непрекидно мијењају и еволуирају. Значајан дио тих измјена односи се на структуру програма: преименовање елемената кода, издвајање дуплиране програмске логике или реорганизацију одговорности како систем расте [Vis01, MAP+08]. Рефакторисање је дисциплинован облик такве измјене. Оно мијења структуру изворног кода тако да његово споља видљиво понашање остане очувано [BR98, FBB+99].

Опдајк (Opdyke) је дао рану формализацију рефакторисања као измјена изворног кода које чувају понашање и дефинисао каталог објектно-оријентисаних (ОО) рефакторисања [Opd92]. Фаулеров (Fowler) каталог је касније утврдио рјечник и приближио праксу рефакторисања програмерима [FBB+99]. Робертс (Roberts), Брант (Brant) и Џонсон (Johnson) имплементирали су први систем за рефакторисање по тим дефиницијама, *Refactoring Browser*, и поставили темеље за рефакторисање подржано алатима [RBJO96, RBJ97]. Систем за рефакторисање је развојни алат који, на захтјев програмера, примјењује повезане измјене изворног кода и провјерава услове очувања понашања, које би иначе требало ручно анализирати. Такви системи су стандардни развојни алати чија поузданост непосредно утиче на квалитет и брзину свакодневних измјена кода, али и на спремност програмера да се ослоне на аутоматизоване измјене кода [KBDA16, WXZ+26].

Динамички типизирани објектно-оријентисани језици, као што су *Pharo*, *Python* и *JavaScript*, додатно отежавају аутоматско рефакторисање. Касно везивање (енгл. *late binding*) ограничава статичку анализу, па неке промјене у везивању имена (енгл. *binding*) и одабиру методе при позиву (енгл. *dispatch*) постају видљиве тек током извршавања програма [FMM+11]. Због тога семантички неисправан резултат рефакторисања може бити синтаксно исправан, а ипак изазвати грешку током извршавања или промијенити понашање програма. Компајлер у статички типизираним језицима открива многе неисправне резултате, али не и такве. Статички се могу провјерити синтаксна исправност и поједине структурне чињенице, али не и очување понашања за све програме и улазе. Систем зато може одбити структурно непримјењиву измјену и упозорити на познате семантичке ризике, али изостанак упозорења не гарантује да је понашање очувано.

Системи за рефакторисање морају пратити и еволуцију самог језика и његових

библиотека, па поједина рефакторисања временом постају непотпуна или застарјела. Ови изазови дјелимично објашњавају зашто се програмери не ослањају досљедно на аутоматизовано рефакторисање: дефекти и неочекивани резултати смањују повјерење [MHPB11, HO15], а архитектура система може отежати поновну употребу и проширивање логике трансформација [ACD+22].

Проблем истраживања

Очување понашања је кључно својство рефакторисања, али истовремено може ограничити употребу алата. Програмерима су често потребне и структурне измјене које не чувају понашање, на примјер при миграцији на нови програмски интерфејс (енгл. *application programming interface*, API), када је промјена понашања намјерна или неизбјежна [ACD+22]. Такву структурну измјену у овој дисертацији називамо трансформацијом, док појам рефакторисања задржавамо за измјене чији је циљ очување понашања програма. Термин операција употребљавамо када се тврдња односи и на рефакторисања и на трансформације.

У многим системима за рефакторисање структурна измјена и провјере очувања понашања могу се користити само као једна недјељива операција [ACD+22], па корисник нема могућност избора када очување понашања није нужно. Преостаје само директно мијењање модела програма, при чему се губи подршка алата за провјеру да ли је тражена структурна измјена примјењива. Оно што недостаје јесте средњи ниво: структурна измјена за коју систем и даље гарантује да се може конструисати и примијенити.

Када систем нуди само рефакторисања као недјељиве цјелине, поновна употреба логике (енгл. *reusability*) промјене изворног кода постаје тежа. На примјер, додавање или уклањање методе јесте провјерена структурна измјена од које се граде сложенија рефакторисања, али је доступна само у два облика: као рефакторисање, са провјерама очувања понашања и интеракцијом које у том контексту нису потребне, или као директна измјена модела програма, која губи предуслове примјењивости и дуплира анализу коју систем већ има [LT11]. Ниједан облик не одговара ономе ко тај корак уграђује у сложенију операцију.

Употријебљена самостално, иста измјена не мора да чува понашање, јер додавање методе може редефинисати наслијеђену. Зато се тврдња о очувању понашања везује за рефакторисање као цјелину, а не за поједине кораке од којих је састављено. Раздвајање трансформације од рефакторисања захтијева и раздвајање двију врста предуслова. Предуслови примјењивости (енгл. *applicability preconditions*) одређују да ли се структурна измјена може уопште конструисати. Предуслови очувања понашања (енгл. *behavior-preserving preconditions*) упозоравају да измјена која се може конструисати ипак може промијенити понашање програма [KWK04, ACD+22]. Без тог раздвајања не може се затражити измјена која задржава провјеру примјењивости, а свјесно изоставља провјеру очувања понашања.

У многим интерактивним системима за рефакторисање проблем се додатно

појачава тиме што је логика операције спрегнута са корисничким интерфејсом кроз уграђене интерактивне упите. Таква спрегнутост онемогућава покретање дате логике без интеракције са корисником, а управо такво покретање захтијевају писање скрипти, серијско извршавање и аутоматизовано тестирање.

Други дио проблема односи се на поузданост. Дефекти се јављају и у зрелим алатима [WXZ+26]. Ранији аутоматизовани приступи њиховом откривању често су генерисали синтетичке програме ради тестирања рефакторисања [DDGM07, SGSM10, DHT10]. Предност аутоматизованог генерисања је у томе што омогућава контролисано варирање улаза, али реални пројекти садрже зависности, хијерархије, обрасце позива и семантичке везе које је тешко генерисати. Глигорић и сарадници [GBL+13] тестирали су рефакторисања на реалном софтверу већег обима, користећи повратне информације компајлера као критеријум исправности (енгл. *test oracle*). У динамички типизираним језицима, у којима неисправан резултат рефакторисања може остати синтаксно исправан, тестирање рефакторисања не може се ослонити само на повратне информације компајлера, већ захтијева извршиви критеријум исправности понашања.

Предуслови су посебан извор непоузданости. Предуслов може бити изостављен или преслаб, па тако пропустити конфигурацију која мијења понашање, или може бити претјерано конзервативан, те тако одбити конфигурацију која би била прихватљива. Обје врсте грешака смањују предвидљивост система. Тестирање само исправних улаза не покрива гранања и посебне токове извршавања, а поменути дефекти налазе се управо у тим токовима унутар провјере предуслова.

Циљ истраживања, хипотезе и истраживачка питања

Циљ ове дисертације је да унаприједи поузданост и поновну употребу у системима за рефакторисање у динамички типизираним објектно-оријентисаним (ОО) језицима кроз два узајамно повезана доприноса:

1. референтну архитектуру која раздваја структурно примјењиве трансформације од рефакторисања са предусловима очувања понашања, те изолује корисничку интеракцију од језгра операције и
2. Refazz, метаморфни метод тестирања који користи тестове реалних пројеката као дјелимичну спецификацију очекиваног понашања и испитује параметре и одбијене предуслове.

Архитектура је специфицирана у поглављу 3 на начин који не зависи од конкретне имплементационе платформе. Имплементирана је у оквиру *Pharo* екосистема, а прототипска имплементација изабраних операција, ради провјере преносивости на друге динамички типизиране ОО језике, спроведена је у систему за рефакторисање *rope* за језик *Python*.

Pharo је изабран као примарно окружење за имплементацију и емпиријску евалуацију јер обезбјеђује постојећи систем за рефакторисање (који се у наставку

назива затеченим системом), моделе програма и измјена, реалне пројекте са извршивим тестовима и интерактивно развојно окружење. Refazz је такође имплементиран у *Pharo*-у.

Дисертација се бави начином на који се рефакторисања и трансформације структурирају, компонују, извршавају и тестирају. Она се не бави препорукама рефакторисања, откривањем лоших образаца у коду нити рударењем рефакторисања из историја верзија: те области помажу у одлуци коју измјену треба направити, док систем који се овдје проучава спроводи затражену измјену.

Истраживање полази од четири хипотезе. Прве двије односе се на архитектуру, а друге двије на метод тестирања.

X1 Архитектура у којој трансформације носе предуслове примјењивости, а рефакторисања им додају предуслове очувања понашања, повећава поновну употребу логике структурне измјене и смањује њено дуплирање када се уведе у постојећи систем за рефакторисање.

X2 Издвајање корисничке интеракције из основне логике операције омогућава да иста логика подржи и интерактивно и неинтерактивно извршавање, без двије одвојене имплементације.

X3 Постојећи тестови реалних пројеката откривају промјене понашања које рефакторисања изазивају, а структурне провјере пропуштају, у мјери у којој ти тестови покривају измијењено понашање.

X4 Испитивање неисправних и граничних параметара и принудно извршавање откривају дефекте предуслова које тестирање само исправних улаза не открива.

Свакој хипотези одговара једно истраживачко питање. Питање одређује шта се пројектује или мјери, а хипотеза какав се исход очекује. На питања RQ1 и RQ2 одговара архитектура у поглављу 3, док хипотезе X1 и X2 провјерава њена евалуација у поглављу 5. Метод Refazz одговара на RQ3 и RQ4 у поглављу 6, а X3 и X4 провјерава његова евалуација у поглављу 7.

RQ1 Како објединити трансформације и рефакторисања у градивне блокове погодне за поновну употребу и композицију?

RQ2 Како изоловати корисничку интеракцију од основне логике операције тако да иста логика подржи и интерактивни рад и неинтерактивно извршавање без графичког интерфејса?

RQ3 У којој мјери се постојећи тестови реалних пројеката могу користити као дјелимичне извршиве спецификације које откривају промјене понашања изазване рефакторисањем?

RQ4 На који начин испитивање простора параметара и принудно извршавање послије одбијања предуслова примјењивости или упозорења о очувању понашања откривају недостајуће или претјерано конзервативне предуслове које тестирање само исправних улаза не открива?

Хипотезе X1 и X2 провјеравају се мјерењем поновне употребе и дуплираности кода, студијама случаја и анализом издвајања интеракције над мигрираним дијелом система. Хипотезе X3 и X4 провјеравају се примјеном метода Refazz на пет реалних пројеката и класификацијом потврђених дефеката према режиму извршавања који их је открио.

Преглед стања у области

Преглед стања у области у дисертацији (поглавље 2) уводи историјат и терминологију рефакторисања, анализира претходне архитектуре система за рефакторисање и механизме поновне употребе, те разматра интеракцију, подршку за писање скрипти и методе провјере и тестирања. Он такође описује анатомију типичног система за рефакторисање: модел програма за семантичке упите, модел измјена за представљање одложених измјена изворног кода, предуслове и слој интеракције са корисником.

Значајан дио претходних рјешења развијен је у контексту језика *Java*, гдје су истраживане архитектуре система за рефакторисање, декомпозиција и спецификација рефакторисања, као и тестирање засновано на генерисаним програмима, понашању и предусловима. *Refactoring Browser* и *Eclipse LTK* раздвајају провјеру услова од одложених измјена [Rob99]. Микрорефакторисања и примитивна рефакторисања обезбјеђују мање кораке за изградњу већих измјена [SVEdM09,SdM10,HKT16,KWK04], а скрипте у алату *Wrangler* и библиотеке за преписивање кода обезбјеђују програмским клијентима команде или измјене на нижем нивоу апстракције [LT12a,RBD15].

У прегледу области се разматрају и постојећи приступи за тестирање рефакторисања. Они се разликују према начину добијања програма за тестирање, варирању улаза, контроли извршавања и провјерама исхода. Једна група приступа генерише мале програме или синтаксна стабла како би контролисано испитала структурне облике. Тако *ASTGen* провјерава структурне исходе над генерисаним програмима, а *SafeRefactor* генерише тестове којима пореди понашање програма прије и послије промјене [DDGM07,DHT10,SGSM10,SGM13].

Друга група примјењује рефакторисања на реалне програме и као показатеље користи грешке компајлера и изузетке система [GBL+13]. Поједини приступи искључују изабране предуслове, а резултујућу трансформацију провјеравају алатом *SafeRefactor*, чиме откривају претјерано конзервативне предуслове [SMG11,MGS+18].

Сваки од ових приступа обрађује дио проблема. Генерисани програми омогућавају контролу над синтаксном структуром, али тешко представљају семантичке везе реалног система. Грешке компајлера и изузеци откривају неисправан или прекинут исход, али не нужно и промјену понашања. Тестови могу открити промјену понашања

за случајеве које извршавају и провјеравају, али не доказују очување понашања за све могуће случајеве.

Ови приступи, дакле, раздвајају избор улаза од провјере исхода: избор улаза одређује који се програми, циљеви, параметри и режими извршавања испитују, а провјере исхода одређују које се грешке могу уочити и шта се из запажања не може закључити. Ниједан од ових показатеља сам по себи не покрива све аспекте, па их је потребно комбиновати.

У овој дисертацији рјешавамо сродне проблеме рефакторисања и тестирања, али у домену динамички типизираних ОО језика. Ти језици додају и насљеђивање са редефинисањем метода и одабир методе према класи примаоца, па се промјене понашања скривају и тамо гдје их функционални системи не сријеђу. Претходно размотрена рјешења се зато не могу једноставно искомбиновати и као таква пренијети: сваки циљни систем зависи од свог модела програма, модела измјена, информација о везивању имена и механизама за интеракцију и извршавање.

Преглед области обухвата и затечени систем за рефакторисање у *Pharo*-у, као конкретну полазну основу за анализу проблема истраживања и практичну имплементацију (одјељак 2.7).

Анализа затеченог система показала је да су рефакторисања и трансформације припадали одвојеним хијерархијама са неусаглашеним протоколима извршавања, да су многе операције биле имплементиране два пута, једном као рефакторисање и једном као готово истовјетна трансформација, те да су провјере предуслова и интерактивни упити били уграђени у саму логику операције. Отклањање тих недостатака захтијевало је нову архитектуру, а не појединачне исправке.

Референтна архитектура

Прву компоненту рјешења чини референтна архитектура, која редефинише однос између трансформација и рефакторисања (слика 3.1). Све појмове у наставку уводимо без формалних ознака, које се могу наћи у поглављу 3.

Обје врсте операција смјештене су у заједнички модел. Трансформација садржи логику структурне измјене и предуслове примјењивости, па може служити као провјерени градивни блок. У моделу је трансформација одређена предикатом примјењивости и функцијом конструкције измјене: за прихваћени улаз она мора произвести добро формиран програм, док за одбијени улаз нема резултата.

Рефакторисање се гради као декоратор (енгл. *decorator*) [GHJV95] око трансформације и додаје предуслове очувања понашања који упозоравају на познате ризике од промјене понашања (семантичке ризике). Предуслови очувања понашања не сужавају домен примјењивости, већ га дијеле на конфигурације без познатог ризика и конфигурације са упозорењима. Празан скуп упозорења значи само да имплементирана анализа није открила ризик, а не да је понашање доказано очувано. Рефакторисање се покреће када је очување понашања циљ, док се трансформација на којој је оно засновано може директно употријебити када то није случај. Таква употреба

не слаби тврдњу рефакторисања о очувању понашања, јер постоји експлицитан избор нивоа операције, а предуслови примјенљивости остају на снази. Неуспјела провјера примјенљивости зауставља операцију, док неуспјела провјера очувања понашања производи упозорење о којем одлучује политика извршавања.

Појединачни предуслови представљени су као објекти (слика 4.2). Предуслов представљен као објекат (енгл. *reified precondition*) провјерава себе над моделом програма и биљежи ентитете који су изазвали неуспјех. Позивалац зато може извијестити корисника о конкретном узроку, разликовати одбијање примјенљивости од упозорења и изабрати одговор примјерен контексту.

Композитна операција координише трансформације и рефакторисања над међустањима програма, поново користи њихове провјере и додаје само услове који зависе од више корака. Сваки корак композиције провјерава се над стварним међустањем на којем се извршава, а успјешна композиција враћа само коначно стање програма. Тврдња о очувању понашања односи се на композитну операцију као цјелину, а не на међустања. Пошто појединачни корак може да промијени понашање, композитно рефакторисање предуслове очувања понашања провјерава на свом нивоу.

Интеракција преко корисничког интерфејса издваја се у посебне објекте под називом *драјвери* (енгл. *interaction drivers*). Драјвер прикупља параметре, провјерава предуслове примјенљивости, тражи од рефакторисања да процијени предуслове очувања понашања, а неуспјеле провјере претвара у листу могућих сљедећих корака и приказује их кориснику као изборе (енгл. *choices*, слика 4.3).

Ова подјела је подлога за богатије корисничко искуство при интерактивном раду. На примјер, при уклањању класе не мора се приказати једно генеричко упозорење: драјвер може да утврди да ли класа има потклассе, стање или референце и да за сваки случај понуди одговарајућу операцију, као што су *Промијени родитеља потклассама*, *Пренеси стање у потклассе* или *Прикажи референце* (слика 4.4).

Основна логика трансформације или рефакторисања може се користити и без драјвера, па скрипте, серијски процеси и тестови постављају сопствену политику одлучивања и извршавају исту логику без блокирајућих дијалога. Архитектура тиме побољшава поновну употребу, композицију и могућност тестирања, а питање поузданости обрађује друга компонента рјешења.

Имплементација у окружењу *Pharo*

Архитектура је реализована миграцијом постојећег система за рефакторисање у *Pharo*-у (поглавље 4). Тиме је показано да се предложено раздвајање одговорности може постићи и у реалном, затеченом систему, а не само ако се пројектовање врши од нуле. Миграција је задржала *Pharo*-ове моделе програма и измјена као основу за имплементацију, а затечене и мигриране класе су постојале упоредо током поступне миграције. У мигрираној хијерархији, рефакторисања и трансформације дијеле заједничку апстрактну класу (слика 4.1), па обје врсте операција имплементирају исти полиморфни протокол извршавања. Тај протокол има три нивоа: (1) метода `execute`

генерише и примјењује измјене без корисничке интеракције; затим (2) методе generateChanges и performChanges раздвајају конструкцију измјена од њихове примјене, што омогућава преглед и композицију, а (3) методе prepareForExecution, checkPreconditions и privateTransform обезбјеђују појединачне фазе потребне композитним операцијама и драјверима.

Провјере су сада централизоване: предуслови враћају своје резултате, а кораци трансформације не јављају грешке и упозорења кроз кориснички интерфејс. Измјене се примјењују тек пошто припрема, провјере и конструкција успију. Неуспјех у било којој од тих фаза оставља циљни систем непромијењеним. То важи и за композитне операције: њихови кораци конструишу измјене над моделом програма, тако да међустања постоје у моделу, а не у циљном систему, а прикупљене измјене примјењују се на крају као једна цјелина.

Мигрирана архитектура је дио *Pharo* издања, чиме је стављена на располагање реалним корисницима.

Евалуација архитектуре

Евалуација архитектуре (поглавље 5) комбинује више повезаних показатеља успјешности миграције затеченог система на нову архитектуру и подржава хипотезе X1 и X2 за операције које су мигриране.

Поређење је вршено на затеченом систему за рефакторисање, имплементираном у верзији *Pharo 11*, и новом систему имплементираном у верзији *Pharo 14*. У мигрираном коду 35 непосредних позива моделу програма замијењено је експлицитном употребом трансформација. Два рефакторисања, додавање и уклањање методе, реализована су као декоратори око својих трансформација, а имплементирано је 35 драјвера за више фамилија рефакторисања (табела 5.1). У развојној грани, која је у вријеме писања *Pharo 15*, декоратора је девет, око шест различитих класа трансформација. Миграција читавог каталога још није довршена.

Најшира поновна употреба појавила се код трансформација које представљају цјеловите програмске операције, као што су додавање и уклањање методе (табела 5.2). Трансформацију додавања методе у новом систему директно користи 28 различитих операција, наспрам 5 у затеченом систему, док трансформацију уклањања методе користи 14 операција у новом наспрам 3 у затеченом.

Постојећи омотачи локалних измјена синтаксног стабла имају знатно мању поновну употребу: 8 од 11 нема ниједно мјесто поновне употребе, а свих 11 заједно имају само 4 поновне употребе. Ти омотачи потичу из затеченог дизајна, па је ово мјерење емпиријска провјера наслијеђеног рјешења, а не оцјена нечега што је миграција увела. Њихова слаба поновна употреба показује да представљање измјене као засебног објекта само по себи није довољно да операција постане корисна јединица композиције.

Анализа дуплираног кода спроведена је над пакетима који у оба система садрже рефакторисања, трансформације и њихове предуслове (табела 5.4). Упркос расту са

179 у затеченом на 243 датотеке у мигрираном систему и са 24.254 на 25.261 линију, број парова клонова смањен је са 104 на 32. Удио дуплираних линија пао је са 6,02% на 1,52%, што је релативно смањење од 74,7%. Удио дуплираних токена пао је са 8,00% на 2,22%. У претходном систему 79 од 104 пара клонова повезивало је класу рефакторисања с класом трансформације. Иста операција била је имплементирана двапут, једном као рефакторисање и једном као готово истовјетна трансформација. У мигрираном систему тај број пада на 19: потпуно мигриране фамилије операција нестају из извјештаја о клоновима, а 13 од 19 преосталих парова припада двјема још немигрираним операцијама (табела 5.5).

Ови резултати су у складу са очекиваним понашањем архитектуре, али као такви не доказују да је миграција једини узрок свих разлика између двије верзије система. Утицај промјена које нису дио миграције је ублажен на три начина: анализирани су само пакети непосредно промијењени преласком на нову архитектуру, провјерено је у којим фамилијама пакета се клонови губе, а за обје верзије коришћен је исти алат jscpd 4.2.5 и исто правило бројања. Подаци упућују на то да је раздвајање предуслова допринијело смањењу дуплираности више него сама декорација.

Композитно рефакторисање Extract Method служи као студија случаја за сложене операције. Затечена имплементација у једном току мијеша провјере, израчунавања потребна за издвајање методе, структурне промјене и интеракцију. Нова, композитна имплементација раздваја припрему, предуслове и конструисање, а промјену описује као низ постојећих операција [dSS17,SADT24]: додавање нове методе, замјену изабраног подстабла позивом и уклањање неупотријебљених привремених промјенљивих. Композитна верзија има 28% мање метода (34 умјесто 47), 30% мање линија (258 умјесто 370) и 51% нижу збирну цикломатску сложеност (46 умјесто 94, табела 5.6).

Двије доменски специфичне варијанте рефакторисања показују различите облике проширења. Издвајање *SUnit*[3] методе setUp мијења предуслове и кораке извршавања, али поново користи заједничку припрему пред издвајање. Издвајање методе за *Slang*[4] радни оквир задржава кораке извршавања и притом додаје трансформације које преносе одговарајуће прагме (анотације). Ови случајеви показују да композиција може замијенити или проширити дио поступка без копирања цјелокупне логике.

Евалуација другог истраживачког питања (RQ2) прати издвајање интеракције. Број метода које непосредно позивају стари механизам за пријављивање упозорења смањен је у читавом систему са 28 на 15, а заједнички број таквих метода за грешке са 163 на 98. Преостали позиви показују тачна мјеста на којима миграција није довршена. Иста конфигурисана операција покреће се и кроз интерактивни драјвер и кроз извршавање путем скрипти, без раздвајања на двије одвојене имплементације. Ови резултати подржавају хипотезе X1 и X2 везане за архитектуру система за рефакторисање, док се хипотезе X3 и X4 обрађују у наставку, у оквиру новог метода тестирања Refazz.

[3] *SUnit* је радни оквир за писање јединичних тестова у *Pharo*-у, претеча *JUnit*-а. Метода setUp припрема податке над којима се сваки тест извршава.

[4] *Slang* је подскуп језика *Smalltalk* који се преводи у C и користи за имплементацију виртуелне машине *Pharo*-а [MBBI18].

Метод тестирања Refazz

Представљена архитектура не гарантује исправност операција, нарочито у динамички типизираним језицима, гдје се компајлер не може користити за одбацивање семантички погрешног резултата. Из тог разлога у поглављу 6 се уводи Refazz, нови метод тестирања који комбинује метаморфно тестирање (енгл. *metamorphic testing*) [CCY98] са испитивањем различитих вриједности улазних параметара рефакторисања. Његов преглед је дат на слици 6.1.

У наставку ћемо користити појмове са сљедећим значењем. Под тестовима подразумијевамо постојеће тестове пројекта над којим се рефакторисање примјењује. Метод их не пише и не мијења, него их покреће прије и послије измјене. Параметри су подаци који одређују конкретно рефакторисање, на примјер ново име методе, циљна класа или изабрани дио кода, а не улази програма који се тестира. Дефекти који се траже су дефекти система за рефакторисање, а не пројекта. Пројекат је само улаз над којим систем за рефакторисање показује своје понашање.

Refazz полази од особине која дефинише рефакторисање и формулише један основни метаморфни однос. Пошто рефакторисање треба да очува понашање, тестови који покривају разматрани програмски код треба да прођу и прије и послије његове примјене. Ако су тестови прије измјене прошли, а послије ње нису, добијен је непосредан доказ да се промијенило понашање које тај скуп тестова покрива. Ако тестови прођу, не гарантује се исправност операције, јер понашање које није покривено или је слабо покривено тестовима може бити промијењено, а да се то не испољи током тестирања. Тестови конкретног реалног пројекта из тог разлога служе као његова дјелимична извршива спецификација понашања.

Сваки покушај конфигурише рефакторисање конкретним параметрима, при чему се разликују исправне конфигурације од неисправних, граничних и оних које могу нарушити понашање. Покушај рефакторисања је *одбијен* када га предуслов заустави прије почетка трансформације. Он је *прекинут* када је извршавање почело, али није произведен резултујући програм. Покушај је *довршен* када је резултујући програм произведен, након чега слиједи извршавање тестова ради провјере понашања. Прекид може настати када парсер или компајлер одбије измјену, као и због дефекта у самој трансформацији.

Метод разликује четири режима извршавања, MR1–MR4 (табела 6.1). MR1 и MR2 провјеравају полазну особину рефакторисања (да тестови пролазе након извршавања), а MR3 и MR4 испитују ограничења предуслова примјењивости и предуслова очувања понашања. MR3 може да заобиђе провјеру примјењивости јер је та провјера засебан објекат. MR4 може да настави послије упозорења јер о наставку одлучује политика, а не дијалог који блокира извршавање.

Параметри се генеришу стратегијама специфичним за рефакторисање, изведеним из семантике предуслова и груписаним у три фамилије. Стратегије имена производе неисправна, резервисана, нова и постојећа имена, селекторе са погрешним бројем аргумената и имена која редефинишу декларације у хијерархији. Стратегије пермутације мијењају редослијед параметара метода. Стратегије интервала

симулирају селекције кода за операције издвајања и укључују исправне, празне и помјерене границе интервала. Када операција има више параметара, метод варира параметре из једне фамилије, а остале параметре држи исправним. Тако се избјегава истовремено комбиновање више неповезаних неисправних вриједности и олакшава откривање узрока.

Refazz је реализован проширењем *MuTalk*[5] оквира за мутационо тестирање. Конфигурисано рефакторисање (операција са изабраним циљем и конкретним параметрима) представљено је као мутациони оператор. *MuTalk* обезбјеђује обилазак пројекта, избор тестова према степену покривености, изолацију покушаја, биљежење исхода и чување објекта мутације ради поновног извршавања. Поред наведеног, Refazz обезбјеђује и циљеве, стратегије параметара и политику предуслова за изабрани режим. За сваки циљ најприје се успоставља полазно стање, затим се конфигурише операција, спроводи или заобилази одговарајућа провјера, примјењује измјена, покрећу изабрани тестови пројекта и на крају поништава измјена. Овај циклус примијени–тестирај–поништи изолује сваки покушај од осталих и тиме спречава да утичу једни на друге.

Током извршавања прикупљају се записи о покушајима и њиховим резултатима. Записи се групишу према испитиваном рефакторисању, режиму извршавања, употријебљеној стратегији, исходу покушаја и релевантним особинама улаза. Представник сваке групе се поново извршава, и притом се прегледају предуслови, измјене, изузетак и исходи тестова. Кандидат се уноси у каталог дефеката тек када поновно извршавање, или еквивалентан сачуван доказ, утврди конкретан неуспјешан сценарио, припише га систему за рефакторисање и покаже да се разликује од већ увртених дефеката.

При класификацији се почетни показатељ проблема и доказ његовог узрока биљеже одвојено. Неуспјешно извршавање теста, изузетак, прекид или успјешно извршавање послије одбијања могу бити почетни показатељи. Поновно извршавање и увид у изворни код су пратећи докази. Сваки потврђени дефект добија један примарни узрок: дефект трансформације или предуслова. Дефекти предуслова се даље дијеле на недостајуће, претјерано конзервативне и погрешно смјештене предуслове.

Евалуација метода тестирања

Евалуација метода Refazz (поглавље 7) спроведена је на пет намјенски изабраних реалних *Pharo* пројеката, из различитих домена и различите величине, који се могу поновљиво учитати, имају нетривијалне тестове који успјешно пролазе и садрже разноврсне језичке конструкције. Изабрани су пројекти: Artefact (генерисање PDF докумената), AVL (структура података), Graph algorithms (алгоритми над графовима), Microdown (обрада Markdown текста) и Soup (обрада HTML-а), са покривеношћу метода сопственим тестовима између 60% и 87% и чворова синтаксног стабла између 63% и 85% (табела 7.1). Евалуација је обухватила седам рефакторисања и 255.699 покушаја

[5] https://github.com/pharo-contributions/mutalk/

извршавања. По пројектима, *AVL* је имао 24.291 покушај, *Soup* 39.720, *Graph algorithms* 56.877, *Microdown* 60.877, а *Artefact* 73.934. Пуни експерименти трајали су од 19 минута до сат и 24 минута, уз ограничење од пет минута за свако рефакторисање по стратегији параметара, режиму извршавања и пројекту.

Испитана су рефакторисања: Add Argument, Composite Extract Method, Extract Method, Inline Method, Rename Variable, Rename Class и Rename Method. Composite Extract Method је композитна имплементација Extract Method-а која сложену измјену остварује као низ поново употребљивих елементарних трансформација и тиме провјерава механизам композиције архитектуре.

Каталог дефеката садржи 19 јединствених потврђених дефеката: 12 дефеката у трансформацијама и 7 дефеката у предусловима (3 недостајућа предуслова, 2 претјерано конзервативне провјере и 2 погрешно смјештене провјере). Детаљан каталог, у којем је сваки дефект повезан са режимом и стратегијом који су га открили и са пријавом у систему за праћење грешака, налази се у додатку C.

Шест дефеката пронађено је у Extract Method, десет у Composite Extract Method, а три у осталим испитаним рефакторисањима.

Према захваћеном аспекту система, један дефект односио се на имена, опсеге и везивање, пет на ток управљања и повратне вриједности, укључујући блокове и нелокалне повратке, девет на структурну и синтаксну исправност, попут неисправних синтаксних стабала и парсерских грешака, и четири на прецизност и мјесто предуслова.

Неуспјело извршавање тестова открило је 7 од 19 потврђених дефеката. Међу њима су: промјена садржаја низа литерала, увођење рекурзије при издвајању кода са позивом super, промјена понашања нелокалног повратка (енгл. *non-local return*) и ненамјерно редефинисање методе у којем је систем требало да упозори прије примјене семантички ризичне трансформације, а није то учинио.

Од преосталих дванаест, једанаест се прво показало у покушајима који нису довршени, а један кроз довршено принудно извршавање и накнадну ручну инспекцију. Шест структурних дефеката произвело је неисправна стабла, неформатиран изворни код, грешку при парсирању или изузетак, тако да до фазе извршавања тестова није ни дошло.

Тестови су открили промјене понашања, прекиди су открили отказе прије него што је семантичка провјера уопште могла да се покрене, а довршено принудно извршавање омогућило је процјену изводљивости одбијене конфигурације. Сви они заједно били су неопходни за процјену исправности рефакторисања.

Од 19 дефеката, 13 је приписано режиму MR1, пет MR2, један MR3, а ниједан MR4 (табела 7.4). Три недостајућа предуслова, једна претјерано конзервативна провјера и једна погрешно смјештена провјера очувања понашања приписани су MR2, а још један претјерано конзервативан предуслов MR3, тако да MR2 и MR3 заједно обухватају шест дефеката, односно 32% каталога.

MR4 је испитао извршавања након упозорења о очувању понашања, али није донио ниједан нови потврђени дефект. Пролазак тестова није био довољан доказ јер ризично понашање можда није покривено тестовима, па су успјешни тестови у тим случајевима

остали неубједљиви.

Конкретне стратегије показале су додатну вриједност испитивања простора параметара. Стратегије пермутације параметара открилe су недостајућу провјеру приликом рефакторисања додавања параметра. Стратегија која бира наслијеђени селектор открила је погрешно смјештену провјеру редефинисања у издвајању методе. Стратегије граничних интервала открилe су недостајуће провјере при издвајању методе, као и случајеве у којима је принудно извршавање показало да је одбијена измјена ипак изводљива. Стратегија која бира постојеће име дијељене промјенљиве открила је у преименовању класе провјеру која је увијек заустављала операцију, а испољавала се тек касно, током извршавања. Исправка ју је претворила у експлицитан предуслов очувања понашања који драјвер приказује као упозорење.

Сви дефекти су пријављени у систему за праћење грешака, а за шест је исправка до сада интегрисана у развојну грану пројекта *Pharo*. Поред дефеката самог система за рефакторисање, метод је открио и регресију у компајлеру, која је пријављена, али није урачуната у каталог.

Ови налази подржавају хипотезе X3 и X4 у обиму испитаних пројеката, рефакторисања и режима извршавања. Тестови пројеката послужили су као дјелимичне извршиве спецификације понашања и открили седам дефеката. Испитивање неисправних и граничних улаза открило је пет дефеката предуслова, а принудно извршавање још један. Тестирање само исправних улаза није открило ниједан од тих шест дефеката.

Пријетње ваљаности укључују ручну класификацију аномалија, непотпуност критеријума исправности, намјенски избор пројеката и непостојање независне аутоматизоване основе за поређење у *Pharo*-у. У дисертацији су ове пријетње ублажене груписањем по стабилним обиљежјима извршавања, поновним извршавањем представника, повезивањем сваког дефекта са пријавом у систему за праћење грешака и каталогом дефеката.

Преносивост рјешења

Емпиријски резултати ове дисертације потичу претежно из *Pharo* окружења. Ради провјере преносивости на друге динамички типизиране објектно-оријентисане језике, основне поставке архитектуре (раздвајање трансформација и рефакторисања, декорација и композиција) имплементиране су у оквиру *rope*-а (независно развијеног система за рефакторисање за *Python*). Провјера је спроведена над двије операције: преименовањем методе и измјеном потписа методе (енгл. *Change Signature*). Оне су изабране зато што заједно покривају оба механизма архитектуре: прва декорацију над једном трансформацијом, а друга композицију више корака. Кораци трансформације могу се засебно конфигурисати, провјеравати и извршавати.

Имплементиране провјере показују и језички специфично ограничење које архитектура предвиђа: шта провјера може да утврди зависи од тога шта модел програма циљног језика умије да разријеши. За позиве са познатим примаоцем и за

методе у разријешеној хијерархији провјера поуздано утврђује чињенице, на примјер да ли преименовање изазива сукоб имена. Када се тип примаоца не може одредити прије извршавања, таква чињеница остаје недоступна, па провјера умјесто тврдње издаје упозорење о којем одлучује позивалац. Иста подјела на одлучиве провјере и пријављене ризике важи и у *Python*-у, само са другачијим ограничењима.

Доказ концепта показује да се архитектонске улоге могу реализовати у независно развијеном систему. Могућности које систем за то мора да пружи наводи табела 5.7. Он, међутим, не обухвата пренос цијелог каталога операција, интеграцију с развојним окружењем и пренос самог метода Refazz.

Метод тестирања Refazz евалуиран је у *Pharo* окружењу. Његов пренос би захтијевао програмски интерфејс за рефакторисање, моделе програма и измјена који чувају изворни код, контролу над параметрима и обрадом предуслова, изолацију или поуздано поништавање измјена и пројекте са извршивим тестовима. Додатно, стратегије параметара морале би се прилагодити синтакси и правилима везивања циљног језика, тако да би пренос метода Refazz захтијевао засебну имплементацију и емпиријску евалуацију.

Закључак и правци даљег развоја

Закључак и правци даљег развоја дати су у поглављу 8 које сумира доприносе и анализира могућности за унапређење рјешења.

Ова дисертација показује да систем за рефакторисање не мора бирати између претјерано конзервативног одбијања и непровјерене измјене програма. Предложена архитектура омогућава да се иста логика структурне измјене користи на два начина: као рефакторисање, када је циљ очување понашања, и као провјерена трансформација, када је промјена понашања намјерна. Издвајањем корисничке интеракције и раздвајањем двију врста предуслова операције постају погодније за поновну употребу, композицију и неинтерактивно извршавање. Други, повезани допринос је Refazz, метод који испитује исправност трансформација и њихове предуслове на реалним пројектима.

У дијелу система обухваћеном миграцијом повећана је поновна употреба кључних трансформација, а удио дуплираних линија у анализираним пакетима смањен је са 6,02% на 1,52%. Евалуацијом метода Refazz потврђено је 19 јединствених дефеката. Неуспјех тестова пројеката открио је седам дефеката. Према режиму извршавања, пет дефеката предуслова приписано је испитивању неисправних и граничних улаза, а један принудном извршавању. Тестирање само исправних улаза није открило ниједан од тих шест дефеката. Шест исправки интегрисано је у развојну грану пројекта *Pharo*. Ових 19 дефеката не представља потпун попис, јер њихов број зависи од изабраних пројеката, рефакторисања, стратегија и покривености тестовима.

Доказ концепта у систему *rope* показује да се основне архитектонске улоге могу реализовати и у независно развијеном систему за *Python*. Пренос метода Refazz захтијева засебну имплементацију и евалуацију, при чему би сам приступ остао исти.

Ограничења евалуације указују на правце даљег рада. Преостало је довршити миграцију каталога рефакторисања на нову архитектуру и документовати одлуке које се понављају при миграцији сваког рефакторисања, као каталог за програмере сличних система. Метод Refazz може се укључити у провјере прије објављивања нових верзија *Pharo*-а и проширити на додатна рефакторисања.

Шира примјена на друге динамички типизиране језике захтијевала би засебну евалуацију, јер зависи од интерфејса конкретног система. Послије *Python*-а, непосредан сљедећи циљ преноса је класно заснован језик као што је *Ruby*, док би вишепарадигмални језици попут *JavaScript*-а испитали шири опсег језичких конструкција. Исто раздвајање може се евалуирати и у статички типизираним системима, гдје компајлер утврђује више чињеница и обезбјеђује додатну провјеру исхода, па би однос између архитектонских провјера и емпиријских критеријума исправности могао бити другачији.

Abstract

Refactoring engines automate structural changes while aiming to preserve observable behavior. Developers also need checked structural changes that do not guarantee behavior preservation, for example when migrating an API. Existing engines often expose either a complete refactoring or a lower-level rewrite that leaves applicability checks to its caller. In dynamically typed languages, testing either level is difficult because a syntactically valid result can still change binding, dispatch, or control flow. This thesis addresses the architecture and testing of these operations.

The thesis presents a reference architecture that separates structural applicability, behavior-preserving checks, and interaction policy. Transformations implement checked structural changes, refactorings decorate them with behavior-preserving checks, and interaction drivers keep parameter collection and warning decisions outside the operation core. The engine can therefore expose both refactorings and behavior-agnostic transformations, and the caller chooses between the two. It can also compose operations and run the same core under interactive and headless execution.

The architecture was realized by migrating Pharo's refactoring engine. In the analyzed code, Add Method and Remove Method are reused by 28 and 14 distinct refactorings and transformations, respectively. Duplicated lines in the analyzed packages decreased from 6.02% to 1.52%, a relative reduction of 74.7%. Case studies of a composite Extract Method and two domain-specific refactorings show that the resulting operations compose and that a composed sequence can be extended. Pharo is the primary empirical setting. A proof of concept in Python's *rope*, on two representative operations, shows that separation, decoration, and composition can transfer to an independently developed engine.

The thesis also presents Refazz, a metamorphic testing method for refactoring engines. For completed refactoring applications, Refazz tests whether selected project tests remain passing. Its four execution modes cover valid input and parameter exploration under normal checks, forced execution after an applicability rejection, and continued execution after a behavior-preserving warning. Refactoring-specific parameter strategies generate configurations on real projects.

Across 255,699 refactoring attempts involving seven refactorings and five projects, Refazz confirmed 19 defects: 12 transformation defects and 7 precondition defects, including 3 missing, 2 overly conservative, and 2 misplaced checks. Project-test failures exposed seven of these defects. Six defects required parameter exploration or forced execution. Fixes for six of the 19 were integrated into Pharo.

List of Figures

List of Tables

Listings

List of Abbreviations

API	application programming interface
AST	abstract syntax tree
DSL	domain-specific language
IDE	integrated development environment
JDT	Java Development Tools (Eclipse)
LoC	lines of code
LTK	Language Toolkit (Eclipse)
OO	object-oriented
PoC	proof of concept
PR	pull request
PSI	Program Structure Interface (IntelliJ Platform)
UI	user interface

Labels used throughout the thesis:

H1–H4	hypotheses (Section 1.3)
RQ1–RQ4	research questions (Section 1.4)
MR1–MR4	execution modes of Refazz (Table 6.1)

Chapter 1

Introduction

Software systems change continuously, and much of that change is structural: developers rename program elements, extract duplicated logic, and reorganize responsibilities as a system grows [Vis01, MAP+08]. Refactoring is the disciplined form of this activity. It changes the structure of source code while preserving its observable behavior [BR98, FBB+99]. Opdyke provided an early formalization and catalog of object-oriented refactorings as behavior-preserving source modifications [Opd92]. Fowler's catalog later helped establish the practitioner vocabulary and design practice of refactoring [FBB+99].

Roberts, Brant, and Johnson built the first refactoring engine, the Refactoring Browser, on Opdyke's definitions and established tool-supported refactoring [RBJO96, RBJ97]. A refactoring engine executes requested refactorings by applying coordinated source changes and checking preconditions that would otherwise require manual inspection. A precondition is a condition that must hold before the refactoring proceeds. Refactoring support is now standard infrastructure in integrated development environments (IDEs), so its reliability directly affects everyday software evolution. Recent studies still report defects and architectural constraints in refactoring engines [KBDA16, WXZ+26]. This thesis addresses both concerns, engine reliability and engine architecture.

Dynamically typed object-oriented languages, such as Pharo, Python, and JavaScript, make automated refactoring more challenging. With late binding and limited static information, some binding and dispatch changes go undetected until execution. The result of a faulty change may remain syntactically valid, and the fault may surface only at runtime. In a statically typed language, name and type checks would detect some of these errors, but an error that satisfies these checks would still go undetected. In a dynamically typed language, even fewer issues are identified before execution. Engines must also evolve alongside the language and its frameworks, and engine architecture can make transformation logic hard to reuse or extend [ACD+22]. Individual refactorings can therefore become incomplete or outdated. These pressures help explain why developers do not rely consistently on automated refactoring: bugs and unexpected results reduce trust [MHPB11, HO15].

Behavior preservation, the property that defines refactoring, is also the source of a recurring limitation. Developers frequently need a structural change without the full behavior-

preserving level, for example, when migrating an application programming interface (API) where the change is intentionally behavior-altering [ACD+22]. This thesis uses *transformation* for such behavior-agnostic source changes, and reserves *refactoring* for changes where behavior preservation is the goal.

1.1 Problem Statement

In many refactoring engines, the structural change and the behavior-preserving checks are exposed as one operation [ACD+22]. The caller receives a complete refactoring, not a selectable behavioral level. At the other extreme, lower-level rewriting or transformation interfaces may leave applicability checks (can the structural change be constructed?) to the caller. This gap between complete refactorings and unchecked rewrites raises the cost of constructing and reusing operations inside the engine. It also raises the cost of testing and maintaining the engine as the language evolves.

1.1.1 Architecture: Reuse, Preconditions, and Interaction

This gap has architectural consequences for reuse, composition, preconditions, and interaction.

- **Reuse and composition:** A structural step, such as adding or removing a method, can be useful both inside a behavior-preserving refactoring and on its own. Larger refactorings are often composed from these recurring steps. When the engine exposes that step only through complete refactorings or low-level model operations, the step cannot be reused at the intended level. A caller can reuse a complete refactoring, which brings behavior-preserving checks and user interaction that a composite or domain-specific change may not want [ACD+22]. The alternative is raw tree rewriting, which duplicates analysis the engine already implements and shifts checking responsibility onto the caller [LT11]. Composition then stays tied to behavior-preserving workflows, or it loses the applicability checks that the engine provides.
- **Precondition separation:** Preconditions contribute to refactoring correctness [KWK04], but existing engines do not always separate applicability preconditions from behavior-preserving preconditions [ACD+22]. An applicability precondition checks whether the structural change can be constructed. A behavior-preserving precondition checks whether the change might alter behavior. Without this separation, a caller cannot request the structural change while retaining applicability checks and deliberately setting aside a behavior-preserving check.
- **Interaction coupling:** Refactorings are most often invoked interactively through a user interface (UI) or keyboard shortcuts. In many interactive engines, hardcoded blocking dialogs and prompts couple the core transformation logic to that interface. Scripting and automated testing require the same core logic to run without a user interface, and this coupling prevents that. This thesis uses *headless* for such execution.

1.1.2 Reliability and Engine Evolution

Refactoring engine defects remain common across mature tools [WXZ+26]. Earlier automated approaches to finding them generated synthetic programs or test cases to exercise refactorings [DDGM07,SGSM10,DHT10]. Synthetic inputs provide controlled variation, while real projects provide the dependencies and semantic coupling that are difficult to generate. Gligoric *et al.* tested refactorings on real software at scale and used compiler feedback as an oracle [GBL+13]. However, compiler feedback cannot expose a faulty result that still compiles, in any language. In dynamically typed languages, it also misses errors that a type checker would reject, so detection rests on an executable semantic oracle.

Preconditions are a second reliability concern. A missing precondition can let an invalid or behavior-changing configuration through, while an overly conservative applicability precondition can reject one whose structural change can be constructed. Both reduce the predictability of the engine, and valid-input testing does not exercise the precondition paths where these defects appear.

Refactoring engines also depend directly on the syntax and semantics of the language, and both evolve over time. As language complexity grows, for example, with modern features in C++ or Rust [TCGS23], refactorings become harder to implement and verify.

1.2 Research Objective and Scope

The focus of this thesis is on dynamically typed object-oriented languages. Late binding and reflection leave more behavior-preserving facts undecidable before execution, so an engine must report warnings and let the caller decide. Compilation is not a sufficient oracle in any language, so the thesis relies on executable project tests. The proposed architecture could also be evaluated in a statically typed engine, where more behavior-preserving checks become decidable, but that is outside the empirical scope of this thesis.

Inheritance with override and dispatch based on the receiver's class are where behavior changes can go unnoticed. An extracted method can silently override an inherited one, and a renamed selector can affect unrelated receivers. Classes also shape the choice of behavioral level in the engine itself. The caller instantiates either the transformation class or the refactoring class that adds behavior-preserving checks on top. The languages studied are class-based.

The goal of this thesis is to improve reliability and reuse in dynamically typed object-oriented refactoring engines. The research addresses this goal by separating two behavioral levels. Behavior-preserving refactorings are used when preservation is the goal. Behavior-agnostic transformations are used when a developer needs a checked structural change without the full refactoring layer. The work has three parts.

First, the thesis defines a reference architecture for refactoring engines and describes its roles in language-independent terms. The architecture separates applicability checks from behavior-preserving checks, and transformations from refactorings. It isolates interaction decisions from both kinds of operation. It supports defining general-purpose and domain-specific refactorings through composition of reusable transformations. A domain-specific

refactoring adapts a general structural change to a framework-specific or language-specific constraint.

Second, the thesis defines a testing method for refactoring engines. The method, implemented as Refazz, combines parameter exploration with forced execution past applicability rejections and behavior-preserving warnings. It treats project tests as partial executable behavioral specifications and relies on MuTalk-supported replay[1] during outcome triage. It is designed to expose transformation defects and precondition defects, including missing, misplaced, and overly conservative checks.

Third, the thesis realizes and evaluates the architecture and testing method in Pharo. Pharo provides the primary empirical setting: a mature legacy refactoring engine, existing program and change models, and real projects with executable tests. The thesis also reports a scoped proof of concept in the Python *rope* engine[2] for Rename Method and Change Signature. That proof of concept provides additional evidence for the selected architectural roles in another dynamically typed language, although the main architecture evaluation and the Refazz evaluation remain Pharo-based.

The thesis concerns how refactorings and transformations are structured, executed, and tested. It does not address refactoring recommendations, code-smell detection, refactoring mining from version histories, or general program repair. Those areas decide which change to make. The engine studied here carries out a requested change.

1.3 Hypotheses

The research is guided by four hypotheses:

H1 (Architecture & Reuse): An architecture in which transformations carry applicability preconditions and refactorings decorate them with behavior-preserving preconditions increases the reuse of structural-change logic and reduces its duplication when it is introduced into an existing engine.

H2 (Execution & Decoupling): Isolating user interaction from the core logic of an operation lets the same logic support both interactive and headless execution without two separate implementations.

H3 (Semantic Oracles): Existing test suites of real-world projects expose behavior changes that refactorings cause and that structural checks miss, to the extent that the tests cover the changed behavior.

H4 (Boundary Verification): Exploring invalid and boundary parameters and forcing execution expose precondition defects that valid-input testing does not.

[1] MuTalk is Pharo's mutation-testing framework: https://github.com/pharo-contributions/mutalk/.

[2] *rope* is an open-source refactoring library for Python, used by editor and IDE integrations: https://github.com/python-rope/rope.

1.4 Research Questions

Each hypothesis is paired with one research question. The question states what is designed or measured, and the hypothesis states the expected outcome. RQ1 and RQ2 are answered by the architecture in Chapter 3, and H1 and H2 are checked by its evaluation in Chapter 5. RQ3 and RQ4 are answered by Refazz in Chapter 6, and H3 and H4 are checked by its evaluation in Chapter 7.

RQ1: How can a refactoring engine be architected to unify behavior-agnostic transformations and behavior-preserving refactorings into reusable, composable building blocks?

RQ2: How can user interaction be isolated from core refactoring logic while preserving both interactive guidance and headless batch execution?

RQ3: To what extent can existing test suites from real-world projects act as partial executable behavioral specifications that expose behavior changes caused by refactorings?

RQ4: How do parameter exploration and forced execution past an applicability rejection or a behavior-preserving warning reveal missing or overly conservative refactoring preconditions that valid-input testing does not expose?

1.5 Research Method

The research proceeds from problem analysis through artifact design, implementation, and empirical evaluation. Chapter 2 combines a review of refactoring engine research with an analysis of the legacy Pharo engine to derive the architectural and testing requirements. Chapters 3 and 4 define the architectural roles and realize them through an evolutionary migration of Pharo's refactoring engine. Chapter 5 evaluates the realized architecture through source-based measurements, migration evidence, composite-refactoring case studies, and a scoped cross-language proof of concept in the Python *rope* engine. Chapter 6 defines the testing method, and Chapter 7 evaluates it through refactoring attempts over selected real-world projects followed by replay, inspection, and unique-defect classification.

The unit of analysis changes with the research question. RQ1 and RQ2 concern architectural roles and their realization in engine code, while RQ3 and RQ4 concern refactoring attempts, observed outcomes, and confirmed defects. Table 1.1 connects each question to its research activity, principal evidence, and empirical scope. The evaluation chapters report the sampling, procedures, results, and threats to validity for each study.

1.6 Contributions

This thesis makes two primary contributions, each established through implementation in Pharo and empirical evidence.

Contribution 1. *A reference architecture for transformation/refactoring unification in dynamically typed object-oriented engines.*

Table 1.1: Traceability from research questions to research activities, evidence, and empirical scope.

RQ	Research activity	Principal evidence	Empirical scope
RQ1	Architecture design, migration analysis, case studies, and a scoped cross-language proof of concept	Reuse-client and direct-replacement counts, clone analysis, composite cases, and the *rope* implementation and tests	Pharo 11 and 14 engine snapshots, migrated Pharo operations, and two *rope* operations
RQ2	Interaction-separation design and Pharo migration analysis	Interaction-layer inventory, signaling counts, and interactive and headless invocation of the same core operation	Structural decoupling in the migrated Pharo operations
RQ3	Real-project refactoring testing and defect confirmation	Baseline and post-refactoring project-test outcomes, replay, and the issue-linked defect catalog	Five project test suites, seven refactorings, and behavior exercised by those suites
RQ4	Parameter exploration and within-study execution-mode attribution	Strategy and mode records, forced executions, replay, and unique-defect classification	Five projects, seven refactorings, and the studied parameter strategies and execution modes

This thesis revisits the relationship between refactorings and transformations [BGH07, ACD+22] and defines the architectural roles in language-independent terms. A *transformation* is a structurally applicable source change guarded by *applicability* preconditions, so it can be reused as a building block that keeps those checks. A *refactoring* is a transformation *decorated* with *behavior-preserving* preconditions. The caller can therefore request the same change at either behavioral level, behavior-preserving or behavior-agnostic. The refactoring keeps its behavior-preserving claim. *Interaction drivers* isolate parameter collection, warning resolution, and preview from the refactoring core, so the same logic runs under interactive use and headless execution.

Evidence. The architecture was realized by migrating Pharo's legacy engine and has shipped in a released Pharo version. Reuse counts and a clone-detection study demonstrate the reuse enabled by the architecture. Add Method is reused by 28 refactorings and transformations in Pharo 14, up from 5 in Pharo 11. Duplicated lines decreased from 6.02% to 1.52%, a relative reduction of 74.7%. The composite Extract Method refactoring case study and two domain-specific extensions demonstrate composition of reusable transformations. A scoped proof of concept in the Python *rope* engine introduces the transformation/refactoring separation, decoration, and composition. It does so without replacing the engine's existing program

and change models. The Rename Method proof of concept demonstrates decoration around one transformation. The Change Signature proof of concept demonstrates composition of parameter-level steps.

Contribution 2. *Refazz, a metamorphic testing method for refactoring engines that uses project tests as a partial semantic oracle.*

Rather than relying on compiler checks or synthetic code generation alone, Refazz exploits the behavior-preserving property of refactorings. It defines one metamorphic relation [CCY98]: the selected project tests that pass on the original project must also pass after the engine applies the refactoring without reporting a warning. Failing tests provide direct evidence of a behavior change covered by the suite, while passing tests provide incomplete evidence of preservation. Four execution modes cover valid input and parameter exploration under the engine's normal checks, forced execution after an applicability rejection, and forced execution after a behavior-preserving warning. The first two modes test the relation, while the two forced modes use the same test observations as diagnostic evidence. Forced execution relies on the architecture: applicability is a separate check, and the core runs without UI prompts. Each configuration is retained for replay during manual triage.

Evidence. Refazz was applied to Pharo's engine across seven refactorings and five projects. It executed 255,699 attempts and confirmed **19 defects**: 12 transformation defects and 7 precondition defects (3 missing preconditions, 2 overly conservative checks, and 2 misplaced behavior-preserving checks). Project-test failures exposed seven of the confirmed defects. Six defects were attributed to parameter exploration or forced execution rather than to valid-input mode. Six confirmed defects were found in Extract Method, ten in Composite Extract Method, and three in the other studied refactorings. Fixes for six of the confirmed defects have been integrated into the Pharo development branch. These counts are lower bounds for the studied projects, strategies, and oracle.

1.7 Outline

The remainder of this thesis is structured as follows:

Chapter 2 establishes the theoretical foundations of refactoring and reviews the state of the art in refactoring engines. It analyzes the legacy Pharo engine to locate the architectural causes of rigidity and duplication, and derives the requirements addressed by the rest of the thesis.

Chapter 3 presents the reference architecture. It defines a language-independent model of transformations and refactorings and presents reified preconditions and interaction drivers.

Chapter 4 details the implementation of this architecture within the Pharo ecosystem and shows how the legacy engine was migrated to the new design.

Chapter 5 evaluates the structural improvements using software metrics, migration evidence, and case studies of composite refactorings. It then reports a scoped cross-language proof of concept in the Python *rope* engine.

Chapter 6 introduces Refazz. It explains how the method combines project tests, parameter exploration, and forced execution.

Chapter 7 evaluates Refazz through an empirical study of Pharo's refactoring engine and presents the confirmed defects discovered during the process.

Chapter 8 summarizes the contributions, restates the scope, and outlines future directions, including the requirements for transferring the approach to other dynamically typed object-oriented languages. Appendices A to D provide the program-model interface map, the legacy Extract Method mechanics, the confirmed-defect catalog, and per-project data.

Chapter 2

Background and Related Work

2.1 Introduction

In an integrated development environment (IDE), an automated refactoring often appears as a single command: rename this method, extract this expression, or move this class. Behind that command, the engine works in several stages. It identifies the selected entities, analyzes the possible effects of the change, and decides whether the change may proceed. It then constructs the change and applies it within an interactive or scripted workflow. A defect in any of these stages can make the engine reject a valid change or produce a program that no longer behaves as intended. The architecture of the engine determines how the logic of these stages can be reused in new operations and how the engine can be tested.

This chapter supplies the background for both parts of the thesis, the reference architecture and the testing method. It traces how refactoring developed into an established IDE capability and fixes the terminology used throughout. It then presents a common engine pipeline and the constraints of dynamically typed object-oriented languages. With that vocabulary, it compares engine architectures, reuse mechanisms, interaction models, and testing approaches, and examines the Pharo Refactoring Browser as the baseline for this work. The chapter ends with the research gap and the four requirements that the following chapters address.

2.2 Evolution of Refactoring

Programmers have restructured code for a long time without intending to change its behavior. Earlier restructuring and program-transformation work studied such changes before object-oriented refactoring became a standard vocabulary [GN93]. Object-oriented refactoring connected this practice to design improvement and made behavior preservation an explicit requirement [Opd92, MT04]. This requirement distinguishes refactoring from unrestricted source rewriting. It does not imply that every refactoring improves every quality attribute. The effect of applying a refactoring depends on the selected refactoring, the program, and the developer's goal.

Opdyke [Opd92] provided the first extensive formalization of object-oriented refactoring.

His work described program changes such as moving a method or a variable to another class and reorganizing inheritance hierarchies, together with conditions intended to preserve behavior. A refactoring was described by its target, the transformation to perform, and the conditions under which that transformation preserves behavior. Opdyke and Johnson [OJ93] also showed how smaller refactorings could support larger design changes, such as introducing abstract superclasses.

Roberts, Brant, and Johnson [RBJO96,RBJ97] transferred these ideas into the first practical Smalltalk refactoring engine. The Refactoring Browser connected program analysis and automated program transformation to an interactive workflow that is still visible in today's IDEs: a developer selected a program entity and an operation, and the engine checked conditions, constructed the change, and updated the program. Roberts [Rob99] later described the analysis functions needed to check those conditions and to support practical refactoring in Smalltalk.

Fowler and colleagues [FBB+99] carried refactoring from the Smalltalk and framework-refactoring literature into a practitioner guide for Java programmers. The book connected code smells, named refactorings, motivations, mechanics, and worked examples. It gave developers a way to recognize design problems and change code in small behavior-preserving steps. Its role differed from that of Opdyke's formalization and of the Refactoring Browser implementation: it presented refactoring as a disciplined part of incremental design practice, including cases where developers applied the steps manually rather than through an IDE. The second edition [Fow18] shifted the examples to JavaScript, generalized several operation names, and reflected a setting in which many common refactorings were already automated by development environments.

IDE integration made automated refactoring part of day-to-day development. Studies of Eclipse activity show that developers use this support in mixed ways. Tool-assisted refactorings are often batched and interleaved with other edits, while many refactorings are still performed manually [MHPB09,NCV+13]. This mixed practice matters for engine design: an engine must support checked interactive commands, but it also needs feedback and control points for partial automation, repeated operations, and workflows that combine manual and automated changes.

Refactoring research subsequently expanded in several directions. Engines were built for languages whose semantics make different demands on analysis and transformation. Wrangler [LTOT08], for example, refactors Erlang programs using annotated syntax trees that retain binding and source information. JavaScript refactoring work relies on conservative static approximations when dynamic features cannot be resolved precisely [FMM+11]. C refactoring engines must preserve the relationship between source code and preprocessing directives [GJ05]. Refactorings share the goal of changing program structure without changing behavior, but the information that an engine needs to preserve behavior depends on the language.

A second line of work changed how refactorings were represented for reuse. Early engines exposed complete refactorings as the main reusable units. Later work decomposed them into conditional transformations, fine-grained steps, schemes, or scripts that could be combined to implement larger changes [KWK04, SK09, HKT16]. Scripting interfaces such as Wrangler's domain-specific language (DSL) [LT12a] extended this idea by allowing developers

to define project-specific sequences without modifying the engine itself. These approaches also broadened the meaning of composition. In the literature, a composite refactoring can mean an engine-level operation built from smaller operations, a scripted migration, or a sequence of refactorings later observed in a project history [BUA+23]. These meanings all concern sequenced changes, but they make different assumptions about representation, parameter flow, and correctness.

Research on refactoring engine correctness developed in parallel. Early engines relied on handwritten tests and regression suites. Later work added generated programs, property-based testing, large-scale application to existing projects, equivalence checks, static analysis, and studies of reported IDE defects [DDGM07, LT07, GBL+13, SHT24, OGRG25, WXZ+26]. This work made correctness a practical engineering concern as well as a specification problem. Section 2.6.5 discusses the testing literature in detail.

Refactoring is now a mature practice, but refactoring engines still raise open engineering problems. These problems concern how responsibilities are separated inside the engine. The engine must construct the code change, check the conditions that make it a refactoring, report warnings and choices, and expose a result that can be inspected. The same refactoring may need to run in an IDE, in a script, or inside a domain-specific refactoring. These uses show why the separation of those responsibilities is important. Chapters 3 and 6 address that separation.

2.3 Terminology and Correctness

This thesis uses *program transformation* as the broad term for an automated modification of a program. A transformation may rename an entity, replace an expression, create a declaration, or perform a larger migration. What a transformation must preserve depends on its purpose. Some transformations preserve only structural validity, while others are intended to preserve selected aspects of program behavior.

A *refactoring* is a program transformation that is expected to preserve behavior [Opd92, MT04]. Behavior preservation depends on which program observations count as behavior. For many refactorings, the relevant observations include outputs, exceptions, object state, name bindings, and dispatch targets. Other domains may also treat timing, resource use, or interaction with the environment as behavior [Rob99, MT04]. Behavior preservation is therefore a claim about selected observations, not about every possible property of the program.

A refactoring engine uses analyses and preconditions to decide whether a requested refactoring may proceed. After an engine applies a refactoring, the result can be inspected at several levels. These levels are useful to keep separate because satisfying one level does not establish the next:

Syntactic validity: The transformed source can be parsed according to the language grammar.

Static semantic well-formedness: The transformed program satisfies the language checks available before execution, such as name resolution, type checking, or ownership checking.

Successful execution: The transformed program can execute for a selected input without an engine or program failure.

Behavior preservation: The original and transformed programs produce equivalent observations within a stated scope.

A program may parse but fail name resolution. A statically well-formed program may raise an exception at runtime. A program may execute successfully for one input while differing for another input. The distinction is especially important for dynamically typed languages, where some binding and dispatch errors remain syntactically valid and are observed only during execution.

These levels also separate what an engine can decide from what it cannot. For a fixed target language and program representation, parsing and the implemented static well-formedness checks are decidable and can be enforced by the engine. Behavior preservation is different because it requires observational equivalence (the same observable behavior for every input). That equivalence is undecidable in general, and no static analysis can settle it for all inputs [FMM+11]. Testing does not settle it either. A finite suite can falsify preservation for the behavior it exercises but cannot prove it for behavior it does not exercise. Where a correctness proof exists, it holds for a formal model, not for the running engine, its compiler, and its runtime. Refactoring correctness is therefore not a property that can be fully established. What an engine can do is enforce decidable applicability preconditions. It can also check conservative sufficient preconditions for behavior preservation and warn wherever it cannot decide. Finally, it can gather empirical evidence against behavior change. This thesis takes that stance throughout: it separates the decidable applicability checks from the undecidable behavior-preserving ones (Section 3.1), and treats project tests as a partial, falsifying oracle rather than a proof of preservation (Chapter 6).

The following terms distinguish a requested refactoring from its outcome. A *refactoring attempt* is a request to apply a refactoring to a selected target with concrete parameters. The *configuration* of an attempt consists of the program and the parameters of the request. The selected target is one of these parameters. An attempt is *rejected* when an applicability precondition stops it before the transformation executes. It *stops after a warning* when a behavior-preserving precondition reports a warning and the attempt does not continue past it. A *refactoring application* occurs only when the engine applies the transformation and produces a changed program. An attempt *terminates* when transformation execution starts but does not produce a transformed program. It *completes* when execution produces a transformed program.

When a statement applies to both refactorings and behavior-agnostic transformations, this thesis uses *operation* as the umbrella term. Some operations are discussed as single steps, while others are built or observed as sequences.

Atomic operation: A change treated as one step by a particular engine or composition mechanism.

Composite refactoring: A change that coordinates several operations expected to preserve behavior together.

Migration or codemod: A coordinated change that does not require every intermediate or final step to preserve behavior.

These distinctions allow reusable transformations to be discussed without labeling every structural change as a refactoring.

A *precondition* is a condition that must hold before an operation proceeds. A *postcondition* states a property expected after the change. Early refactoring specifications used conditions to state when a program transformation should preserve behavior [Opd92, Rob99]. Implementations encode conditions as boolean checks, constraints, dependency comparisons, warnings, or queries over a semantic model. The encoding affects when the condition is evaluated and how its result is reported. The condition still serves the same purpose: deciding whether the engine rejects the request, warns about it, or applies it.

Preconditions differ by purpose. Following Anquetil *et al.* [ACD+22], we distinguish two kinds of preconditions:

Applicability precondition: A condition required to construct and apply the transformation. This category concerns syntactic and structural correctness: an applicability precondition ensures that the requested change can be constructed and is well-formed. The engine must construct code that can be parsed and, where a compiler is available, compiled without syntax errors or system exceptions such as duplicate instance-variable declarations or invalid selector syntax. Failure means that the engine cannot construct the requested change, and execution must stop immediately to prevent program-model corruption.

Behavior-preserving precondition: A condition intended to prevent an application that changes behavior within the selected observations. This category includes checks that go beyond constructibility and structural validity. For example, introducing a method that overrides an inherited one can alter dynamic dispatch. A check for such an override is a behavior-preserving precondition.

Applicability and behavior-preserving preconditions can lead to different engine responses. If an applicability precondition fails, the engine usually rejects the request because it cannot construct the requested change. If a behavior-preserving precondition fails, the transformation may still be constructible, but the engine must report the behavior risk. In an interactive client, that warning can be shown to the developer before continuing. In a script or batch client, the same warning must be exposed through the API so the caller can catch it and decide what to do.

Precondition defects appear in three forms.

Missing precondition: A missing check lets an invalid or unsafe (behavior-changing) configuration proceed without rejection or warning.

Overly conservative precondition: An overly conservative check rejects a configuration that the refactoring should allow.

Misplaced precondition: A misplaced check occurs too late, after the transformation has already started, or after the caller can no longer handle the warning cleanly.

These labels are relative to a particular refactoring, selected input, and expected engine response.

A central concern of the following chapters is that reusable transformations should retain the checks needed to construct a valid change, even when they do not include behavior-preserving checks. A refactoring adds checks concerned with preserving the selected behavior. The architectural consequence of this precondition partition, treating a refactoring as a transformation decorated with behavior-preserving checks, is developed as a model in Section 3.1.

2.4 Anatomy of a Refactoring Engine

Refactoring engines use different internal representations, but most automated applications can be described through a common pipeline. The pipeline provides the vocabulary for comparing systems later in this chapter. It is a conceptual model: engines need not expose the same classes or execute every stage in the same order.

An attempt begins with *target selection* and *parameter collection*. The target may be a declaration, a source interval, a reference, or a set of program entities. Parameters describe choices such as a new name, destination class, or method signature. Interactive clients collect these values through editor selections and dialogs. Scripts and batch clients construct the same information through an API. The engine input includes the source program, selected location, requested refactoring, parameters, and choices about warnings or confirmation.

The engine then queries a *program representation*. A syntax tree provides the grammatical structure needed to locate and rewrite program elements. Many refactorings also require semantic information such as bindings, inheritance, call targets, control flow, or data dependencies. Engines obtain this information through annotated trees, compiler services, graph models, relational representations, or dedicated program models. Roberts's analysis work [Rob99] showed why practical refactoring requires queries that extend beyond local syntax. The required queries depend on the requested refactoring. Rename needs binding information, Extract Method may need control-flow and data-flow information, and Move Method may need inheritance or dispatch information.

Precondition evaluation decides whether the requested operation may proceed. Some engines evaluate a fixed set of checks before constructing changes. Others derive conditions for a composition, solve constraints, or compare semantic information before and after a tentative transformation [KWK04, Ste18, Ove11]. The result may be acceptance, rejection, or a warning that delegates a decision to the caller. Automated clients need this outcome to be explicit, because they cannot respond to an embedded dialog.

During *change construction*, the engine *reifies* the modifications it intends to apply, that is, it represents them as explicit objects or data. An engine may modify a separate model of the program, construct a new program snapshot without altering the current one, or emit a set of text replacements. Reified changes can be inspected and coordinated before the live program

or source files are updated. They can also support preview and undo, although those properties depend on the surrounding workspace and change model rather than on reification alone.

Change construction also has a source-fidelity responsibility. A syntax tree may omit comments, whitespace, preprocessing directives, or exact token positions that are not needed for semantic analysis. Regenerating an entire file from such a tree can alter unrelated formatting. Engines address this problem through concrete syntax information, token-preserving trees, localized replacements, or source mappings [WY05, Ove13]. Source fidelity is separate from behavior preservation, but unexpected, unrelated edits reduce the usefulness of an otherwise correct refactoring.

Application commits the constructed changes to the development environment. In a file-based environment, this can mean updating documents in a workspace. In a live environment, it can mean compiling methods and changing active program objects. A transaction boundary defines which changes are committed together and which changes can be rolled back after failure or cancellation. Undo reverses a completed application using stored inverse changes, workspace history, or a previous program snapshot.

Failures can occur before or during each stage. A missing target can reject an attempt before analysis. An analysis can fail on an unsupported construct. Transformation logic can raise an exception or construct an invalid intermediate representation that parsing or compilation rejects. An application can fail after only part of a change reaches the environment. An engine architecture determines which failures can be represented, whether partial state can be restored, and what information is available to the caller. These properties later affect both user feedback and batch execution.

The same pipeline can be invoked by different clients. An IDE client gathers selections, displays warnings, requests additional choices, previews changes, and asks for confirmation. A script or batch client must provide those choices through an API. A batch client also needs to repeat attempts, isolate each attempt, record the outcome, and restore the project before the next attempt. Sharing analysis and transformation logic across these clients depends on whether those decisions are explicit or embedded in the refactoring implementation.

2.5 Dynamically Typed Object-Oriented Languages

Dynamically typed object-oriented languages complicate refactoring because some program relationships are determined only during execution. A message send may dispatch according to the runtime receiver, reflective code can construct or access program entities indirectly, and code loading can change the set of available classes and methods. Aliasing and mutable state introduce further effects that are not visible from a local syntax tree. These properties limit what an engine can establish from static source alone, but they do not make static analysis impossible.

Name binding illustrates the problem. Renaming a local variable can make an occurrence resolve to a different declaration, while renaming a method can change which implementation receives a message. In a statically typed language, compiler name and type checks expose some of these changes. In a dynamically typed language, the transformed source can remain valid

even when a runtime receiver selects a different method. The engine must use the bindings it can resolve and flag cases where name lookup or dispatch remains uncertain.

JavaScript refactoring research illustrates this limit. Feldthaus *et al.* [FMM+11] use static pointer analysis and refactoring-specific preconditions to establish safe changes for a supported subset of the language. When the analysis cannot prove a required property, the tool rejects the operation. This is a common response to uncertainty: a conservative approximation protects behavior preservation but can also reject valid applications. Support for dynamically typed languages involves a tradeoff between the programs an analysis accepts and the properties it can establish.

Different language environments expose different information. File-based Python and JavaScript tools commonly analyze source files and project configuration. Pharo [BDN+09, TDTP23] executes in a live image containing active classes, compiled methods, and ordinary objects that represent the running program. A refactoring can affect future compilation and active runtime state. Work on atomic refactoring in live environments addresses this additional state-preservation problem through dynamic software-update mechanisms [TPF+20]. Results obtained for one dynamically typed language cannot be transferred directly without accounting for these environmental differences.

Engines respond to incomplete static information in several ways. Optional or inferred types can rule out some invalid changes. Static analyses can establish selected binding, dispatch, or data-flow facts for supported language features. Runtime information can resolve facts about one loaded image. Warnings leave unresolved cases to the developer.

Reflection also limits static information, because the source does not show every use of a method or a class. A method name can be stored as data and sent later, a class can be looked up by a constructed symbol, and a framework can discover extensions after the analyzed package has been loaded. An engine can search literal reflective uses and known conventions, but it cannot generally enumerate values constructed by arbitrary execution. Practical tools restrict the supported cases, request confirmation, use runtime information, or accept that some dependencies remain outside the static model.

Dynamic features limit what an engine can establish before execution. The question is whether an engine can represent these responses separately from the refactoring logic, so that scripts and batch clients can use them as well.

2.6 State of the Art in Refactoring Engines

Existing refactoring infrastructures expose the responsibilities of an engine in different ways. This section compares research prototypes, production APIs, and the Pharo baseline through four questions:

- How does an engine represent and change programs?
- Which analyses and transformations can be reused?
- How are user decisions separated from execution?

- Which inputs and oracles are used to test the engine?

2.6.1 Engine Architecture and Program Representation

Program representation determines which questions a refactoring can ask and how precisely it can construct a change. A plain syntax tree supplies grammatical structure, but many operations also require bindings, types, inheritance, control flow, or source-token information. Engines package this information in different ways. This section compares which program information each engine makes available and how that information connects to condition checking and change construction.

The Smalltalk Refactoring Browser [Rob99] uses a dedicated program model over the language's compiler representation. This model gives refactorings a uniform view of classes and methods, while a separate change model records modifications before they are applied. Pharo kept this separation between the program model and the change model. Section 2.7 examines this implementation in detail. Here it serves as the historical example of a model separated from the live reflective API.

Eclipse's Language Toolkit (LTK) defines the refactoring lifecycle with initial conditions, additional parameters, final conditions, and construction of a Change object.[1] Refactorings in the Eclipse Java Development Tools (JDT) build on this lifecycle using Java model handles such as ICompilationUnit, IMethod, and IType. They parse compilation units into abstract syntax trees (ASTs), often with resolved bindings, use search infrastructure to find declarations and references, and construct edits through AST rewrites, import rewrites, and CompilationUnitChange objects. This combination gives JDT both a workspace-level model of Java elements and source-level rewrite machinery.

The IntelliJ Platform represents source code through the Program Structure Interface (PSI), a syntactic and semantic tree used by IDE features.[2] PSI references connect usages to declarations, and reference search supports features such as find usages and rename.[3] Code changes are performed by adding, replacing, or deleting PSI elements, often using newly parsed PSI fragments as replacements. These modifications must run inside the platform's write-action and command mechanisms.[4]

R3 [KBDA16] represents Java programs as relations over program entities. Refactorings query and update an in-memory database rather than repeatedly traversing source trees. After relational updates, R3 reconstructs the source program. This representation is intended to reduce repeated analysis work during refactoring and improve execution time.

Roslyn exposes immutable syntax trees, symbols, semantic models, compiler snapshots, and workspace models for C# and Visual Basic.[5] The compiler layer provides immutable syntax and semantic information for a compilation. The workspace layer organizes solutions, projects,

[1] https://help.eclipse.org/latest/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/ltk/core/refactoring/Refactoring.html
[2] https://plugins.jetbrains.com/docs/intellij/psi.html
[3] https://plugins.jetbrains.com/docs/intellij/psi-references.html
[4] https://plugins.jetbrains.com/docs/intellij/modifying-psi.html
[5] https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/compiler-api-model

and documents, and is the entry point for code analysis and refactoring over whole solutions. Refactoring tools analyze the current document or solution and construct a changed document or solution rather than mutating the existing syntax tree in place. This gives Roslyn a snapshot-based representation in which changes are expressed as new program states managed through the workspace model.

Erlang engines illustrate two other representations. Wrangler [LTOT08, LT11] augments syntax trees with binding, source-location, and variable information. RefactorErl [HLK+08, KCH+08] stores syntax and semantic relationships in a graph-oriented model queried by refactorings and analyses. The graph supports dependency queries across program entities, while Wrangler's annotations keep semantic information close to the syntax being rewritten.

Language constraints can require additional source and semantic information. Rust Extract Method must account for ownership and lifetime constraints that syntax alone does not capture [TCGS23]. C and C++ tools must retain relationships between the preprocessed program and the original source. Garrido's work [GJ05] incorporates conditional compilation into refactoring analysis, CScout [Spi10] uses token equivalence classes, and Proteus [WY05] emphasizes source fidelity. Clang represents refactorings as actions that choose among rules, check initiation requirements, and produce source replacements for command-line and editor clients.[6] These cases show that syntax-tree abstractions remain useful, but a refactoring engine also needs the source mappings and semantic information required by its language.

The established Python refactoring library is *rope*.[7] It builds a project-level model of modules, names, and inferred objects, and its refactorings construct a ChangeSet that the project applies as a unit with history-based undo. Section 5.6 reports a scoped proof of concept that realizes selected architectural roles in *rope*.

Table 2.1 summarizes the mechanisms described by the cited papers and public API documentation. A cell lists only the mechanisms that the cited source describes directly.

The systems in Table 2.1 separate the responsibilities of an engine in different ways. Eclipse separates condition checking from change construction, and Roslyn separates compiler snapshots from workspace updates. Clang distinguishes refactoring actions, rule requirements, and source replacements, while Wrangler adds a DSL for user-defined refactoring commands over its annotated syntax trees. These examples show that refactoring infrastructure can expose smaller units than a complete refactoring command. A source replacement, semantic query, change object, script command, or complete refactoring gives a client different information and different control.

Whichever representation is chosen, a refactoring engine needs access to the language information required by its refactorings, enough source information to construct acceptable edits, and an explicit representation of the changes it intends to apply. Once those pieces exist, the next question is which analyses and transformations can be exposed independently.

[6] https://clang.llvm.org/docs/RefactoringEngine.html
[7] https://github.com/python-rope/rope

Table 2.1: Documented mechanisms in representative refactoring infrastructures.

Infrastructure	Program representation	Condition/change separation	Composition or extension	Documented client model
Refactoring Browser [Rob99, dSS17]	Dedicated Smalltalk program model	Preconditions and change objects	Inheritance; composite execution; transformation hierarchy	Interactive browser; programmatic API
Eclipse LTK/JDT	Java model handles, JDT ASTs with bindings, search, and rewrites	Initial/final conditions and Change objects	Subclassed refactorings and JDT helper classes	IDE lifecycle; validation context can omit a UI shell
IntelliJ Platform	PSI tree with references, indexes, and mutable PSI elements	Write actions and commands over PSI changes	PSI mutation and language-extension APIs	IDE actions and plugin APIs
R3 [KBDA16]	In-memory relational representation	Refactoring workflow over relational updates	Relational schema and refactoring framework	Evaluated Java refactoring framework
Roslyn	Immutable syntax trees, semantic models, compilations, and workspace snapshots	Code actions construct changed documents or solutions	Compiler, workspace, analyzer, and code-action APIs	Visual Studio and other workspace hosts
Wrangler [LTOT08, LT12a]	Annotated Erlang syntax trees	Refactoring conditions and source updates	Refactoring DSL and API	Emacs/Eclipse integration and scripts
RefactorErl [HLK+08, KCH+08]	Syntax and semantic graph	Graph queries and transformations	Query and transformation interfaces	Interactive and programmatic clients
Clang	Clang AST plus source locations	Rule requirements and atomic source changes	Actions with alternative rules	Command line, editors, and IDEs
rope	Python project model with module ASTs, name and object inference	Refactoring classes construct a ChangeSet applied through project.do()	Refactoring classes; argument changers for signature changes	Library API; editor plugins

2.6.2 Reuse, Composition, and Extensibility

Reuse in a refactoring engine can happen at several levels. A client may reuse a semantic query, a precondition, a source transformation, a change object, a complete refactoring, or the coordination code around a refactoring. Treating all of these as "refactoring reuse" hides important differences. Reusing a binding query gives a client information about the program. Reusing a complete refactoring also brings the checks, parameter flow, intermediate states, and interaction behavior of that refactoring.

Class-based refactoring engines make inheritance an immediate reuse mechanism. A base refactoring class can provide program-model access, shared precondition checks, and shared transformation steps. This reduces duplication when subclasses follow the same execution structure. It becomes restrictive when one subclass needs only one step or a different order, because the reusable code is tied to the inherited workflow. The Pharo baseline contains this pattern and is examined in Section 2.7.3.

Another reuse strategy treats existing refactorings as building blocks. The reused unit is a complete refactoring. A larger refactoring invokes existing refactorings inside its own logic and coordinates their parameters, intermediate states, and failures. Work on design-pattern transformations used smaller refactorings as programmatic building blocks [OCN99, OC00], which made the composite logic explicit. If each component applies changes immediately, a later failure also requires a recovery decision.

Static composition describes a larger change before execution. Kniesel and Koch [KWK04] use transformation descriptions to derive preconditions for statically composed refactorings. Ajouli [AC11] and Cohen [CA13] apply static composition to design-pattern transformations and remodularization examples. This work asks how the requirements of the smaller transformations determine the requirements of the larger change.

Other work decomposes refactorings into smaller units. Schäfer *et al.* [SVEdM09, SdM10] describe micro-refactorings used to implement Extract Method. Saadeh *et al.* [SK09] use fine-grained transformations as steps in composite refactorings. Horpácsi *et al.* [HKT16, HKH17] define refactoring schemes from verifiable prime refactorings. These mechanisms differ in formal basis and execution model. They share the observation that a complete refactoring can contain smaller operations worth representing explicitly.

The literature uses *composite refactoring* for different kinds of sequences, including mined histories, prescribed sequences, scripts, and engine-level commands [BUA+23]. This thesis uses *engine-level composition* for the case where a larger change must define parameter flow, conditions, intermediate validity, and failure handling.

Scripting moves the extension point to users and external tools. It lets users and tools define project-specific sequences without changing the engine. Wrangler's DSL [LT12a, LT12b] supports user-defined refactoring commands over its program representation. JunGL [VEdM06] provides a .NET refactoring language, while the Rascal/Eclipse integration [HKV12] uses a meta-programming language to express larger transformations. These systems provide expressive extension points, but the script author remains responsible for the conditions and interaction decisions not supplied by the scripting interface.

Libraries such as Spoon [PMP+16], Jrbx [MY05], Overbey's toolkit [Ove11], and the Pharo

Rewrite Tool [RBD15] expose lower-level transformation mechanisms. This access is valuable for migrations and domain-specific changes that do not fit a fixed catalog. It also leaves more behavior-preservation work to the extension author. The library can support structural validity, but behavior preservation requires additional analyses or checks.

Domain-specific refactorings show this tradeoff between low-level access and behavior-preservation work. A framework migration may need to replace a call pattern, add an adapter, and update configuration under assumptions known only to that framework. A strict general-purpose refactoring may reject the intermediate states, while raw tree rewriting duplicates the analysis already present in the engine. Reusable transformations allow the domain-specific operation to share structural mechanisms and add its own conditions.

Extract Method shows why reuse sometimes needs a smaller unit than a complete refactoring. The engine must validate the selected region, analyze variables and control flow, construct the new method, and replace the selected code with a call to the new method [AMO24, SDPR24]. It must also preserve name bindings and dispatch after the replacement. Some of these analyses depend on the language. For example, a Smalltalk block can contain a non-local return from its enclosing method [SDPR24]. A domain-specific variant of Extract Method may retain these analyses while changing the generated method or its destination. Section 5.3 presents two such variants, the Extract SetUp refactoring and the Extract with Pragma refactoring. A coarse-grained implementation forces the variant to subclass, copy, or bypass the full operation. An engine with explicit transformation units lets the variant reuse only the steps it needs.

Prior work provides several composition mechanisms, but no single mechanism covers all reuse needs. Inheritance shares implementation, command-level composition shares complete refactorings, static composition reasons about larger changes before execution, scripting exposes user-defined sequences, and rewrite libraries expose lower-level program changes. RQ1 concerns how transformations and refactorings can share a compatible architecture. RQ2 concerns how the decisions required during their execution can be supplied by different clients.

2.6.3 Interaction and Scriptability

The original Refactoring Browser [RBJ97] established an interaction pattern still used by IDEs: select a program element, choose a refactoring, provide parameters, inspect warnings or a preview, and apply the change. This workflow suits a developer who applies one refactoring at a time and inspects its result, but it is not the only way to drive a refactoring engine.

Developer studies show that automated refactorings are often interleaved with manual edits, a practice described as floss refactoring [MHPB09]. Tools such as WitchDoctor [FGL12] and BeneFactor [GDMH12] detect a manual refactoring in progress and offer to complete it. They move the initiation from an explicit menu command to recognition of an edit pattern. They still rely on engine services for analysis and transformation.

Other interfaces reduce reliance on menus and dialogs. Visual transformations [BGH07], drag-and-drop refactoring [LCJ13], and structural multi-cursor editing [VRS22] let developers manipulate program entities through more direct interactions. Vakilian *et al.* [VCN+12,

VCM+13] make a related argument for complex refactorings: a tool can expose smaller, predictable steps instead of one large wizard. Their studies are relevant here because they connect refactoring decomposition with developer feedback.

Scripts and batch clients have different requirements. They must provide parameters and decide what to do with warnings without UI prompts. Wrangler scripts [LT12a] and Rascal transformations [HKV12] encode those choices in code. Batch clients make similar choices for repeated invocations and must restore state between attempts. In each case, embedded prompts or UI-specific exceptions prevent the core refactoring from being reused directly.

The engine must therefore keep the refactoring logic separate from the decisions required during its execution, such as parameter values and responses to warnings. To do so, it reports parameter requests, warnings, and constructed changes through an API. An IDE client can turn these results into dialogs, previews, and confirmations. A headless client must respond to the same results with preselected choices, or reject the attempt. This separation allows the same analysis and transformation logic to support several interaction models.

2.6.4 Specification and Preconditions

A refactoring definition must describe more than the source change to perform. It must also state the conditions under which the change is expected to preserve behavior. Opdyke [Opd92] and Roberts [Rob99] expressed these requirements through preconditions and postconditions. Kniesel and Koch [KWK04] studied how such conditions change when conditional transformations are composed. This line of work treats the required conditions as part of the refactoring definition, not as incidental checks inside one tool.

Other approaches describe the required conditions in terms of program relationships rather than as individual boolean checks. Schäfer *et al.* [SdM10, STST12] specify expected preservation of bindings, control flow, and data flow. Steimann's constraint-based approach [SvP12, Ste18] represents refactoring as constraint repair. Differential precondition checking [Ove11] tentatively applies a change and compares semantic models to determine whether the observed differences match those expected for the refactoring. These approaches differ in form, but each depends on the program properties available in its model.

Some work defines a refactoring independently of a particular language. Refactoring based on the FAMIX metamodel [Tic01], cross-language metamodels [MSL12], conditional term-rewrite schemes [SKL06, HKN21], and synthesis-based approaches [Kes18] do so at different levels of abstraction. These frameworks can only reason about the semantics and language constructs they represent.

A concrete engine still has to realize the refactoring through its program model, reusable components, and interaction API. Defects can enter through analysis, condition checking, change construction, warning handling, or failure handling. Roberts observed that defects persisted in his engine despite its formal specification [Rob99], so implementation testing is not subsumed by specification. Checking engines is therefore a separate concern from defining refactorings.

2.6.5 Reliability and Automated Testing of Refactoring Engines

Testing a refactoring engine involves two independent decisions. The input space determines which programs, targets, parameters, and refactorings are exercised. The outcome checks determine what evidence is collected after an attempt, such as compilation errors, engine exceptions, test failures, semantic differences, or inspection findings. Studies combine these decisions in different ways: generated programs with a compiler oracle, existing projects with failure inspection, or generated inputs with behavior comparisons.

Handwritten and Generated Inputs. Engine developers commonly write regression tests for known refactorings and defects. A test can specify the input program, selected refactoring, parameters, expected rejection or change, and expected output. Such tests are precise and easy to inspect, but they cover only anticipated examples. The number of possible programs, targets, and parameter combinations makes exhaustive handwritten testing infeasible for general-purpose engines.

Generated-program testing expands the input space with synthetic programs produced by a generator. Daniel *et al.* introduced ASTGen [DDGM07], which generates structurally varied Java abstract syntax trees and tests Eclipse and NetBeans refactorings with several automated oracles. JDolly uses bounded Alloy specifications to generate Java programs [SGM13]. SafeRefactor [SGSM10, SGM13] generates tests that compare common behavior before and after a transformation. Generated inputs are useful because the tester controls which language features, program shapes, and program sizes are produced. That control is also the main limitation. A language feature, library pattern, or semantic dependency absent from the generator is also absent from the generated tests. This limitation has been examined directly for a single well-studied operation: automated test-case generation detected only 43% of injected Extract Method faults in one study [SAM18].

Later work asked how to make generated testing cheaper and how to use it for more specific defect classes. Jagannath *et al.* [JLDM09] reduce bounded-exhaustive testing cost by clustering or pruning redundant inputs. Mongiovi *et al.* [MGS+14, MMG+14, Mon16] use impact analysis, skip parameters, and multiple oracles to reduce test execution while preserving useful checks. Their work on overly strong preconditions [SMG11, MGS+18] disables selected checks and uses SafeRefactor to test the resulting transformation. If the transformation without that precondition passes all checks, the disabled precondition may be overly strong. The result is still limited by the generated program and by the behavior checked by SafeRefactor.

Generated testing was also explored through property-based testing for Erlang. Drienyovszky *et al.* [DHT10] generate Erlang programs from an attributed grammar and compare executions of selected functions before and after refactoring.

Testing on Real Projects. Testing on real projects uses maintained software as the input source. This avoids building a generator for the complete language and includes dependencies, idioms, and semantic coupling already present in the project. With real projects as input, some approaches generate the refactoring commands instead of the programs. Li and Thompson [LT07] generate refactoring commands over existing Erlang programs with QuickCheck

and check properties such as compilation and binding information. Spinellis [Spi10] applied identifier renaming across the Linux kernel, while Thies and Steimann [TS10] tested Eclipse refactorings over existing code. Gligoric *et al.* [GBL+13] combined large-scale application, failure clustering, and inspection over Java and C projects. They reported 77 Java and 43 C engine bugs. Their automated failure signals were compilation errors and engine exceptions.

Real projects do not solve input selection by themselves. The tester must still choose projects, locations, refactoring types, and parameters. Common constructs may be exercised repeatedly, while rare language features or library patterns remain absent. Generated and real-project inputs are therefore complementary: generators target selected program shapes, while projects contribute naturally occurring combinations that the tester did not plan.

Outcome Checks and Oracles. Table 2.2 summarizes the principal outcome checks. For each check, it lists what the check can expose and what it does not establish by itself. One refactoring attempt can produce several of these results, such as a parse result, an engine exception, and a project-test result.

Table 2.2: Outcome checks used in refactoring engine testing.

Check	Can expose	Does not establish by itself
Parsing	Source or tree-serialization errors	Name, type, binding, or runtime correctness
Compilation or static checks	Language well-formedness, type, ownership, or selected analysis errors	Equivalent runtime behavior
Engine differential	Disagreement between independent implementations	Which implementation is correct without further evidence
Generated tests	Behavior differences exercised by generated inputs and assertions	Behavior outside the generated test space
Project tests	Behavior differences exercised by existing project assertions	Equivalence for untested behavior
Equivalence checking	Difference under the modeled semantics and generated inputs	Properties outside the model or checked input domain
Exception or termination	Engine failure during analysis or transformation	A behavior change in an applied refactoring
Structural inspection	Invalid ASTs, unexpected edits, or broken source fidelity	General behavior preservation
Replay and manual inspection	Reproduction and contextual classification of an anomaly	Exhaustive absence of other defects

Compiler-based checks are effective when a transformation violates a statically checked property. A compiling program can still behave differently. Project tests provide a practical semantic oracle for the behavior they exercise. For Extract Method, one study found that

manually written suites detected more injected faults (61.9%) than automatically generated suites (Randoop 46.7%, EvoSuite 55.8%) [GCA22]. A failing test after an application is evidence of an observed behavior change. Exceptions and terminated executions provide additional signals, while replay supports the subsequent diagnosis of an observed anomaly. Those signals should be classified separately from project-test failures.

Static and equivalence-based oracles add language-specific checks beyond parsing and compilation. Oliveira *et al.* [OGRG25] detect type errors introduced by Python refactoring engines through static analysis. Seres *et al.* [SHT24] compare affected Erlang functions using generated inputs. These methods can detect failures that parsing alone misses, but their conclusions remain limited by the analysis or equivalence model.

Metamorphic Relations and Parameter Exploration. Metamorphic testing [CCY98, CKL+18,SFSRC16] is useful when the correct output for a given test input is difficult to specify. It checks relations between original and follow-up executions rather than requiring a complete expected output for each case. For refactoring, the relation can connect the project before and after an application. The relation still requires an observable outcome, so metamorphic testing does not remove the oracle problem.

Parameter exploration treats the refactoring configuration as part of the input space. Valid parameters exercise cases where the engine applies the refactoring. Invalid, boundary, and context-dependent parameters exercise parsing, applicability checks, warnings, and rejection paths. Forced execution can examine an attempt that normal preconditions rejected. This is related to Disabling Preconditions [MGS+18] because both examine rejected attempts and overly conservative checks, but the input source and execution control differ.

Recent Empirical and Bug-Report-Based Work. Wang *et al.* [WXZ+26] analyzed 518 reports from Eclipse, IntelliJ IDEA, and NetBeans and classified symptoms, root causes, and triggering conditions. Their taxonomy includes incorrect transformations, precondition checks, flow analysis, and type resolution. The study reports where defects appeared across refactoring types.

RETESTER [WXT25] uses historical bug reports to guide input generation. It extracts bug-related templates and characteristics, uses a large language model to generate variants, and compares engine outcomes. Gheyi *et al.* [GRO25] instead prompt small language models with a program before and after a refactoring to flag behavior differences. They detect up to 84.3% of one bug class in a 100-bug Java and Python benchmark. In both approaches, the outcomes still have to be compared and inspected.

The reviewed approaches therefore provide three dimensions for the testing method developed later: input selection, execution control, and outcome evidence.

2.7 The Pharo Legacy Engine

2.7.1 Architectural Overview

The Pharo legacy engine is the baseline for the implementation of the architecture in this thesis. The discussion below uses the implementation analyzed in prior architecture work

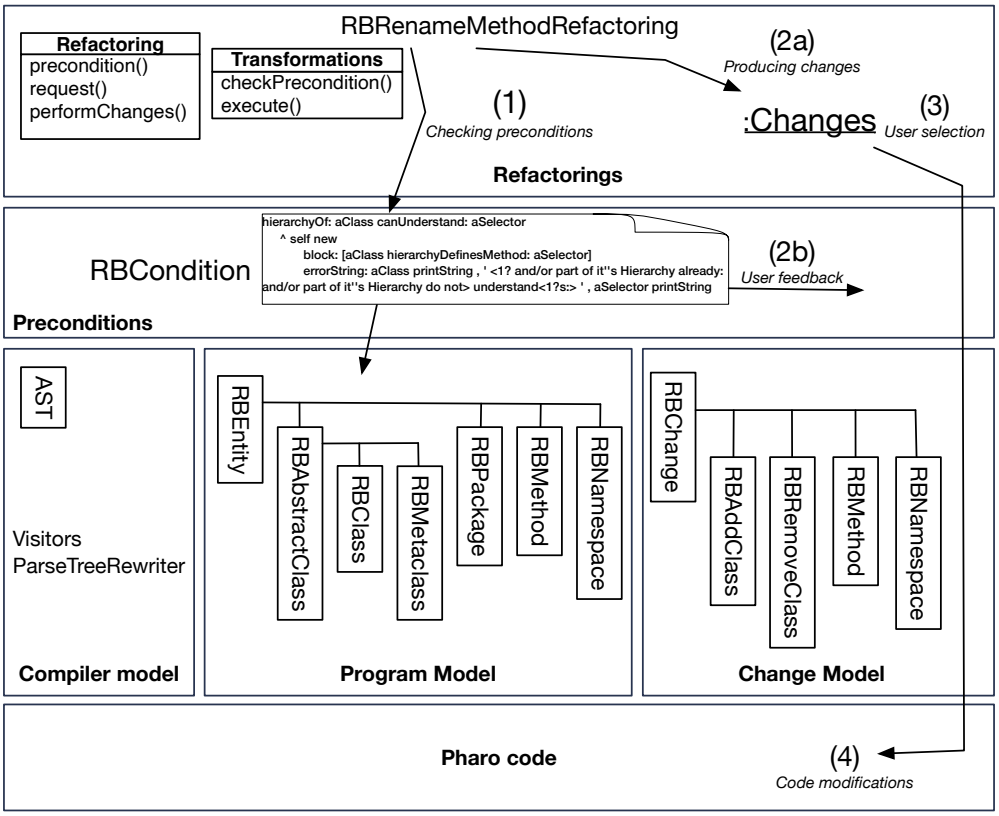

Figure 2.1: Typical legacy-engine execution flow. A refactoring checks preconditions through RBCondition (Step 1). It either reports a failed precondition (Step 2b) or constructs change objects (Step 2a). The developer can inspect the changes (Step 3) before they are applied to the live system (Step 4).

[SADT24]. It identifies the limitations in precondition handling and reuse that the reference architecture in Chapter 3 addresses.

The engine descends from the Smalltalk Refactoring Browser [RBJO96, RBJ97, Rob99]. Over time, Pharo maintenance added operations and changed existing conditions. A separate RBTransformation hierarchy [dSS17] was later introduced along with declarative composition. The result is two hierarchies and several reuse styles.

Pharo [TDTP23] is a live environment in which classes and compiled methods are active objects. The engine does not analyze these live objects directly, but works on separate models of the program. Its *compiler model* provides syntax trees, visitors, and a parse-tree rewriter. The *program model* wraps syntax and program entities in objects such as RBMethod and RBClass. This separation allows an operation to query and modify a model before changing the corresponding live objects.

The *change model* reifies modifications such as adding, removing, or renaming classes,

methods, and variables. Refactorings construct these objects instead of applying each source edit immediately. This model provides an application boundary: a caller can inspect the intended changes before committing them to the image. This boundary supports preview and cancellation.

The *precondition library* provides checks through RBCondition. Refactoring classes query the program model, combine conditions, and construct change objects when conditions are satisfied. Conditions can also signal errors or warnings that interact directly with the user, so an operation is not always independent of the interactive client.

Figure 2.1 summarizes the usual flow. The refactoring queries the program model and the precondition library. A failed precondition can report an error or a warning. A successful path constructs change objects, which can be previewed and selected before application. This selection matters in a dynamic environment, where the engine may propose changes that require developer judgment. Preview lets the developer inspect the proposed change set, keep the relevant parts, and gain confidence before applying it.

The legacy engine separates analysis from the reflective runtime, represents changes explicitly, and preserves source layout through its compiler model and rewriter. The limitations studied below concern two other parts of the engine, its preconditions and its reuse mechanisms.

2.7.2 The Challenge of Preconditions

The precondition challenge concerns both which properties are checked and how a failed precondition is represented for the caller.

Preconditions are expressed in one of three ways:

- As simple (low-level) conditions implemented as class-side (static) methods of the RBCondition class (Listing 2.1);
- As a composition of these simple conditions using two aggregator classes: RBConjunctiveCondition and RBNegationCondition (Listing 2.2);
- As manual checks implemented directly via the program-model API (Listing 2.3).

Listing 2.1 illustrates an isolated check where the RBCondition is sufficient. The preconditions method checks that a class defines a variable (lines 3 and 4) before creating its accessors. It calls two methods from RBCondition: definesClassVariable:in: and definesInstanceVariable:in:. For a single check, a boolean result and a message are usually enough. The caller only needs to know that the selected class does not define the requested variable.

Listing 2.2 shows a composite condition for removing a set of classes. The refactoring aggregates several checks for each target class.

The boolean composition uses the & operator. The resulting condition can report a combined message, but it does not expose which sub-check failed for which target. For example, one observed Rename Method failure reports: *"The arguments are not permuted AND RBAddMessageSendTransformation and/or part of its Hierarchy already understands*

```
RBCreateAccessorsForVariableRefactoring >> preconditions
    ^ classVariable
        ifTrue: [ RBCondition definesClassVariable: variableName asSymbol in: class ]
        ifFalse: [ RBCondition definesInstanceVariable: variableName in: class ]
```

Listing 2.1: A basic case of refactorings using methods from RBCondition.

```
RBRemoveClassRefactoring >> preconditions
    ^ classNames inject: self emptyCondition into:[ :sum :each |
     | aClassOrTrait |
     aClassOrTrait := self model classNamed: each asSymbol.
     aClassOrTrait ifNil: [
        self refactoringFailure: 'No such class or trait' ].
     sum & ((self preconditionIsNotMetaclass: aClassOrTrait)
         & (self preconditionHasNoReferences: each)
         & (self preconditionEmptyOrHasNoSubclasses: aClassOrTrait)
         & (self preconditionHasNoUsers: aClassOrTrait)) ]
```

Listing 2.2: Remove Class preconditions.

#preconditions". The caller receives a combined error string rather than structured information for classification or recovery.

Listing 2.3 shows the alternative used by Pull Up Instance Variable: a block queries the program model and signals failures and warnings directly.

```
RBPullUpInstanceVariableRefactoring >> preconditions
    ^ RBCondition withBlock: [ (class hierarchyDefinesInstanceVariable: variableName)
        ifFalse: [ self refactoringFailure: 'No subclass defines ' , variableName ].
    (class subclasses
        anySatisfy: [ :each | (each directlyDefinesInstanceVariable: variableName) not
    ])
        ifTrue: [ self
            refactoringWarning: 'Not all subclasses have an instance variable named.<n>
            Do you want to pull up this variable anyway?' , variableName , '.' ].
    true ]
```

Listing 2.3: Pull Up Instance Variable refactoring preconditions checking that variable exists in all the subclasses.

Manual checks are more flexible because they can use local program-model queries and produce operation-specific messages. Here, that flexibility comes with coupling: the condition, warning, and continuation decision are embedded in one refactoring method. That is acceptable for a specialized check, but it does not scale as a general reuse mechanism. To reuse the check, another refactoring, a transformation, or a headless client must inherit, extract, or duplicate its logic.

Applicability and behavior-preserving checks are mixed within individual refactoring definitions. Both kinds are built from the same generic RBCondition composition mechanism. Failure and warning signaling is also scattered through the refactoring implementation. The legacy hierarchy exposes three related signaling entry points: refactoringFailure:, its successor refactoringError:, and refactoringWarning:. The warning path delegates interactive presentation and continuation to a UIManager. Scripts may catch all three signals. The architectural problem is where the signals are raised: any legacy precondition block or transformation method can signal a failure or a warning directly. Every such method introduces another interaction or signaling dependency and another site that must be found and migrated.

To support headless rewriting, the legacy engine introduced a separate RBTransformation hierarchy that allowed developers to disable precondition checking entirely. Because the legacy engine did not separate applicability checks from behavior-preserving checks, this bypass was an all-or-nothing tradeoff. Disabling behavior-preserving warnings also disabled the structural applicability checks, so an invalid structural operation could reach application and then fail there or in the compiler instead of being rejected by an applicability check.

2.7.3 The Challenge of Reuse

Reuse appears in the legacy engine at several levels. The legacy implementation offers three mechanisms: class inheritance, imperative delegation, and declarative composition. They reuse different units: a shared workflow, a complete refactoring, or separate transformations.

Approach 1: Reuse via Inheritance. The first approach is class inheritance. This section examines it through the family of refactorings that change a method name. This family includes adding parameters, removing parameters, renaming a method, and replacing message sends. In Pharo, adding or removing parameters changes the method selector, so these operations share name-change logic.

Listing 2.4 shows the transform method of the abstract superclass RBChangeMethodNameRefactoring, which dictates a strict three-step execution flow these refactorings share:

- rename the implementors with the new name (line 2);
- replace all the old references to the method (line 3);
- remove the old selector (line 4).

```
1 RBChangeMethodNameRefactoring >> transform
2     self renameImplementors.
3     self renameMessageSends.
4     self removeRenamedImplementors
```

Listing 2.4: RBChangeMethodNameRefactoring dictates a strict execution flow reused by subclasses like RBRenameMethodRefactoring.

The class RBReplaceMessageSendTransformation implements a related action: it replaces the invocations of a method without changing the invoked method itself. This action is the second step of the three-step workflow. This class inherits from the same abstract superclass, but because it only needs the second step of the flow, it completely overrides the transform method (see Listing 2.5).

```
1 RBReplaceMessageSendTransformation >> transform
2     self replaceInAllClasses
3         ifTrue:[ self renameMessageSends ]
4         ifFalse:[ self renameMessageSendsIn: {class} ]
```

Listing 2.5: The class RBReplaceMessageSendTransformation overrides the inherited method to skip unwanted steps.

Inheritance reuses a workflow. This form of reuse suits subclasses that need all the steps of the workflow. A subclass that needs only one step must override the inherited method to skip the others. The reusable message-send replacement is therefore tied to a superclass workflow even though the subclass bypasses most of that workflow.

Approach 2: Imperative Delegation. Imperative delegation reuses a complete refactoring object. Listing 2.6 shows RBPullUpMethodRefactoring instantiating and executing RBPullUpInstanceVariableRefactoring for variables required by the target method.

```
1 RBPullUpMethodRefactoring >> pushUpVariable: aVariable
2     | refactoring |
3     refactoring := RBPullUpInstanceVariableRefactoring
4         model: self model
5         variable: aVariable
6         class: targetSuperclass.
7     self performCompositeRefactoring: refactoring
```

Listing 2.6: RBPullUpMethodRefactoring delegating logic by explicitly executing RBPullUpInstanceVariableRefactoring.

The reuse is explicit through performCompositeRefactoring:, which transfers configuration to the delegated refactoring. This avoids forcing both operations into one inheritance branch. The delegated unit, however, is a complete refactoring rather than a transformation step. A complete refactoring includes applicability checks, behavior-preserving checks, and the warning or failure behavior implemented by that refactoring. Delegation is appropriate when the composite needs the same checks and the same warning or failure behavior. However, the composite cannot use this delegation mechanism to reuse the structural change while varying the behavior-preserving checks around it. The composite also depends on how the delegated refactoring reports warnings and failures. If different refactorings handle interaction differently, a composite or batch client cannot handle warnings and failures consistently without additional adaptation.

Approach 3: Declarative Composition. Declarative composition is found entirely outside the refactoring hierarchy, within the separate RBTransformation hierarchy [dSS17]. Declarative composition reuses transformations as explicit steps. Instead of executing steps imperatively, a transformation returns a list of configured transformations that the engine will then execute.

For example, Listing 2.7 shows the buildTransformations method of RBMoveInstance-VariableTransformation. It achieves its goal by returning a collection of two reusable transformations: one to add the variable to the new class, and one to remove it from the old class.

```
1  RBMoveInstanceVariableTransformation >> buildTransformations
2      ^ OrderedCollection
3        with: (RBAddVariableTransformation
4           model: self model
5           instanceVariable: variableName asString
6           class: className)
7        with: (RBRemoveVariableTransformation
8           model: self model
9           instanceVariable: variableName asString
10          class: oldClass)
```

Listing 2.7: The buildTransformations method uses declarative composition to reuse existing transformations.

This gives composition an explicit representation before execution, but only inside the transformation hierarchy. Refactorings cannot appear in the same list because they do not share the same construction and execution API.

Each of these mechanisms fixes a different unit of reuse. Inheritance shares a workflow, which is restrictive when a subclass needs only one step. Delegation calls a complete refactoring. That is useful when the composite should include the delegated refactoring's checks, but the composite also inherits its warning and failure behavior. Declarative composition exposes transformations, but only inside the transformation hierarchy. The architectural problem is that the engine lacks a shared interface where applicability checks, behavior-preserving checks, and transformation steps can be reused separately when needed.

2.8 Synthesis and Research Gap

The reviewed systems already provide substantial refactoring infrastructure: syntax and semantic models, explicit change representations, previews, undo, scripting interfaces, and a range of testing inputs and oracles. Java research advanced engine architecture and specification through relational representations [KBDA16], decomposition [SVEdM09, SdM10], and dependency-based descriptions [Sch10], as well as generated-program [DDGM07], behavioral [SGM13], and precondition testing [MGS+18]. Erlang work, especially RefactorErl [HLK+08, KCH+08, BHH+11] and Wrangler [LT12a], shows how a dynamically typed *functional* engine

can address the same problems through semantic program models, scripting, transformation support, property-based testing [LT07, DHT10], and decomposition-based validation [HKT16, HKH17]. This thesis explores how a dynamically typed *object-oriented* engine can provide comparable support. Such an engine must also handle inheritance with overriding and dispatch based on the receiver's class, two sources of behavior change that functional engines do not face.

The reviewed dynamically typed object-oriented engines lack an intermediate level between complete behavior-preserving refactorings and unchecked structural rewrites. This absence is the architectural gap addressed here. A client may need the same structural change as a refactoring or as a behavior-agnostic transformation. However, the reviewed mechanisms do not consistently let clients bypass a conservative behavior-preserving check while retaining the structural applicability checks. The goal is to supply a checked structural change next to the refactoring, without weakening the refactoring checks. A conservative or irrelevant behavior-preserving check then does not force developers to make manual edits or to use unchecked rewriting.

The reviewed systems expose different units of reuse: complete refactoring commands, scripts, verified transformation schemes, or lower-level edits. None of them supplies a shared interface through which refactorings and transformations can reuse the program model, change construction, applicability checks, and behavior-preserving checks separately.

Pharo is the baseline used to study this gap, and Section 2.7 showed why. It contains refactorings and transformations in incompatible hierarchies with different construction and execution APIs. Its reuse is split across three mechanisms that each fix a different unit of reuse, and some conditions interact with the user while they are evaluated. These limitations do not concern its program and change models. The models should be retained because they already separate analysis from the live reflective API, preserve source information, and defer application until changes can be inspected.

The rest of this section states one baseline constraint and four requirements that follow from the review.

Baseline constraint: Program and change representation. A dynamically typed object-oriented refactoring engine needs a program representation that supports semantic queries, precondition checks, and source-preserving changes. It also needs an explicit change representation so that constructed edits can be inspected before they are applied.

The Pharo baseline already satisfies this constraint, so its program and change models remain part of the proposed architecture rather than being replaced.

Requirement 1: Refactorings and transformations. An engine should provide checked transformations as reusable units distinct from behavior-preserving refactorings. A transformation should keep the applicability checks needed to construct and apply the structural change. A refactoring should reuse that transformation and include behavior-preserving checks when behavior preservation is required. Both should share program models, change construction, and relevant analyses.

This is the architecture question behind RQ1: how to structure transformations and refactorings so that implementation logic can be reused without copying a complete behavior-preserving workflow. Anquetil *et al.* [ACD+22] provide the conceptual starting point for this question by arguing that engines need both refactorings and transformations, and by distinguishing applicability preconditions from behavior-preserving preconditions. The remaining question is how to realize that distinction across reusable transformation logic, composite operations, and clients with different interaction requirements. Chapters 3 through 5 define, implement, and evaluate an architecture that answers it.

Requirement 2: Explicit interaction orchestration. Parameter requests, warnings, confirmations, previews, and application decisions need to be represented separately from analysis, precondition checking, and transformation logic. Interactive, script, and batch clients should be able to supply parameters, handle warnings, confirm or reject changes, and decide how failures are reported.

This is the interaction question behind RQ2: how to run the same transformation or refactoring through IDE, script, and batch clients without embedding one interaction model in the implementation.

The testing literature provides program generators, real-project inputs, and outcome checks but does not connect them to the engine's precondition responses. A testing method for this setting must connect real-project targets and parameter variation to explicit precondition responses and execution outcomes, and record the evidence used to classify each anomaly.

Requirement 3: Real-project testing and outcome checks. Testing should include real projects so the engine is exercised on existing dependencies, idioms, and test suites, not only generated language fragments. Reported outcomes should state which check produced them because compiler errors, engine exceptions, and project-test failures support different conclusions. For dynamically typed languages, project tests can check the behavior they exercise, but passing tests do not prove that the application was correct.

RQ3 evaluates this requirement over real projects and their existing tests.

Requirement 4: Precondition validation. Testing needs a way to exercise configurations that expose missing or overly conservative preconditions without interactive intervention and to retain their outcomes for triage. It must distinguish rejected attempts from terminated executions and completed applications. Forced execution is needed when evaluating whether a rejection was overly conservative.

This is the precondition question behind RQ4, which examines missing and overly conservative preconditions. It follows from Disabling Preconditions and metamorphic-testing work, but uses the execution controls and semantic oracle available in Pharo.

Requirements 1 and 2 define the architectural gap, while Requirements 3 and 4 define what testing must exercise. All four retain the baseline program and change representation rather than redesigning it. The next chapters address them in that order.

Chapter 3

Reference Architecture Design

This chapter presents a reference architecture for refactoring engines in dynamically typed object-oriented languages. The architecture addresses the limitations identified in Section 2.8 by answering two research questions, RQ1 and RQ2: how refactorings and transformations can share implementation logic, and how user interaction can be separated from that logic, so that the same refactoring core supports both interactive IDE use and headless scripting.

The architecture is described through four roles and four contracts. A transformation is a source change guarded by applicability preconditions, and a refactoring is a transformation decorated with behavior-preserving preconditions. Preconditions are reified as objects that record what caused a failure. Interaction drivers gather parameters, resolve warnings, and preview changes, so the core operation stays callable without UI prompts. A contract is a condition that an implementation either satisfies or violates. Section 3.1 states the transformation, refactoring, and composition contracts, and Section 3.3 the interaction contract.

The architecture assumes that the target engine already satisfies the baseline constraint identified in Chapter 2. That constraint requires a program model that supports semantic queries and precondition checks, and an explicit change model that can be inspected before changes are applied. The architecture treats these two models as required engine capabilities and builds upon them to address the four requirements from Chapter 2. Section 3.4 records where each requirement is addressed.

Figure 3.1 gives an overview of the architecture, and Chapter 4 provides one realization of it in Pharo. The following sections describe its components and how each supports the migration of an existing engine.

3.1 A Model of Transformations and Refactorings

Before describing the components, we present the model that the architecture realizes. The model defines the roles, their domains, and the contracts they must satisfy. The rest of the chapter, the Pharo implementation in Chapter 4, and the testing method in Chapter 6 can then all refer to a consistent vocabulary.

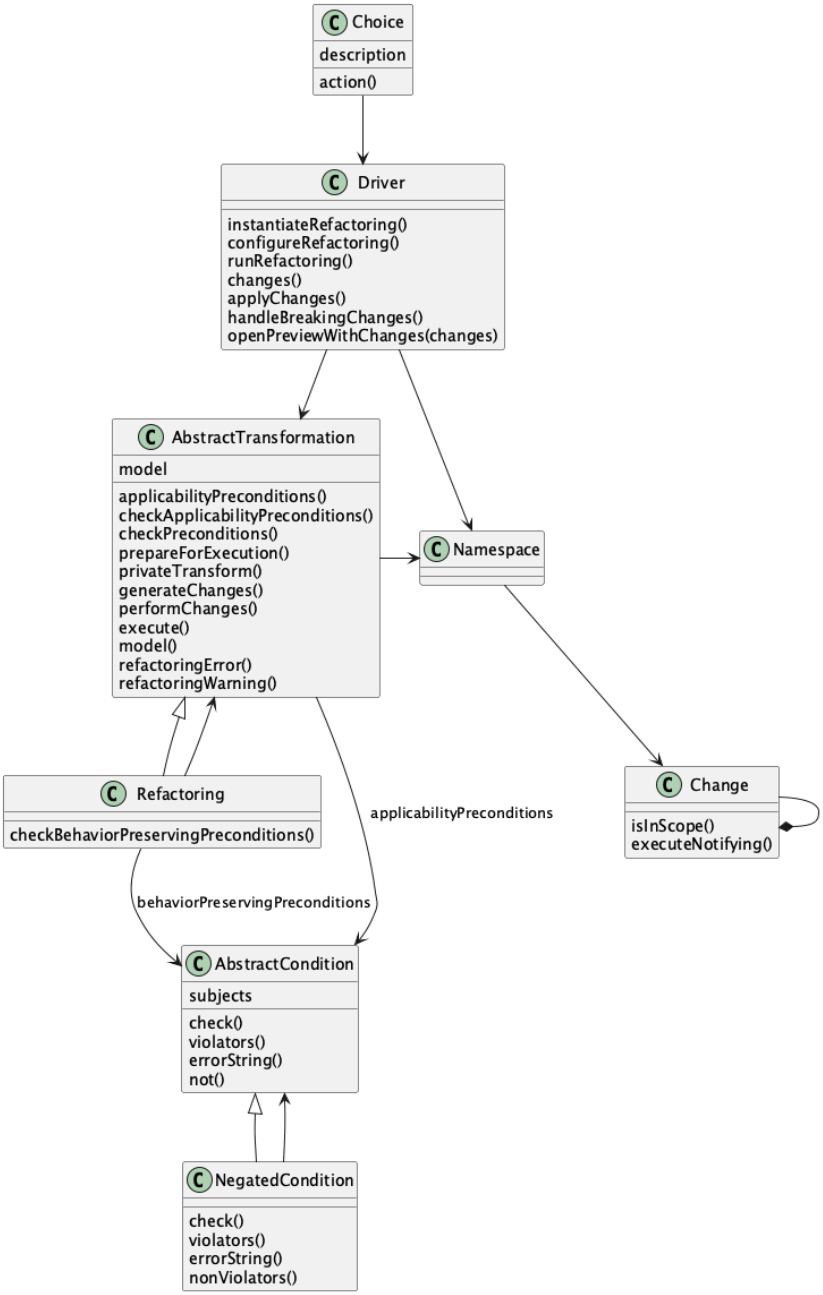

Figure 3.1: Overview of the reference architecture: drivers, preconditions, refactorings, and transformations as separate roles.

Notation. The model uses standard set-theoretic notation [Ros19].

- For sets X and Y, $X \times Y$ denotes their Cartesian product, the set of all pairs (x, y) with $x \in X$ and $y \in Y$.
- 2^X denotes the power set of X, the set of all subsets of X.
- A *total* function $f\colon X \to Y$ assigns exactly one element of Y to every element of X. A *partial* function $f\colon X \rightharpoonup Y$ may be undefined for some elements of X. Its domain of definition $\mathrm{dom}(f) \subseteq X$ is the set of elements of X for which f is defined.
- Angle brackets $\langle \cdot \rangle$ denote finite sequences.

Let $\mathcal{P}$ be the set of well-formed programs of the target language. An operation is a transformation or a refactoring (Section 2.3). Each operation o has its own parameter space Θ_o (for example new selectors, destination classes, or source intervals). A *configuration* of o is a pair $(P, \theta) \in \mathcal{P} \times \Theta_o$, where $P \in \mathcal{P}$ is the program being changed and $\theta \in \Theta_o$ supplies the parameters of the requested change.

Transformations. A *transformation* is a pair

$$\tau = (A_\tau,\ f_\tau),$$

where $A_\tau\colon \mathcal{P} \times \Theta_\tau \to \{\mathsf{true}, \mathsf{false}\}$ is the *applicability predicate* and $f_\tau\colon \mathcal{P} \times \Theta_\tau \rightharpoonup \mathcal{P}$ is a partial change-construction function. Under the *transformation contract*, a change must be constructed for exactly the configurations that the applicability predicate accepts:

$$\mathrm{dom}(f_\tau) = \{(P, \theta) \mid A_\tau(P, \theta) = \mathsf{true}\}.$$

An accepted configuration (P, θ) produces a constructed result $f_\tau(P, \theta) = P'$, which must be a well-formed program, while a rejected configuration has no result. We call $\mathrm{dom}(f_\tau)$ the *construction domain* of τ, and the set of configurations accepted by A_τ its *applicability domain*. For a transformation that satisfies the contract, the two domains coincide. The architecture reifies the individual applicability preconditions as objects that record what caused a failure (Section 3.2). The model uses only their conjunction, the single predicate A_τ.

The contract covers structural applicability only. A transformation keeps its applicability preconditions, so a client that reuses it reuses those checks as well. However, an implementation may omit or incorrectly implement a check. If parsing or compilation then rejects the generated change, execution terminates without a successful result, and the contract is violated.

Refactorings as decorations. A *refactoring* adds the behavior-preserving level on top of a transformation. It does so through behavior-preserving preconditions, which Anquetil *et al.* [ACD+22] distinguish from applicability preconditions (Section 2.3).

Let $\mathcal{D}$ be the set of *diagnostic records*. A diagnostic record identifies a violated behavior-preserving precondition along with the program entities that violate it, that is, one detected *behavior risk*. Reporting a record to the caller is a *warning*.

The behavior-preserving diagnostics of a refactoring R over a transformation τ form a total function

$$BP_R \colon \mathcal{P} \times \Theta_\tau \to 2^{\mathcal{D}}.$$

For each configuration (P, θ), the value $BP_R(P, \theta)$ is the set of behavior risks detected for that configuration. Outside $\mathrm{dom}(f_\tau)$ the configuration is rejected before any behavior-preserving analysis, and we define $BP_R(P, \theta)$ to be the empty set. $BP_R(P, \theta) = \varnothing$ means that the implemented analyses detected no known risk. It is not a proof of observational equivalence, which is undecidable in general (Section 2.3).

We define a refactoring as the decoration of a transformation with that diagnostic function:

$$R = \mathsf{decorate}(\tau,\ BP_R).$$

The refactoring R shares the applicability domain and the change function of τ.

While static analysis can establish some behavior-preserving facts, in a dynamically typed language, many binding and dispatch facts remain unavailable until execution [FMM+11]. For this reason, BP_R does not redefine the structural applicability domain. It partitions that domain into configurations for which no behavior risk is detected, and configurations with one or more detected risks. A violated behavior-preserving precondition issues a warning before the caller decides whether to proceed. The *refactoring contract* therefore keeps the requirements of the transformation contract. It adds that every record in $BP_R(P, \theta)$ must be reported as such a warning. It does not require the declared preconditions to be complete.

Execution under a policy. An execution policy is a function $\pi \colon 2^{\mathcal{D}} \to \{\mathsf{abort}, \mathsf{proceed}\}$ that receives the set of detected behavior risks, $BP_R(P, \theta)$, and decides whether execution continues. We require $\pi(\varnothing) = \mathsf{proceed}$, which means that when no behavior risk is detected, execution proceeds without a decision by the caller. The execution of R under π is the partial function $exec_{R,\pi} \colon \mathcal{P} \times \Theta_\tau \rightharpoonup \mathcal{P}$ with

$$\mathrm{dom}(exec_{R,\pi}) = \{(P, \theta) \in \mathrm{dom}(f_\tau) \mid \pi(BP_R(P, \theta)) = \mathsf{proceed}\},$$

and $exec_{R,\pi}(P, \theta) = f_\tau(P, \theta)$ on that set. A configuration that A_τ rejects is outside $\mathrm{dom}(f_\tau)$ and therefore outside $\mathrm{dom}(exec_{R,\pi})$, so the policy is consulted only for configurations in $\mathrm{dom}(f_\tau)$. A transformation invoked directly has no diagnostic function. Its execution is f_τ itself, and no policy is involved.

The controlled path from refactoring to transformation. Because R and its underlying τ share A_τ and f_τ, the caller has a controlled path between the two behavioral levels:

- invoking R requests the behavior-preserving level, where BP_R is evaluated and the detected risks are reported as warnings;
- invoking the underlying τ requests the behavior-agnostic level, which is R with BP_R set aside while A_τ is kept.

The same structural change is therefore available both as a behavior-preserving refactoring and as a behavior-agnostic transformation, with the applicability checks evaluated at either level.

Composition. Larger operations are modeled as orchestration over a sequence of operations. Let $C = \langle o_1, \ldots, o_n \rangle$, where each o_i is a transformation or a refactoring. For step i, let f_i and A_i denote the change-construction function and the applicability predicate of o_i. When o_i is a refactoring that the composite invokes at the behavior-preserving level, BP_i denotes its diagnostic function. Otherwise, $BP_i(P, \theta_i) = \varnothing$ for every configuration. Let Θ_i denote the parameter space of o_i. The parameter space of the composite is $\Theta_C = \Theta_1 \times \cdots \times \Theta_n$.

Change construction is defined without reference to any policy, as for a single transformation. For a program P and parameters $\theta = (\theta_1, \ldots, \theta_n) \in \Theta_C$, the sequence of intermediate states is defined inductively:

$$P_0 = P; \qquad P_i = f_i(P_{i-1}, \theta_i) \quad (1 \leqslant i \leqslant n),$$

where P_i is defined if and only if P_{i-1} is defined and $A_i(P_{i-1}, \theta_i) = \mathsf{true}$. The composite itself denotes the partial function $f_C \colon \mathcal{P} \times \Theta_C \rightharpoonup \mathcal{P}$ with

$$f_C(P, \theta) = P_n,$$

defined if and only if P_n is defined and the composite's own cross-step preconditions hold for (P, θ). These preconditions form a predicate $A_C \colon \mathcal{P} \times \Theta_C \to \{\mathsf{true}, \mathsf{false}\}$ that covers requirements that cannot be assigned to any single component.

Each step therefore meets the state that the preceding steps produced, not the initial program. Each θ_i is a description, such as a selector or a class name, that step i resolves against P_{i-1}. A later step can therefore refer to entities created by an earlier one. The composite reuses the checks from its components and adds only cross-step conditions. Let $A^\star_C$ denote the applicability of the sequence as a whole. It holds when A_C holds and P_n is defined, that is, when each step in turn is applicable in the state its predecessors produce. Under the *composition contract*, every step must be checked in the state it meets. Because f_C is defined if and only if $A^\star_C$ holds, the pair $(A^\star_C, f_C)$ satisfies the transformation contract. A composite therefore satisfies the same contract as an atomic operation and can itself appear as a step of another composite, with Θ_C as its parameter space.

A composite also reports the behavior risks detected by its steps. Each step reports its diagnostics for the state that it meets, $BP_i(P_{i-1}, \theta_i)$. The composite adds its own diagnostics about the sequence as a whole, $D_C(P, \theta) \subseteq \mathcal{D}$. The behavior-preserving diagnostics of the composite are the union of these sets:

$$BP_C(P, \theta) = D_C(P, \theta) \cup \bigcup_{i=1}^{n} BP_i(P_{i-1}, \theta_i) \quad \text{for } (P, \theta) \in \mathrm{dom}(f_C),$$

and $BP_C(P, \theta) = \varnothing$ otherwise, by the same convention as for a single refactoring, so the function is total.

Decorating the composite transformation $(A^\star_C, f_C)$ with that diagnostic function yields a composite refactoring. It is executed under a policy in the same way as a single refactoring, so the policy decides once, on all the diagnostics of the composite. An engine may instead consult the policy at each refactoring step. It can then abort earlier and construct fewer intermediate

states. When the policy proceeds at every step, the engine constructs the same P_n. Whether P_n is then applied can still differ, because a decision at each step receives one BP_i while the single decision receives all the diagnostics of the composite.

A successful composite returns only the final state P_n. If a step is rejected or fails, the model defines no successful composite result. Atomic application is a realization concern: an implementation can construct all changes before committing them or provide rollback for already-applied changes.

Observation points. The model also fixes the observation points on which the testing method in Chapter 6 relies. An attempt is rejected when an applicability precondition stops it before transformation execution. A violated behavior-preserving precondition reports a warning, and the execution policy then aborts or proceeds. Once transformation execution begins, the attempt either completes with a transformed program or terminates without one.

Precondition defects appear in these outcomes. Completed attempts can expose missing checks through behavior changes observed by project tests. Forced execution after an applicability rejection can expose an overly conservative applicability precondition when the change can still be constructed. Forced execution bypasses the applicability decision that the implementation reports. It therefore tests whether that decision conforms to the transformation contract. Chapter 6 defines the metamorphic relation, execution modes, and outcome checks that use these observations.

3.2 Reified Preconditions

The model in Section 3.1 separates applicability preconditions from behavior-preserving preconditions. To make that separation usable by scripts, tests, and drivers, the architecture represents individual preconditions as explicit objects. If an engine collapses a compound precondition to a single boolean result, the caller can only observe success or failure. It cannot inspect which program entity caused the failure, distinguish applicability rejection from a warning, or choose a context-specific response.

A reified precondition evaluates itself against the program model and records the entities or facts that caused the failure. Because each failed precondition stays visible, a caller can produce a specific diagnostic, offer a targeted warning choice, or classify the outcome automatically. The same precondition object serves interactive and headless execution.

3.3 Interaction Drivers and Execution Flow

When interaction is embedded in the operation itself, execution becomes difficult to reuse in scripts and tests. The operation also accumulates two responsibilities: changing the program and guiding the user through interactive decisions. Interaction drivers separate those responsibilities. They are the orchestration layer for interactive use.

The *interaction contract* states what the separation requires of the core operation. The core operation must be configurable and executable programmatically, without UI prompts. It

reports warnings to a policy that the caller supplies, and the policy decides whether execution proceeds.

The standard execution flow proceeds as follows:

1. **Configuration:** A UI command delegates to a driver and passes the initial context (*e.g.*, the currently selected method). The driver configures the refactoring and invokes the preparation hook that maps the UI context to program-model entities.

2. **User input:** If the refactoring requires user input (*e.g.*, a new method name), the driver gathers it before continuing. For inputs with applicability constraints, the driver can use the corresponding applicability preconditions to validate the input.

3. **Applicability checks:** The driver invokes its applicability-check hook. Each concrete driver defines which applicability preconditions it handles at this point, because some may already have been used while gathering input. If an applicability precondition fails, the driver reports the failure and stops the operation.

4. **Behavior-preserving checks:** The driver requests the refactoring to evaluate its behavior-preserving preconditions, if that refactoring has any.

5. **Warning resolution (choices):** If a behavior-preserving precondition fails, the driver builds a list of possible next steps, represented as *choices*, and displays them to the user. The choices can, for example, let the user stop refactoring to investigate the entities that cause preconditions to fail or continue execution with potential behavior changes. A choice can also offer another refactoring option that might preserve behavior.

6. **Preview and application:** The driver performs the refactoring, collects the changes, previews them, and, once the user confirms, applies them to the system.

Handling complex flows via choices. A concrete driver can specialize the standard flow for a family of refactorings. When a warning occurs, the driver encapsulates alternative execution paths into *choice* objects. Through the choice that the user selects, the driver realizes the execution policy of Section 3.1. Continuing despite the warning corresponds to `proceed`, and stopping to investigate corresponds to `abort`. A choice that starts another operation corresponds to `abort` followed by an invocation of that operation.

For example, Remove Class has a behavior-preserving precondition that the class has no subclasses. When this precondition fails, the driver offers two *choices*. The first continues the refactoring, which removes the class and reparents its subclasses, so the state defined in the class is lost. The second pushes that state down into the subclasses before removing the class. The driver presents these choices according to the failed preconditions and the entities those failures report. With another combination of failed preconditions, the driver offers other choices. Chapter 4 shows the dialog that results.

Figure 3.2 traces such a flow, in which failed preconditions first require additional user input and then offer several resolution choices before the change is applied.

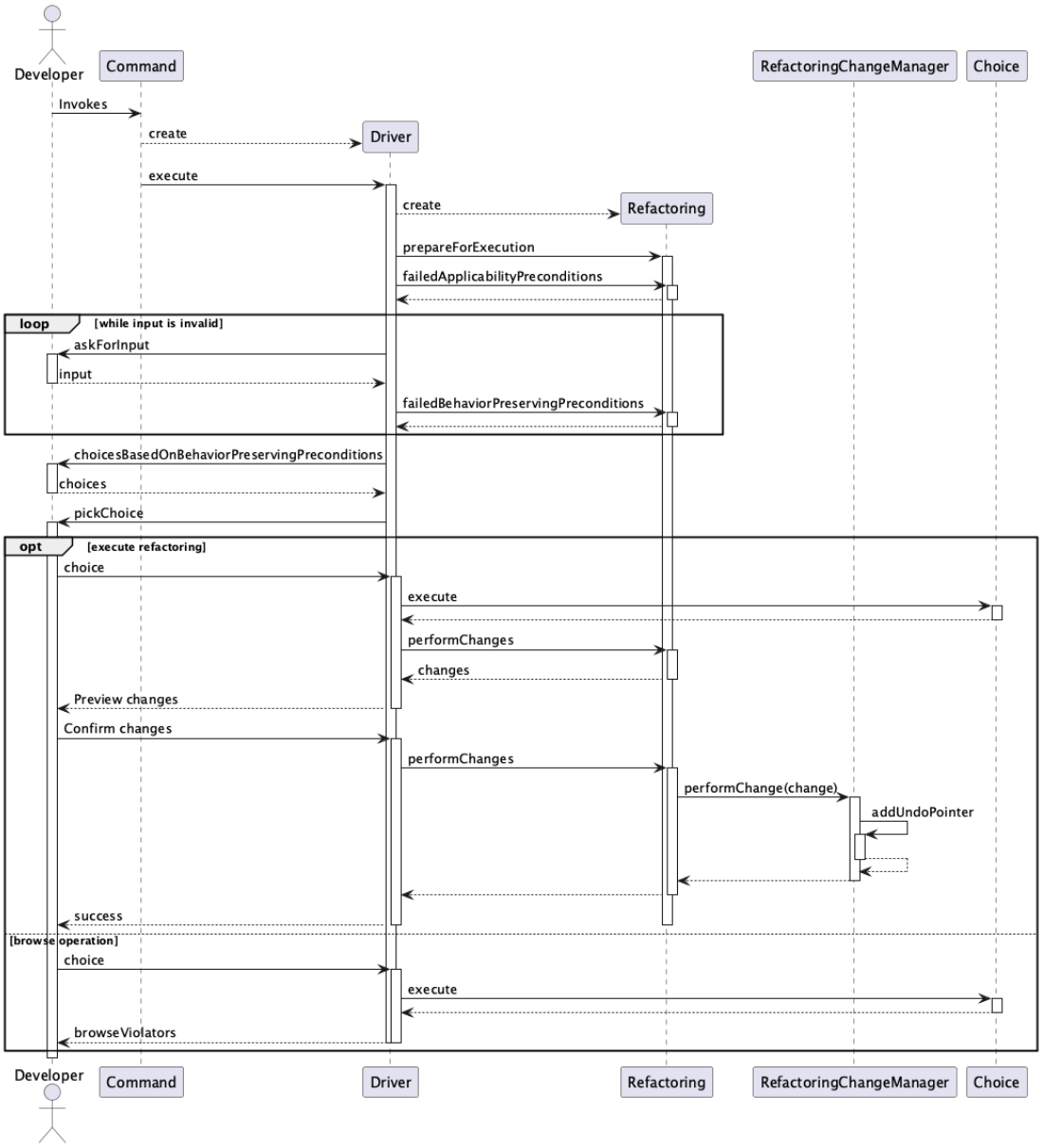

Figure 3.2: A driver flow where failed preconditions trigger user input and multiple resolution choices.

3.4 Summary

The architecture answers RQ1 by placing transformations and refactorings behind a polymorphic API, by decorating transformations with refactorings, and by defining composition over intermediate program states. For RQ2, interaction drivers own parameter collection, warning decisions, previews, and application policy, while the core operation stays callable by headless clients.

The four requirements from Chapter 2 are addressed as follows. Requirement 1, refactorings and transformations, is met by the polymorphic API, the Decorator pattern, and reified preconditions (Sections 3.1 and 3.2). Under a polymorphic API, refactorings and transformations share one execution protocol, so a caller invokes either kind of operation in the same way. Requirement 2, explicit interaction orchestration, is met by the interaction drivers of Section 3.3, which manage parameter requests, warnings, previews, and application decisions. Requirement 3, real-project testing and outcome checks, and Requirement 4, precondition validation, both concern testing. Both need execution that can be driven programmatically, without UI prompts that would block it. The interaction contract requires such execution. Chapters 6 and 7 define and evaluate the testing method itself.

The four roles and their composition are general and would serve a statically typed engine as well. Dynamic typing changes only how many behavior-preserving facts can be decided before execution. Because binding and dispatch facts are often unavailable, the architecture reports every violated behavior-preserving precondition as a warning that the caller decides on, instead of narrowing the applicability domain. The testing method in Chapter 6 relies on those warnings.

Chapter 4 maps the four roles to Pharo classes, Chapter 5 evaluates their reuse and execution in the migrated engine, and Section 5.6 realizes selected roles outside Pharo, in the Python *rope* engine.

Chapter 4

Implementation of the Reference Architecture in Pharo

This chapter describes how the Pharo 14 refactoring engine realizes the reference architecture. It presents the concrete classes and protocols behind the roles defined in Chapter 3: a polymorphic API, refactoring decorators, reified preconditions, and interaction drivers. Chapter 5 reports how far the migration has progressed and evaluates the resulting reuse and headless execution.

Incremental migration. The migration retains Pharo's program and change models as its foundation. Legacy and migrated classes coexist during the incremental migration. The "RB" prefix identifies legacy classes, and the "Re" prefix identifies migrated ones. An existing transformation participates in the architecture when it implements the polymorphic API (Section 4.1), so migrated composites can still contain "RB" transformation classes. The infrastructure for composing transformations predates this work and was adapted to that API.

4.1 Polymorphic API

Legacy refactorings and transformations belonged to separate hierarchies and followed different execution protocols. In the migrated hierarchy, transformations and refactorings are subclasses of ReAbstractTransformation, so both kinds of operation implement the API shown in Figure 4.1.

The polymorphic API has three layers:

- execute generates and applies changes without user interaction.
- generateChanges and performChanges separate construction from application. Preview and composition depend on this separation.
- prepareForExecution, checkPreconditions, and privateTransform expose the individual stages of generateChanges needed by composites and interaction drivers.

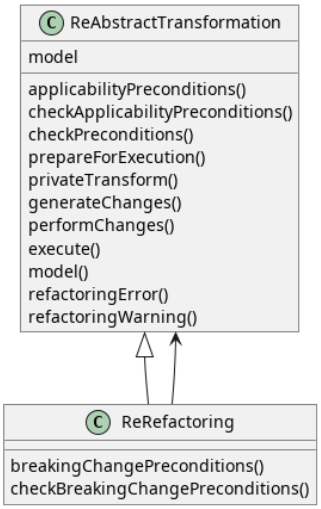

Figure 4.1: Shared design for refactorings and transformations.

```
1  ReAbstractTransformation >> execute
2     self generateChanges.
3     self performChanges
```

Listing 4.1: The execute method of ReAbstractTransformation.

The execute method in Listing 4.1 is the headless entry point. It constructs change objects through generateChanges and applies them through performChanges.

```
1  ReAbstractTransformation >> generateChanges
2     self prepareForExecution.
3     self checkPreconditions.
4     self privateTransform.
5     ^ self changes
```

Listing 4.2: The generateChanges method of ReAbstractTransformation.

Listing 4.2 shows the construction pipeline. prepareForExecution resolves inputs against the current program model, so a later composite step can refer to entities created by an earlier step. prepareForExecution also validates that supplied AST nodes can be resolved to supported program-model entities before change construction begins. checkPreconditions invokes checkApplicabilityPreconditions. In a refactoring, it also invokes checkBreakingChangePreconditions. The first signals an error when an applicability precondition fails, while the second signals a warning when a behavior-preserving precondition fails. In the migrated families, these two checking methods centralize signaling: preconditions return their results,

and transformation steps do not directly send refactoringFailure:, refactoringError:, or refactoringWarning:. A behavior-preserving warning is catchable, so a headless client can install a handler that realizes the execution policy of Section 3.1. A driver realizes the same policy through the choice that the user selects (Section 4.4). If no error is signaled and the policy does not abort, privateTransform constructs the change objects.

The API also exposes applicabilityPreconditions on every operation and breakingChangePreconditions on refactorings. breakingChangePreconditions returns the behavior-preserving preconditions. The name follows the Pharo user interface, which reports a violated behavior-preserving precondition as a potential breaking change (Figure 4.4). Individual precondition accessors let drivers derive messages and alternatives from the preconditions that failed.

Failure and application boundary. performChanges is the application boundary. prepareForExecution, checkPreconditions, and privateTransform run before it, so when preparation, checking, or change construction fails, the accumulated changes are not applied to the target system. A composite follows the same rule. Its steps construct their changes on the program model, so intermediate states exist in the model and not in the target system. The accumulated changes are applied together at the end, so the Pharo realization constructs all changes before committing them (Section 3.1). Application and undo remain delegated to Pharo's existing change model.

4.2 Decorator Realization

Transformations. A migrated transformation implements change construction and the applicability checks that the construction requires. Direct execution retains those checks even when the caller deliberately omits the behavior-preserving level. For example, those checks can reject an accessor for a non-existent instance variable or an invalid selector before compilation. If parsing or compilation nevertheless rejects the generated change, execution terminates without a successful result.

Refactorings. A migrated refactoring uses the Decorator pattern [GHJV95]. It delegates applicability checks and change construction to a transformation, which may itself be composite, and adds behavior-preserving preconditions. The polymorphic API lets the same transformation appear inside a composite or beneath a behavior-preserving refactoring without duplicating its applicability checks. In Pharo 14, the Add Method and Remove Method refactorings are realized this way. Chapter 5 reports the state of the other families.

4.3 Reified Preconditions

A reified precondition is a subclass of ReAbstractCondition. Each instance stores its subjects. These are the objects that it checks, usually program entities and sometimes a name. Calling check evaluates the precondition and records any violating program entities in violators. Clients use those entities and the precondition's errorString to report a specific failure

or construct a contextual choice. A negated precondition (ReNegatedCondition) decorates another precondition and inverts its result while preserving the same inspection protocol. Figure 4.2 summarizes the implementation.

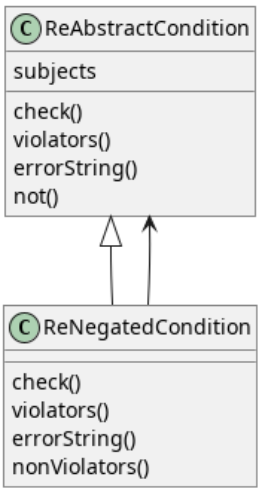

Figure 4.2: Implementation of reified preconditions.

4.4 Interaction Drivers

An interaction driver orchestrates the interactive flow for a refactoring family. A UI command supplies the available context, such as a selected method, and invokes runRefactoring. The driver gathers missing parameters, resolves warnings, previews changes, and lets the user decide whether to apply them. Figure 4.3 shows the resulting execution flow. The driver configures the refactoring and checks its preconditions. If a precondition is violated, the driver reports the violation to the user. Otherwise, it generates, previews, and applies the changes.

```
1 ReInteractionDriver >> runRefactoring
2    self configureRefactoring.
3    self gatherUserInput ifFalse: [ ^ self ].
4    self hasFailedApplicabilityPreconditions ifTrue: [ ^ self ].
5    self hasFailedBehaviorPreservingPreconditions
6       ifTrue: [ self handleBreakingChanges ]
7       ifFalse: [ self applyChanges ]
```

Listing 4.3: The runRefactoring method of ReInteractionDriver.

Listing 4.3 shows runRefactoring, the template method that concrete drivers specialize. It stops on an applicability failure and delegates a behavior-preserving failure to handleBreak-

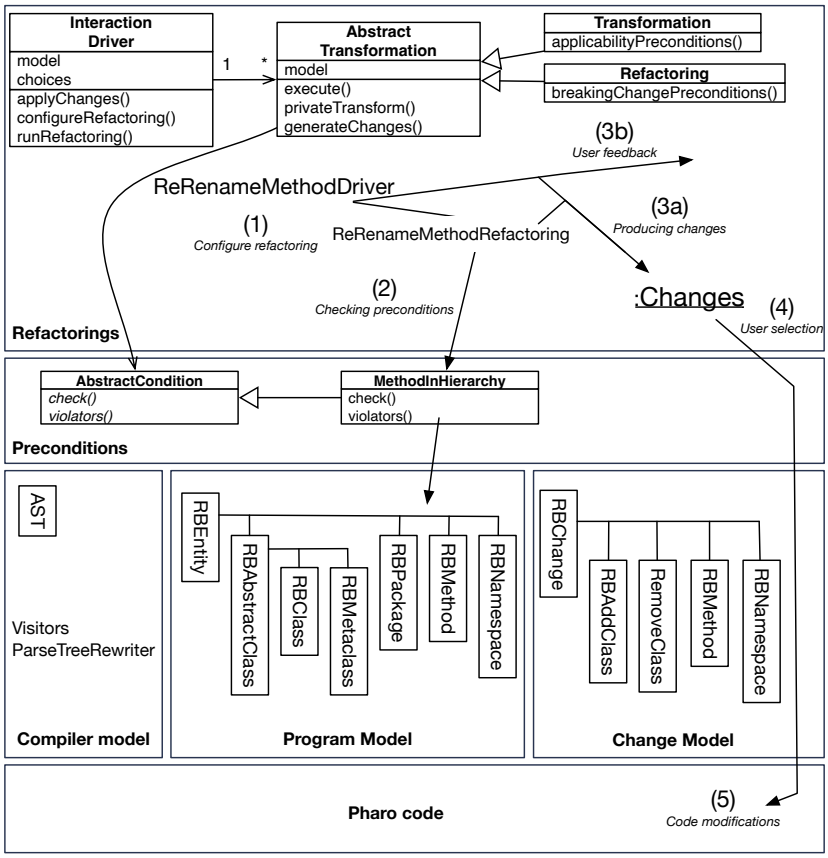

Figure 4.3: The decoupled execution flow: the driver configures the refactoring (1) and checks preconditions (2). Violations are reported to the user (3b). Otherwise, changes are generated (3a), previewed (4), and applied to the code (5).

ingChanges. For Remove Class, breakingChoices builds the choices offered at that point from the individual failed preconditions. Listing 4.4 shows the method, and Figure 4.4 the resulting dialog.

Precondition-level reuse. The implementation preserves the result of each precondition. The driver inspects haveNoReferences, emptyClasses, and noSubclasses individually, so it can offer actions appropriate to the particular combination of failed preconditions. Headless clients evaluate the same preconditions together through checkPreconditions. Section 5.5 evaluates how this representation supports both interactive and headless execution.

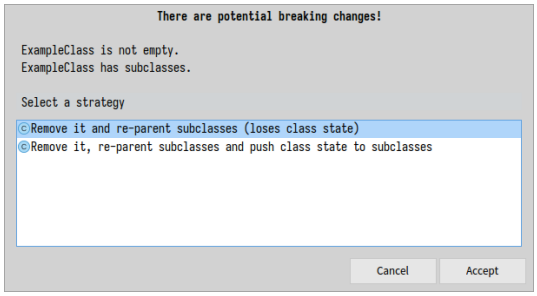

Figure 4.4: Choice dialog derived from failed behavior-preserving preconditions.

```
ReRemoveClassInteractionDriver >> breakingChoices
   | items |
   items := OrderedCollection new.
   items add: (ReRemoveClassReparentChoice new
                  driver: self;
                  classesHaveSubclasses: noSubclasses isFalse;
                  emptyClasses: emptyClasses isTrue).
   (noSubclasses isFalse and: [ emptyClasses isFalse ]) ifTrue: [
      items add: (ReRemoveClassAndPushStateToSubclassChoice new driver: self) ].
   haveNoReferences isFalse ifTrue: [
      items add: (ReBrowseClassReferencesChoice new driver: self) ].
   ^ items
```

Listing 4.4: The breakingChoices method of ReRemoveClassInteractionDriver.

4.5 Program Model Interface

Preconditions query program entities, while transformations use change-producing operations to add, remove, or rewrite those entities. Table A.1 in Appendix A groups the model operations needed for these responsibilities. Reusable transformations represent recurring checked program operations, such as adding or removing methods, classes, and variables. Local AST edits remain implementation steps within those operations.

Listing 4.5 shows a refactoring that still uses the program-model interface directly. It queries subclasses and then calls the change-producing operations of the program model.

```
RePullUpInstanceVariableRefactoring >> privateTransform
   class allSubclasses do:
      [:each | (each directlyDefinesInstanceVariable: variableName)
         ifTrue: [each removeInstanceVariable: variableName]].
   class addInstanceVariable: variableName
```

Listing 4.5: The privateTransform method of RePullUpInstanceVariableRefactoring.

Replacing these calls with Remove Variable and Add Variable transformations would reuse their applicability checks and expose the intended operations as composite steps.

Chapter 5 reports how many such usages have been replaced.

4.6 Summary

Refactorings and transformations share one execution protocol, so the same transformation can appear inside a composite or beneath a decorating refactoring. Each precondition keeps its own result, so a driver builds a choice from the preconditions that failed while a headless client evaluates them together. The program and change models remain the underlying layer. Reusable transformations are checked program operations built on the low-level APIs of these models. Refactorings and composites reuse those operations instead of calling the APIs directly. Chapter 5 evaluates the reuse, composition, and headless execution realized by this migration.

Chapter 5

Architecture Validation and Evaluation

The previous chapter described the Pharo implementation of the reference architecture. This chapter evaluates the structural and practical improvements of the new architecture and addresses **RQ1** and **RQ2**. The evaluation covers the migrated part of the engine. Throughout the chapter, Pharo 11 and Pharo 14 denote the refactoring engines shipped with those versions: the legacy engine before the migration and the migrated engine. All measurements use the build commits `963dc3a7de` for Pharo 11 and `c08e24c493` for Pharo 14, so that the results can be reproduced. A replication package for these measurements is archived on Zenodo [Šar26].

Migrating the refactoring catalog remains a long-term endeavor, and the migration continues in the Pharo development branch. The bulk of the initial work consisted of defining the new architecture, separating the precondition categories, and validating the design on selected refactoring families. In Pharo 14, the migration has progressed at several distinct levels:

- The refactoring catalog now exposes separate applicability and behavior-preserving precondition protocols. Migrated transformations carry applicability checks, while migrated refactorings specify behavior-preserving checks separately.
- A polymorphic API is available for refactorings and transformations, so both kinds of operation are callable through the same protocol.
- In total, 35 usages of the program-model API were replaced with explicit transformation reuse.
- Two refactoring wrappers, Add Method and Remove Method, use the Decorator pattern and delegate to their transformations. In the development branch, which is Pharo 15 at the time of writing, nine wrappers delegate to six transformation classes.[1] The seven additional wrappers are two Remove Variable variants, two Pull Up Variable variants, two Push Down Variable variants, and Remove Class.

[1] Pharo 15 build commit `aaeda0c774`.

Table 5.1: Drivers in Pharo 14.

Abstract Instance Variable	Protect Instance Variable
Convert Temporary To Instance Variable	Pull Up Method
Deprecate Method	Pull Up Variable
Duplicate Class	Push Down Instance Variable
Extract Method	Push Down Method
Extract Temporary	Push Down Method In Some Classes
Generate Accessors	Remove Class
Generate Equal And Hash	Remove Instance Variables
Generate PrintOn	Remove Method
Inline Into Local Senders	Remove Shared Variables
Inline Method	Rename Argument Or Temporary
Inline Temporary	Rename Class
Insert Subclass	Rename Instance Variable
Insert Superclass	Rename Method
Move Method To Class	Rename Package
Move Methods To Class Side	Rename Shared Variable
Move Temporary Definition	Replace Message Send
Move Variables To Class Side	

- A total of 35 drivers have been created for several families of refactorings and are displayed in Table 5.1.
- Other families remain to be fully migrated to the decorator and driver patterns.

5.1 RQ1: Reuse and Expressiveness

This section first shows how refactorings reuse transformations instead of calling the program model directly, then reports reuse counts and clone-detection results. The composite Extract Method refactoring case study and two domain-specific extensions follow.

5.1.1 Reuse of Transformations

The polymorphic API introduced in Section 4.1 makes refactorings and transformations callable through the same execution protocol. This lets a refactoring compose existing transformations instead of directly manipulating the program model. The Change Method Name family illustrates this shift. In the legacy implementation described in Section 2.7.3, related refactorings inherited from an abstract RBChangeMethodNameRefactoring that implemented much of the operation. In the new implementation, the shared behavior is expressed through composition over independent transformations.

Listing 5.1 shows the replacement of inheritance reuse with explicit transformation reuse. ReChangeMethodNameRefactoring delegates message-send updates to ReReplaceMessageSendTransformation and method removal to ReRemoveMethodTransformation. Figure 5.1 shows the resulting design: these transformations are independent operation objects rather than subclasses in the change-method-name hierarchy.

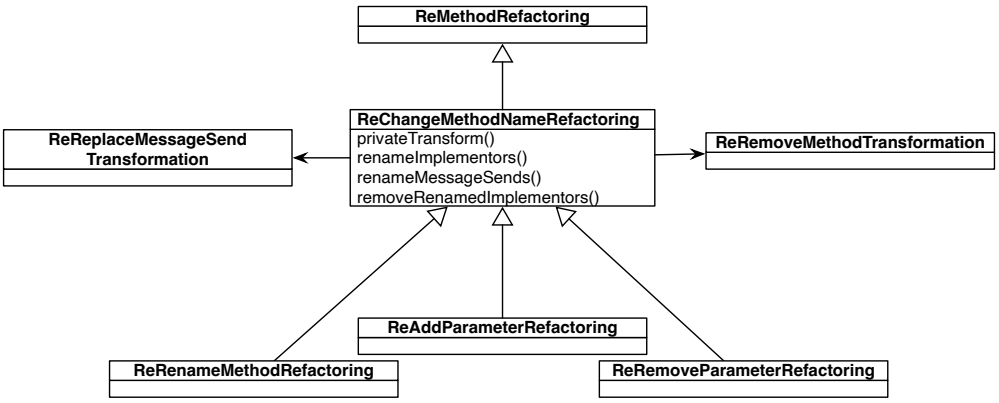

Figure 5.1: Class diagram representing the new ReChangeMethodNameRefactoring implementation.

```
1  ReChangeMethodNameRefactoring >> privateTransform
2     self renameImplementors.
3     self renameMessageSends.
4     self removeRenamedImplementors
5
6  ReChangeMethodNameRefactoring >> renameMessageSends
7     self generateChangesFor: (ReReplaceMessageSendTransformation
8           model: self model
9           replaceMethod: oldSelector
10          in: class
11          to: newSelector
12          permutation: permutation
13          inAllClasses: true
14          newArgs: self newArgs)
15
16 ReChangeMethodNameRefactoring >> removeRenamedImplementors
17    oldSelector = newSelector
18       ifTrue: [ ^ self ].
19    self implementors
20       do: [ :each |
21          self generateChangesFor:
22             (ReRemoveMethodTransformation selector: oldSelector from: each) ]
```

Listing 5.1: Transformation reuse in ReChangeMethodNameRefactoring.

The distinction between refactoring and transformation also records intent. Reusing a transformation, such as ReAddMethodTransformation, requests the checked structural change and deliberately omits behavior-preserving checks, such as those for accidental shadowing or overriding. Reusing the corresponding refactoring decorator requests the behavior-preserving level. The choice is therefore visible in the code instead of having to be inferred from direct model calls.

In migrated wrappers, the refactoring delegates to its transformation through the Decorator pattern. Listing 5.2 shows that Remove Method reuses the applicability checks of its decorated transformation and adds its own behavior-preserving checks.

```
1  ReRemoveMethodRefactoring >> applicabilityPreconditions
2      ^ transformations applicabilityPreconditions
3
4  ReRemoveMethodRefactoring >> breakingChangePreconditions
5     ^ {(RBCondition withBlock: [ self checkSuperMethods ]).
6        (RBCondition withBlock: [ self senders isEmpty ]
7              errorString: 'Cannot remove method because it has senders') }
```

Listing 5.2: Remove Method refactoring precondition checking logic.

The same decorator delegates change construction to its transformation. Whether reused directly or through its decorator, the transformation keeps its applicability checks, and the refactoring adds behavior-preserving checks only where behavior preservation is the goal.

5.1.2 Quantitative Reuse Measurements

Of the 35 replaced program-model usages listed above, most reuse ReAddMethodTransformation and ReRemoveMethodTransformation, with a smaller number reusing ReRemoveMethodRefactoring. This count measures one direct migration activity: replacing model-level calls with operation-level objects. Table 5.2 reports a separate architectural metric, the breadth of reuse of essential transformations across all refactorings and transformations in the analyzed codebase. It compares the reuse of essential transformations between Pharo 11 and Pharo 14. Essential transformations change program entities such as methods, classes, and variables, in contrast to the transformations that encapsulate local AST modifications. We measure the breadth of reuse as the number of distinct refactoring and transformation classes that directly reference each transformation. Class-side and instance-side references are deduplicated, and the transformation itself is not counted as its own client. The increase is concentrated in Add Method, reused by 28 clients in Pharo 14, up from 5 in Pharo 11, and Remove Method, reused by 14, up from 3. These two transformations are the primary composition primitives migrated to the decorator-based architecture. The replacement of model-level calls did not target the remaining essential transformations. The reuse of Add Variable (from 4 to 3), Insert New Class (from 2 to 1), and Remove Variable (from 4 to 3) decreased by one. Between Pharo 11 and Pharo 14, a single unused transformation that had reused all three was removed.

Our earlier journal paper reports 35 clients of Add Method and 18 of Remove Method

in a Pharo 12 snapshot [SADT24]. The current evaluation numbers differ due to additional migration efforts and removal of duplicate logic. The Pharo 11 and Pharo 14 measurements reported here use the same script and counting rule.

Table 5.2: Reuse of essential transformations, comparing Pharo 11 with Pharo 14. Each value is the number of distinct refactorings and transformations that reuse the transformation.

Transformation	Pharo 11	Pharo 14
Add Method	5	28
Add Variable	4	3
Insert New Class	2	1
Remove Class	0	0
Remove Method	3	14
Remove Variable	4	3
Replace Message Send	0	1

Table 5.3: Reuse of transformations that encapsulate AST modifications. The reuse count is the number of distinct refactorings and transformations that reuse each one in Pharo 14.

Transformation	Reuse count
Add Annotation	0
Add Assignment	0
Add Message Send	0
Add Return Statement	0
Add Subtree	0
Add Temporary Variable	1
Remove Annotation	0
Remove Assignment	0
Remove Return Statement	0
Remove Temporary Variable	2
Replace Subtree	1

Table 5.3 shows a different reuse pattern for transformations that encapsulate local AST modifications. Eight of the eleven AST wrapper transformations have no observed reuse, and the group has only four clients in total. This contrasts with the essential transformations in Table 5.2, where Add Method and Remove Method are reused by 28 and 14 distinct refactorings and transformations, respectively.

This result suggests that reification alone is not sufficient to make an operation a useful composition unit. Reuse is strongest when a transformation captures a semantic program operation with its own applicability checks, such as adding or removing a method from the program model. By contrast, AST wrapper transformations operate at a finer syntactic granularity. They are useful for implementing larger transformations. Their low observed

reuse indicates that they are better treated as internal helpers. These wrappers were inherited from the legacy engine.

5.1.3 Clone-Detection Analysis of Refactoring and Transformation Duplication

To complement the reuse analysis, we measured syntactic code duplication in the packages that contain the refactorings, the transformations, and their preconditions. In Pharo 11, these are Refactoring-Core, which contains most refactorings and the precondition classes, and Refactoring2-Transformations, which contains the transformation hierarchy. In Pharo 14, the concrete refactorings and transformations are in Refactoring-Transformations, and the reified preconditions are in the new package Refactoring-Conditions. Refactoring-Core no longer contains a concrete refactoring. It keeps the program model, the base precondition classes, and the roots of both hierarchies. We analyze all three packages in Pharo 14, so that the preconditions are in scope in both versions. Tests were excluded because the goal is to measure implementation duplication in the refactoring engine itself. The measurement uses jscpd 4.2.5, a token-based clone detector whose language-aware tokenizer processes Smalltalk source directly. A clone is reported when at least 50 consecutive tokens recur across the analyzed files.

Table 5.4: Code duplication reported by jscpd for the refactoring, transformation, and precondition packages, excluding tests.

Version	Files	Lines	Clone pairs	Duplicated lines	Duplicated tokens
Pharo 11	179	24,254	104	1,460 (6.02%)	13,198 (8.00%)
Pharo 14	243	25,261	32	385 (1.52%)	3,667 (2.22%)

Table 5.4 shows that duplication in the analyzed packages decreased substantially. Although the analyzed codebase grew from 179 to 243 files and from 24,254 to 25,261 lines, the number of reported clone pairs decreased from 104 to 32. The duplicated-line ratio decreased from 6.02% to 1.52%, a relative reduction of 74.7%. The duplicated-token ratio decreased from 8.00% to 2.22%, a relative reduction of 72.2%.

This result supports the duplication claim of H1. The legacy architecture kept refactorings and transformations in separate hierarchies with incompatible APIs. This separation encouraged parallel implementations of similar behavior.

Where the duplication lived. We classify each clone pair by the kinds of the two classes that it connects. The kind of a class follows from the suffix of its name, Refactoring or Transformation. We do not classify by package, because the refactorings moved into the transformation package between the two versions. In the legacy engine, 79 of the 104 clone pairs (76%) connect a refactoring class with a transformation class. Each of these operations was implemented twice, once as a refactoring and once as a near-identical transformation. Even the base classes RBRefactoring and RBTransformation were clones of each other. In the new engine, these pairs fall from 79 to 19 (Table 5.5). The fully migrated families disappear,

while 13 of the 19 remaining pairs belong to two operations not yet migrated, Add Parameter and Abstract Variables. The other families each account for at most three clone pairs in the legacy engine, and most disappear in the new engine. This is the form of duplication that the polymorphic API and Decorator pattern were designed to remove.

Table 5.5: Largest families of clone pairs that connect a refactoring class with a transformation class. Each legacy operation existed as both a refactoring and a near-identical transformation. Migrated families drop to zero in the new engine, while not-yet-migrated operations persist.

Operation family	Pharo 11	Pharo 14
Inline Method	17	0
Move Method	8	0
Add Parameter	8	7
Pull Up Method	7	0
Abstract Variables	6	6
Other families	33	6
Total	**79**	**19**

With Refactoring-UI included, the overall duplication ratio also decreased, from 5.91% duplicated lines in Pharo 11 to 1.47% in Pharo 14. We treat the comparison without Refactoring-UI as the primary result. Refactoring-UI changed substantially between the two versions, and the central reuse claim concerns the relationship between refactorings and transformations.

5.2 RQ1 Case Study: Composite Extract Method

The Extract Method refactoring combines selection analysis, control-flow handling, preconditions, method construction, and source replacement [FBB+99, SVEdM09, Sch10]. This case study evaluates whether expressing its structural changes as an explicit composition reduces implementation complexity and creates reusable extension points. The legacy operation is described from the Pharo 12 engine, the snapshot the composite rewrite started from. The structural metrics compare both implementations as they stand in Pharo 14.

5.2.1 Analysis of the Legacy Implementation

The legacy Extract Method refactoring, inherited from the original Refactoring Browser [RBJO96, RBJ97] and present in the Pharo 12 engine, is implemented as a single monolithic operation. One method interleaves four concerns that the reference architecture keeps separate: precondition checking, the calculations needed to build the extracted method, the source transformation, and user interaction. The calculations determine, for example, which temporaries to pass as arguments and which value to return. The interaction prompts for the method name and asks whether to reuse an existing equivalent method.

Appendix B documents the legacy execution flow and analyzes two language-specific concerns that make the operation intricate. A return may need to be added to the extracted method or wrapped around its call in the source method, including for multiple assignments. The operation must also account for non-local returns in blocks.

Three properties of the legacy implementation are relevant for evaluating the architecture. First, precondition checking and transformation setup are mixed, so the preconditions cannot be reused or reasoned about independently. Second, transformation logic and user interaction are mixed, so the operation cannot run headlessly. Third, the operation performs every structural change itself rather than reusing the engine's transformations, so it duplicates analysis the engine already implements [ACD+22]. These are the coupling and duplication problems identified for the engine as a whole in Section 2.8. The Extract Method refactoring is their most acute instance. The remainder of this case study re-expresses the operation as a composite sequence over reusable elementary transformations and compares it with the legacy implementation.

5.2.2 Composite Sequence of Operations

The composite implementation builds on the explicit composition of operations developed by Santos [dSS17] and on the transformation/refactoring separation evaluated in our earlier work [SADT24]. It separates preparation, precondition evaluation, and change construction.

- **Preparation for precondition checking and execution.** In the first step, the Extract Method refactoring parses the method and the code selected for extraction. If either of them cannot be parsed, the remaining computations are skipped, and the refactoring is aborted. The remaining computations include: calculating temporaries, determining which assigned variables are read after the selection, identifying arguments for the extracted method, and determining whether the source method needs to return the result (*i.e.,* wrap it in a return statement).

- **Precondition checking.** The refactoring checks whether the parse tree of the source method and the extraction subtree were parsed correctly. It then verifies whether the selection is valid and can be extracted. The refactoring checks:

  - for temporaries or assignments that are read before being written,
  - that the subtree has at most one assignment whose value is used after the subtree, and
  - that a subtree containing a non-local return ends with the source method's explicit return (Section B.3).

- **Transformation phase.** First, a new method node is created based on the selected subtree, temporaries, assignments, and arguments. Next, the refactoring searches the method's class hierarchy to find an existing selector that has an equivalent tree to the selected subtree (*i.e.,* the created method node). If a match is found, a new method is not created. The selected code is instead replaced with an invocation of the found selector.

If an equivalent method is not found, the extraction is performed in three steps (see Listing 5.3):

- The method node is compiled with the Add Method transformation.
- The selected code in the source method is replaced with an invocation to the extracted method.
- Unused temporaries are removed from the source method.

Listing 5.3 shows the change-construction phase.

```
ReCompositeExtractMethodRefactoring >> buildTransformationFor: newMethodName

    ^ OrderedCollection new
        add: (RBAddMethodTransformation
                model: self model
                sourceCode: newMethod newSource
                in: class
                withProtocol: Protocol unclassified);
        add: (RBReplaceSubtreeTransformation
                model: self model
                replace: sourceCode
                to: (self messageSendWith: newMethodName)
                inMethod: selector
                inClass: class);
        add: (ReRemoveUnusedTemporaryVariableRefactoring
                model: self model
                inMethod: selector
                inClass: class name);
        yourself
```

Listing 5.3: buildTransformationFor method of composite extract method refactoring.

The composite reuses RBAddMethodTransformation, RBReplaceSubtreeTransformation, and ReRemoveUnusedTemporaryVariableRefactoring rather than implementing their model updates inside Extract Method refactoring.

5.2.3 Evaluation

This section compares the legacy and composite implementations of Extract Method by counting methods and source lines and summing per-method cyclomatic complexity. Both classes, RBExtractMethodRefactoring and ReCompositeExtractMethodRefactoring, are measured in Pharo 14, so the comparison is between two implementations of the same operation in the same engine.

The composite implementation keeps preparation, checks, and interaction in separate methods and describes structural changes through operation objects. Table 5.6 reports this comparison.

The composite implementation is smaller in this case study: 28% fewer methods, 30% fewer lines, and 51% lower cumulative cyclomatic complexity than the legacy implementation.

Chapter 7 reports the defects confirmed in both implementations.

Table 5.6: Structural metrics for the legacy and composite implementations of the Extract Method refactoring. The method count and the cyclomatic complexity cover instance-side methods only, because the class-side methods are constructors and not part of the operation. Complexity is the sum of blockNodes size over those methods, the measure used by Pharo's own ReCyclomaticComplexityRule. Lines of code come from linesOfCode, which sums the non-empty source lines of both sides.

Metric	Legacy	Composite	Difference
Number of Methods	47	34	∼28% fewer
Cumulative Cyclomatic Complexity	94	46	∼51% lower
Avg. Cyclomatic Complexity per Method	2.0	1.35	∼32% lower
Lines of Code (LoC)	370	258	∼30% fewer

5.3 RQ1 Case Studies: Domain-Specific Extensions

Domain-specific refactorings adapt a general structural change to a framework-specific or language-specific constraint. This section evaluates whether the composite Extract Method refactoring supports such adaptations by changing its checks or inserting transformations rather than copying its extraction logic. The two cases are Extract SetUp refactoring and Extract with Pragma refactoring.

- The Extract SetUp refactoring helps the developer define which part of the test methods should be automatically turned into a setUp method. The setUp method in the SUnit and JUnit 3.0 frameworks creates the test fixture before any test method execution [DPS23, BG98].
- The Extract with Pragma refactoring helps the developer extract methods that carry domain-specific pragmas.

5.3.1 Composite Extract SetUp

The Extract SetUp refactoring replaces the general assignment precondition with checks that the target is a TestCase subclass and that extraction will not replace an existing setUp method. It then composes five structural steps:

- Add a new setUp method based on the selected subtree.
- Add a super send to the setUp method.
- Transform all assignment variables to instance variables.
- Remove the selected subtree from the source method.
- Remove unused temporaries from the source method.

Listing 5.4 shows the resulting sequence.

```
ReCompositeSetUpMethodRefactoring >> buildTransformationFor: newMethodName

    ^ OrderedCollection new
        add: (RBAddMethodTransformation
                model: self model
                sourceCode: newMethod newSource
                in: class
                withProtocol: (Protocol named: #running));
        add: (ReAddSuperSendAsFirstStatementTransformation
                model: self model
                methodTree: newMethod
                inClass: class);
        addAll: (assignments collect: [ :var | RBTemporaryToInstanceVariableRefactoring
                model: self model
                class: class
                selector: selector
                variable: var ]);
        add: (RBRemoveSubtreeTransformation
                model: self model
                remove: sourceCode
                fromMethod: selector
                inClass: class);
        add: (ReRemoveUnusedTemporaryVariableRefactoring
                model: self model
                inMethod: selector
                inClass: class name);
        yourself
```

Listing 5.4: buildTransformationFor method of composite extract setUp method refactoring.

The implementation reuses the extraction preparation and precondition structure. It replaces the general checks and the structural sequence with domain-specific ones.

5.3.2 Composite Extract with Pragma

Slang methods use pragmas to carry compilation directives and type information required when Pharo code is translated to C [MBBI18, IKM+97, DMP16]. The Extract with Pragma refactoring case supports the #type declareC pragma kind by identifying applicable pragmas during preparation and inserting ReAddPragmaTransformation instances into the extraction sequence.

```
ReCompositeExtractMethodWithPragmasRefactoring >> buildTransformationFor: newMethodName

    | messageSend |
    messageSend := self messageSendWith: newMethodName.
    ^ OrderedCollection new
        add: (RBAddMethodTransformation
                model: self model
                sourceCode: newMethod newSource
                in: class
```

```
10                withProtocol: Protocol unclassified);
11          add: (RBReplaceSubtreeTransformation
12                model: self model
13                replace: sourceCode
14                to: messageSend
15                inMethod: selector
16                inClass: class);
17          addAll: (pragmasToExtract collect: [ :p |
18                    ReAddPragmaTransformation
19                       model: self model
20                       addPragma: p
21                       inMethod: newMethod
22                       inClass: class]);
23          add: (ReRemoveUnusedTemporaryVariableRefactoring
24                model: self model
25                inMethod: selector
26                inClass: class name);
27          yourself
```

Listing 5.5: buildTransformationFor method of composite Extract Method with pragmas.

Listing 5.5 differs from the general extraction sequence by inserting one ReAddPragmaTransformation instance for each pragma in pragmasToExtract. The two cases show two extension mechanisms relevant to RQ1. Extract SetUp refactoring replaces general checks and structural steps with domain-specific ones. Extract with Pragma refactoring retains the common preparation and component operations and extends the sequence with pragma additions.

5.4 RQ1 Synthesis

The reuse counts, clone analysis, composite Extract Method refactoring case study, and domain-specific extensions support the same conclusion. The counts and clone analysis show that duplication fell where the migration introduced a reusable operation. The case studies show that the resulting operations compose to form a simplified Extract Method refactoring and domain-specific variants that replace or extend composed sequences. The reuse tables also show the limit: the gains concentrate on transformations that represent checked *program* operations, such as Add Method and Remove Method. The finer-grained AST wrappers remain internal helpers. This is the answer to RQ1: the architecture improves reuse when transformations represent checked program operations and when larger refactorings are expressed as compositions over those operations. It does not improve reuse at every level of syntactic editing.

5.5 RQ2: Interactive and Headless Execution

RQ2 asks whether interaction can be moved out of the refactoring core while the same operation remains available to interactive and headless clients. We examine three consequences of separating drivers from the core: centralized signaling in core classes,

contextual handling of failed preconditions, and headless execution through the polymorphic API.

5.5.1 Quantitative Evaluation of UI Decoupling

In the legacy architecture, operation-specific core methods directly signaled failures, errors, and warnings. The legacy architecture also coupled core operations to UI managers for input and change confirmation. The migration centralizes signaling and relocates interaction decisions to the ReInteractionDriver hierarchy. We count the methods in the whole image that send the legacy signaling messages. A method that signals twice is counted once. These are static counts of signaling methods, not of every UI dependency. The number of methods that send refactoringWarning: decreased from 28 to 15, and the combined number for refactoringError: and the deprecated refactoringFailure: decreased from 163 to 98. These counts describe an incomplete migration. The remaining sending methods identify the legacy sites where warning or failure signaling has not yet been moved to the centralized checking methods. The migrated families examined here contain no direct sends to these legacy signaling entry methods in their core operations.

5.5.2 Qualitative Evaluation of User Feedback

The Remove Class driver in Section 4.4 illustrates how a driver constructs the interaction. Its breakingChoices method (Listing 4.4) inspects the failed preconditions separately and derives the choices in Figure 4.4.

5.5.3 Headless Execution Example

Listing 5.6 invokes the same ReRemoveClassRefactoring used by the interactive driver, but it does so through the polymorphic API, without a driver.

```
1  classesToRemove do: [ :each |
2      (ReRemoveClassRefactoring model: model className: each) execute ]
```

Listing 5.6: An example of scripting refactorings.

The listing demonstrates that the operation has a headless entry point. If a behavior-preserving precondition fails, the operation raises a catchable warning rather than opening a dialog. A batch client can install its own handling policy. Refazz in Chapter 6 uses this entry point.

The signaling counts, driver example, and headless invocation answer RQ2: the migrated operations use one core implementation for interactive and headless execution. Drivers own parameter collection, warning decisions, previews, and application policy.

5.6 Cross-Language Transfer: A Python Proof of Concept in *rope*

The preceding sections evaluated the architecture in Pharo. This section explores whether the architectural concepts can be implemented in a dynamically typed object-oriented language other than Pharo. It first presents a scoped proof of concept based on *rope*, an independently developed Python refactoring engine.[2] This proof of concept covers method rename, change signature, and selected behavior-preserving preconditions without replacing the engine's program or change models. The section then generalizes from the Pharo and *rope* realizations. It maps the architectural roles onto the engine capabilities that other dynamically typed object-oriented languages would need to provide.

The case study examines the transformation and refactoring roles, the caller's choice between their two behavioral levels, and the composition contract. Method rename exercises decoration around an atomic transformation, while change signature exercises composition of parameter-level steps (adding, removing, and reordering parameters).

Scope and availability. The proof of concept adds an architectural layer above the engine's program and change models, as the Pharo migration did over Pharo's. Both models remain untouched, and the engine's own application boundary still applies and undoes the changes. The two operations are built on that layer and delegate change construction to the engine's existing implementations. Their implementation is available as a public branch of a fork of upstream *rope*, based on upstream commit `f005ac78` and pinned at proof-of-concept commit `77b23b4d`.[3] At that commit, the full test suite under Python 3.14.7 reported 2224 passing tests, 7 skipped tests, and 5 expected failures. The passing tests include 103 tests added for the architecture, and all 137 pre-existing tests in the rename and change-signature modules also pass.

The implementation in Python was feasible because *rope* meets the baseline constraint outlined in Chapter 2. Its project and program model answer the semantic queries required by the preconditions, while its change sets offer deferred, source-preserving changes with history undo. Within its stated scope, the *rope* case study provides evidence for the two roles and the composition contract examined here. The separation of transformation and refactoring and the treatment of behavior-preserving violations as the caller's decision needed no Smalltalk-specific mechanism. The explicit choice between the two behavioral levels involves selecting which class to instantiate: the transformation or its decorating refactoring. This choice is supported by any class-based object-oriented language.

Granularity of composition. The composite reuses the engine's own parameter-level steps, promoted to the transformation role. The architecture layer extends the engine's existing objects instead of adding a parallel model beside them. Later steps are configured and checked against the state produced by their predecessors.

For example, on a three-parameter method, adding a parameter at position 3 first creates

[2] https://github.com/python-rope/rope

[3] Proof-of-concept commit: https://github.com/balsa-sarenac/rope/tree/77b23b4dce22ef6f5769d4a041378ebb9f7b0498. Upstream base commit: https://github.com/python-rope/rope/commit/f005ac786da7d556fdf5733e9b9ddae1eaf27242.

that position, so a subsequent removal at position 3 becomes applicable. The removal would be out of range if executed alone. One difference stems from the *rope* interface rather than the model. A Pharo step is constructed with its target and receives the model from its composite. In *rope*, a step names the parameter it adds or removes but not the method it applies to. The composite must then supply the target as well as the program model before the preparation hook runs.

Like a Pharo composite, the *rope* composite can mix both behavioral levels. Adding and removing a parameter can break behavior at call sites, so they are composed at the refactoring level. Reordering cannot, so it is composed at the transformation level.

A view of pending changes. Checking each step against the state its predecessors produced requires a view of the program that includes the changes made so far while nothing is yet written. Pharo has such a view as an existing abstraction. *Rope* does not, but one can be built from parts the engine already has. Pending source can be analyzed without being on disk, and name resolution can be pointed at it. Every intermediate state must then parse. The engine's own name resolution must see the pending changes. Otherwise, the analyses of other modules would still resolve against the old program.

Each step constructs its change into the view, which is discarded when the sequence ends. Construction therefore precedes the behavior-preserving checks, the reverse of the Pharo driver's order. A step's checks concern the call sites its rewrite met, and those are known only once the rewrite has run. The guarantee is unchanged: changes are constructed but never applied until the policy has seen the warnings. In both realizations the policy decides once, on the diagnostics the composite collects from its steps, as Section 3.1 states, rather than at each step.

What the composite exposed in the host engine. The engine's existing change-signature entry point now runs the composite, and on a few inputs it behaves differently from upstream *rope*. Most of the differences come from applicability preconditions that are missing in the original engine. The reified preconditions of the steps make them explicit: an invalid index was a silent no-op or a bare index error and is now refused. One difference is a defect in the rewrite itself. Executing the steps in sequence removes it. These differences surfaced from the retrofit alone, with no test written to look for them, and are now pinned by tests. The module's pre-existing tests are unaffected.

Generalizing beyond the proof of concept. The model presented in Chapter 3 is framed in terms of roles, so other engines can instantiate it. Table 5.7 lists the engine capabilities that these roles require in the Pharo implementation and the Python proof of concept. This mapping specifies what an engine must provide. The roles transfer, but the analyses behind them, such as name lookup and call-site resolution, must be implemented anew in each engine. The two realizations also operate on different representations. Pharo works with a live image, while *rope* operates on source files with static name resolution. The roles transferred across this difference, so they do not rely on a live image.

Table 5.8 shows how the *rope* program model resolves the checks for the two operations. Some facts are established, some are valid only within the configured project scope, and others

remain uncertain and are reported as warnings. A transfer study must therefore clarify which facts the target engine can establish, the level of granularity at which it can reuse and compose operations, and whether it can provide the view of pending changes that composition over intermediate states requires.

Table 5.7: Required engine capabilities in the Pharo implementation and the scoped Python proof of concept. The Pharo column is realized across the migrated families, and the Python column covers Rename Method and Change Signature with representative checks.

Required engine capability	**Pharo realization**	**Python PoC**
Program model (semantic queries)	Live image program model	File-based project model with name resolution
Change model (source-preserving, deferred)	Reified changes applied to the image	Deferred change sets applied through the engine's application boundary
Applicability predicate (A_τ)	Reified preconditions	Reified precondition predicates over the program model
Behavior-preserving diagnostics (BP_R)	Reified preconditions that report warnings	Representative checks; warnings for uncertain facts
View of pending changes (readable by later steps)	Change-scoped overlay of the live image	Absent as an abstraction; assembled from existing parts and installed into the engine's resolution path
Interaction separation (drivers)	Driver objects over the polymorphic API	Headless driver policies; no IDE adapter or contextual choices
Execution control (for testing)	Headless apply, test, and undo	Headless apply, test, and undo through the engine's history

Observed consequences and scope. The architecture prescribes that decisions regarding failed behavior-preserving checks are left to the caller, but it does not define the mechanism for making that decision. Pharo employs resumable warning signals, while the *rope* proof of concept records violations as data and lets a driver policy reject or proceed. Without a driver, a script receives them as an exception it can catch but not resume. The decision protocol is therefore transferable, while the mechanism remains engine-specific. The motivation also differs: in Pharo the layer protects a live image, while in *rope* it makes every warning and the decision on it explicit. The two realizations otherwise differ only in what the model leaves engine-specific: the concrete analyses behind individual preconditions and the granularity of composite steps. The remaining limits are ones of scope: the checks are representative, not exhaustive, and the work does not include IDE driver integration, contextual choices, or a Refazz port.

Table 5.8: Behavior-preserving checks implemented in the two *rope* operations and how the program model resolves them.

Check	Outcome	Basis
New name already defined in the hierarchy	Established	Attribute lookup over the class and its ancestors
Old signature remains in the hierarchy	Established	Definition scan over the analyzed resources
Removed parameter still read in the method body	Established	Scope resolution of the parameter's occurrences in the body
Existing call lacks an added required argument	Resolved calls	Recorded call sites, all flagged when the parameter has neither a default nor a value
Explicit argument of a removed parameter dropped	Resolved calls	Reconstruction of each recorded call through the preceding edits
Possible clients outside the configured file set	Scope-bound	Detectable relative to the selected project files; unknown outside them
Duck-typed or unknown receiver	Warning	Receiver cannot be resolved by the program model

5.7 Threats to Validity

Migration completeness. The evaluation is limited by the ongoing migration of the Pharo engine. Beyond the wrappers counted at the start of this chapter, other families have yet to be fully migrated to the decorator and driver patterns. The migration of individual checks and direct signaling sites is still ongoing.

The results are therefore claimed for the migrated families rather than for the engine as a whole. The remaining methods that signal directly are counted explicitly, so the reader can see where the migration stops.

Indirect measurements. The clone-detection analysis uses syntactic duplication reported by jscpd. It does not establish semantic equivalence between clone fragments, and it relies on a single tool. We therefore inspected the clones themselves. Table 5.5 shows that most legacy clones are structural twins between refactorings and transformations, not tokenizer artifacts.

The interaction measurements are image-wide static counts of signaling methods, and the case-study metrics do not measure behavior or end-user interaction directly. A refactoring that decorates a transformation is a client of that transformation. One of the 28 clients of Add Method and one of the 14 clients of Remove Method are therefore the refactoring wrappers themselves. The migration introduced those two clients, so they do not count as independent reuse. Duplication percentages, reuse counts, signaling counts, and the Extract Method refactoring case study are consistent with each other, so the conclusions do not depend on a single metric.

Attribution of the improvements. The engine also changed between Pharo 11 and Pharo 14 for reasons other than this migration. Part of the duplication reduction may therefore come from those changes.

Two observations limit this threat. The measurement covers only the packages that the new architecture changes directly, that is, the packages that contain the refactorings, the transformations, and their preconditions. Table 5.5 shows that the clone families which disappear are those of the operations migrated to the polymorphic API. Of the 19 pairs that remain, 13 belong to Add Parameter and Abstract Variables, which are not yet migrated.

Cross-language evidence. The *rope* proof of concept covers two operations with selected behavior-preserving preconditions. It does not establish broader reuse across Python refactorings, nor a comparable migration effort. The cross-language claim is therefore formulated at the level of roles and contracts for the two transferred operations. Section 5.6 reports the tests and the public artifact against which the scope of the transfer can be checked.

5.8 Discussion and Summary

The results in this chapter concern modularity, reuse, and execution structure. Chapter 7 evaluates behavioral correctness across all seven studied refactorings, including Composite Extract Method.

H1 (Architecture & Reuse): supported within the scope of the migration. Section 5.1 gives the measurements (Tables 5.2 and 5.4). The clone-detection evidence further suggests that the engine-wide precondition split contributes more to the duplication reduction than the decorator wrappers alone. Once behavior-preserving checks leave the transformations, near-identical refactoring/transformation twins can collapse. Decoration is so far the smaller effect, and it can grow as more refactorings delegate their applicability preconditions instead of re-declaring them. The *rope* proof of concept (Section 5.6) adds preliminary evidence that the separation is not Smalltalk-specific.

H2 (Execution & Decoupling): supported within the implementation scope. Section 5.5 gives the evidence. The number of methods that signal warnings fell by 46%, and the number that signal errors or failures fell by 40%. None remain in the migrated classes examined here.

The cost of the migration. The migration was demanding in time and effort. Separating applicability from behavior-preserving preconditions could not be automated: each precondition had to be analyzed on its own, against the transformations and preconditions it interacts with. Writing a driver likewise requires enumerating the paths a user may take when a check fails. A catalog of these recurring decisions would be a useful next step for engine developers.

Chapter 6

Refazz: Metamorphic Testing of Refactoring Engines

The previous chapters established a modular architecture for refactoring engines, but structural improvements alone do not guarantee correctness, especially in dynamically typed languages. This chapter presents Refazz, a metamorphic testing approach for refactoring engines that combines real-project testing, refactoring parameter exploration, forced execution, and replay during outcome classification. Replay is supported by MuTalk, Pharo's mutation-testing framework. Refazz applies refactorings to real-world projects. Project tests identify observed behavior changes, while exceptions and terminated executions provide additional anomaly signals. MuTalk replay supports the subsequent diagnosis and deduplication of those anomalies.

The method addresses **RQ3** by examining what evidence project tests provide about behavior preservation. It addresses **RQ4** by exercising valid, boundary, invalid, and context-dependent configurations. It also forces execution past applicability rejections and behavior-preserving warnings. This chapter defines the method and its implementation. The following chapter evaluates it.

6.1 Challenges of Testing Refactoring Engines

Testing a refactoring requires an input program, the parameters that configure the transformation, and an oracle for deciding whether the result is acceptable. These concerns interact: realistic program structure determines which semantic cases are reached, parameter choices determine which transformation and precondition paths execute, and the available oracle determines which failures can be observed.

6.1.1 Terminology

This chapter uses the vocabulary fixed in Section 2.3. A violated behavior-preserving precondition is reported as a warning that the caller may override, while a violated

applicability precondition stops the attempt [ACD+22]. The method therefore records, for every attempt, whether it proceeded and which kind of precondition, if any, failed.

Three further terms recur in the rest of the chapter. *Tests* are the existing tests of the project to which a refactoring is applied. Refazz does not write or modify them but runs them before and after the change. *Parameters* are the data that configure a particular refactoring, such as a new method name, a target class, or a selected source interval. They are not the inputs of the program under test. *Defects* are defects of the refactoring engine, not of the project. The project is only the input on which the engine shows its behavior.

6.1.2 Testing Refactorings by Example

Listing 6.1 shows a hierarchy in which ClassA defines methodA, ClassB defines exampleMethod, and ClassC calls the inherited implementation through super. Extracting self computeValue from exampleMethod into a new method also named methodA creates an override in ClassB. The super send in ClassC»methodA then reaches the newly extracted method instead of ClassA»methodA, so behavior changes. A refactoring engine should warn before applying this semantically risky transformation. Refazz reached this case by generating a method name that the superclass already implements, and the engine applied the extraction without a warning (#17235, Appendix C).

```
ClassA subclass: #ClassB
ClassB subclass: #ClassC

ClassA≫methodA
    ^ 42

ClassB≫exampleMethod
    | aValue |
    aValue := self computeValue.
    self log: aValue.
    ^ aValue

ClassC≫methodA
    ^ super methodA + 1

"After extracting `self computeValue' as `methodA':"
ClassB≫exampleMethod
    | aValue |
    aValue := self methodA.
    self log: aValue.
    ^ aValue

ClassB≫methodA
    ^ self computeValue
```

Listing 6.1: Extract Method should warn before creating an accidental override.

6.1.3 Key Challenges

The oracle problem. Existing approaches often use compilation as the primary success criterion [GBL+13]. Parsing and compilation can reject structurally invalid generated changes, but they cannot establish behavior preservation. In a dynamically typed language, binding, dispatch, and control-flow defects may remain syntactically valid and become visible only during execution. Project tests provide a semantic oracle for the behavior they cover. They can detect behavior changes when the affected behavior is exercised and asserted by the suite.

Program diversity and semantic coupling. Synthetic generators can produce structurally varied abstract syntax trees [DDGM07,SGM13,DHT10]. Real projects contain class hierarchies, implicit method dependencies, non-local returns, reflective behavior, and shared variables that are difficult to synthesize. Such coupling is necessary to trigger cases like the accidental override in Listing 6.1.

Configuration and input space. Refactorings require parameters such as a new selector, a target class, or a source interval. Valid parameters exercise accepted transformations, while invalid, boundary, and context-dependent parameters exercise rejection and warning logic.

Dynamic-language uncertainty. Without complete type information, the engine sometimes cannot determine statically whether a change is behavior-preserving. For example, renaming a commonly implemented selector may affect unrelated receivers. The engine must communicate such uncertainty through warnings.

6.2 The Refazz Approach

Refazz formulates one core metamorphic relation for executions that the engine accepts without a warning: a refactoring should leave the selected project tests passing. Parameter strategies and four execution modes, MR1 to MR4, vary how configurations reach the transformation. Figure 6.1 gives an overview of the approach.

Let $P \in \mathcal{P}$ be a project, S the selected project tests, and $T_S(P) \in \{\text{PASS}, \text{FAIL}\}$ the outcome of running S on P. The outcome is PASS when every test in S passes and FAIL otherwise. Each attempt begins from a passing baseline, $T_S(P) = \text{PASS}$. Let π be the engine's normal policy, which aborts on any warning. For a refactoring R and an accepted configuration $(P, \theta) \in \text{dom}(exec_{R,\pi})$ with result $P' = exec_{R,\pi}(P, \theta)$, the relation is

$$T_S(P') = \text{PASS}.$$

Refazz selects the tests in S before the operation is applied and does not transform them. A refactoring may itself update references in test code, as Rename Class does. The same tests run against P and P'. Because the baseline passes, a new test failure after application is direct evidence of a behavior change observed by S.

The modes determine which boundary, if any, is exercised before observation. The two boundaries are the applicability rejection and the behavior-preserving warning. MR1 and MR2 execute configurations under the engine's normal checks and test the core relation when the

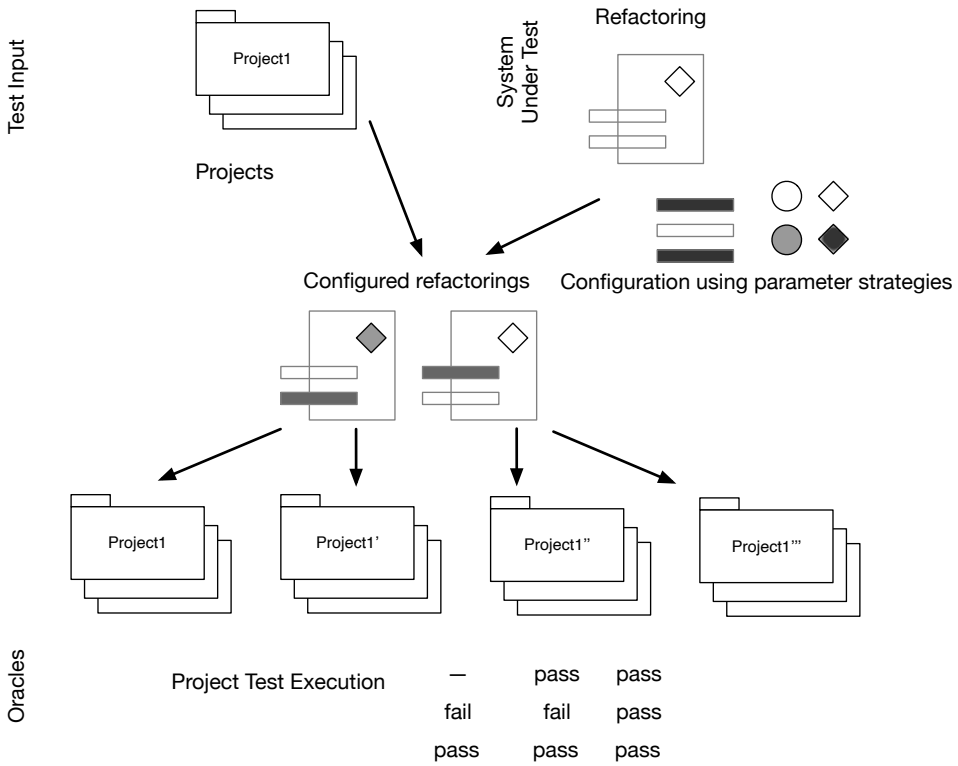

Figure 6.1: The Refazz approach: parameter strategies configure refactorings on real-world projects, and project tests provide a semantic oracle.

refactoring is applied. MR3 and MR4 force execution past one of these boundaries. They use the same project-test observations as diagnostic evidence rather than as assertions of preservation. Keeping the modes separate lets Refazz distinguish normal execution, applicability rejection, and warning paths during analysis.

6.2.1 Core Relation and Execution Modes

Each attempt records P, S, the operation R, and its parameters θ. An attempt is *rejected* when an applicability precondition stops it before transformation execution. It *stops after a warning* when a behavior-preserving precondition reports a warning and the attempt does not continue past it. It *terminates* when execution starts but does not produce P'. It *completes* when execution produces P'. Termination covers two cases: execution raises an exception, or it stops at a warning reached during execution rather than at the checking boundary.

We use $\Theta_{valid} \subseteq \Theta_\tau$ for the parameters of configurations intended to satisfy all preconditions, and $\Theta_{explore} \subseteq \Theta_\tau$ for the parameters of boundary, invalid, or context-dependent configurations. The "MR" prefix of the mode labels is retained for continuity with the experiment labels. The four labels denote modes, not four separate metamorphic relations.

In the terms of Section 3.1, MR1 and MR2 execute under the engine's normal policy π, which aborts on any warning. They differ only in the parameters they supply. MR3 sets aside the check in A_τ that rejected the configuration and otherwise executes under the normal policy. MR4 fixes $\pi \equiv \mathsf{proceed}$, so execution continues past a reported warning.

1. **MR1: Valid input.** For $\theta \in \Theta_{valid}$, the refactoring should apply without rejection or warning and produce P' with $T_S(P') = \mathrm{PASS}$. Terminated executions and failing tests enter triage as defect candidates. Most are transformation defects, but a misplaced check that is reached only during execution also surfaces here.

2. **MR2: Input exploration.** For $\theta \in \Theta_{explore}$, the engine should reject the configuration cleanly, stop after a behavior-preserving warning under the normal policy, or apply the refactoring with $T_S(P') = \mathrm{PASS}$. A warning is an expected outcome when the configuration is structurally applicable but carries a detected behavior risk. Exceptions or termination during configuration, checking, or application; failing tests; and responses inconsistent with an expected warning enter triage as defect candidates.

3. **MR3: Forced execution after rejection.** For a configuration rejected by an applicability check, Refazz skips that check and attempts the transformation. A terminated forced execution does not, by itself, determine whether the rejection was justified. Triage distinguishes an input outside the transformation's construction domain from a transformation defect or another missing check. A completed result supports investigating whether the applicability check is overly conservative. Failing tests can then indicate that the structurally constructible change still needs behavior-preserving protection.

4. **MR4: Forced execution after warning.** For a configuration that triggers a behavior-preserving warning, Refazz proceeds despite the warning. Failing tests may expose the warned behavior risk when the suite covers it. They may also result from a transformation defect, so triage inspects them before attributing them to the warning. Passing tests are inconclusive because project tests observe only covered behavior. Termination suggests that the warning may be masking a missing applicability precondition, a transformation defect, or a precondition placed at the wrong level.

Table 6.1 summarizes these interpretations.

6.2.2 Parameter Exploration Strategies

Refazz uses refactoring-specific parameter strategies derived from precondition semantics. Most strategies are deterministic, such as generating reserved keywords, fresh identifiers, or existing identifiers. A limited number make stochastic selections, such as choosing a method from a superclass. Repeated runs are therefore largely reproducible, although these choices can change the AST nodes and precondition paths that a run reaches. Each attempt records its target, strategy, and generated inputs, so the same attempt can be executed again during triage.

Table 6.1: Execution modes and triage interpretation.

Mode	Target	Expected behavior	Triage signal
MR1	Valid configuration	Apply the refactoring and preserve the project's test outcome	Termination or failing tests
MR2	Exploratory configuration under normal checks	Reject cleanly, stop after a behavior-preserving warning, or apply the refactoring and preserve the project's test outcome	Unexpected termination, failing tests, or an inconsistent response to an expected warning
MR3	Configuration forced past applicability rejection	Termination is the ordinary outcome, since the skipped check usually guards an input outside the construction domain	A completed result is the signal to inspect the rejection for over-conservatism
MR4	Configuration forced past behavior-preserving warning	Produce P'; tests may expose the warned behavior risk; passing tests are inconclusive	Failure to produce P' suggests an applicability defect, transformation defect, or wrong-level precondition

The strategies form three families:

- *Name strategies* generate class, trait, method, and variable names. They include syntactically invalid names, reserved names, fresh valid names, existing names, wrong-arity selectors, and names that shadow or override declarations in the surrounding hierarchy.
- *Parameter-permutation strategies* retain or change the order and correspondence of method parameters.
- *Source-interval strategies* generate source selections for operations such as Extract Method, including valid expressions, empty intervals, shifted boundaries, and partial structures.

When a refactoring has multiple parameters, Refazz varies one strategy family at a time and supplies valid values for the remaining parameters. This avoids multiplying unrelated invalid inputs and makes failures easier to attribute. For one Extract Method target, valid-name strategies produce MR1 attempts, and invalid and context-dependent strategies produce MR2 attempts. The configurations that the engine rejects or warns about serve as MR3 and MR4 attempts.

6.3 Implementation and Analysis Pipeline

We implemented Refazz by extending MuTalk[1] and representing configured refactorings as mutation operators. MuTalk supplies project traversal, coverage-guided test selection, mutation execution and isolation, outcome recording, and the retained mutation object used for replay. Refazz supplies the refactoring target, parameter strategy, generated inputs, and mode-specific precondition policy that each mutation represents.

For each potential target, the runner performs the following cycle:

[1] https://github.com/pharo-contributions/mutalk/

1. **Establish the baseline.** Select the tests covering the classes or methods that the operation will change, run them before the change, and retain only a passing baseline.

2. **Configure.** Generate a concrete operation and parameter tuple from the selected target and parameter strategy.

3. **Check or bypass.** Enforce all checks in MR1 and MR2, bypass a failed applicability check in MR3, or proceed past a behavior-preserving warning in MR4.

4. **Apply and observe.** Execute the operation and record its outcome (Section 6.2.1). When P' is available, rerun the selected tests and compare their outcomes with the baseline.

5. **Record and restore.** MuTalk retains the evaluated mutation, selected tests, test outcomes, coverage, and exception information. The Refazz mutation retains the target, parameters, strategy, mode-specific precondition policy, and the observed precondition outcome. The runner then undoes the change before the next attempt.

The apply-test-restore cycle isolates attempts so that one transformation cannot affect later ones.

6.3.1 From Attempts to Confirmed Defects

The runner produces attempt records, not defect reports. The analysis reduces those records in the following steps:

attempts → anomaly groups → inspected representatives → confirmed unique defects.

First, records with the same refactoring, mode, strategy, failure signature, relevant input characteristics, and, for terminated attempts, exception type are grouped into one anomaly group. This prevents repeated executions of the same underlying problem from inflating the result. A representative of each group is then inspected, and replayed from its stored configuration where that configuration was retained. The analyst inspects the precondition outcome, generated change, exception, and before-and-after test outcomes. The archived exports retain the raw attempt totals but not the intermediate numbers of anomaly groups or replayed representatives. Chapter 7 reports raw outcomes and confirmed unique defects only.

A candidate enters the defect catalog only after replay or equivalent retained evidence establishes a concrete failing scenario, attributes it to the refactoring engine, and distinguishes it from issues already represented in the catalog. The detection signal is the observation that first exposed the candidate, such as a test failure, exception, termination, or successful execution past a rejected check. Replay, source inspection, and issue records provide supporting evidence and identify the root cause. The confirmed defect is then classified as a transformation defect or a precondition defect. Precondition defects are further classified as missing, overly conservative, or misplaced. Chapter 7 applies this pipeline and reports the resulting catalog.

Table 6.2: Positioning of Refazz among approaches that test a refactoring engine by applying refactorings and checking outcomes.

Approach	Input source	Input variation	Precondition handling	Outcome checks and analysis
ASTGen (Daniel *et al.*) [DDGM07]	Generated (Java, static)	Program shapes (ASTs)	Default	Differential and structural checks
SafeRefactor (Soares *et al.*) [SGSM10, SGM13]	Generated; also real projects (Java, static)	Program shapes and tests	Default	Generated tests
Disabling Precond. (Mongiovi *et al.*) [SMG11, MGS+18]	Generated (Java, static)	Program shapes	Selected checks bypassed	Generated tests
Drienyovszky *et al.* [DHT10]	Generated (Erlang, dyn., funct.)	Program shapes from an attributed grammar	Default	Execution comparison of selected functions on generated arguments
Gligoric *et al.* [GBL+13]	Real projects (Java, C; static)	Fixed task set per method, including signature parameters	Default	Compilation and exceptions; clustering and inspection
Li & Thompson [LT07]	Real programs (Erlang, dyn., funct.)	Random commands; locations from the AST, conflict-seeking names	Default	Compilation and binding properties
Refazz	Real projects (Pharo, dyn., OO)	Refactoring-specific parameter strategies: valid, invalid, boundary, context-dependent	Default, plus forced modes	Project tests and engine outcomes; replay and classification

6.4 Positioning Against Prior Testing Approaches

Prior approaches that test an engine by applying refactorings and checking outcomes are reviewed in Section 2.6.5. Refazz connects real-project inputs to refactoring-specific parameter strategies. It runs the resulting configurations in normal execution and in forced execution past an applicability rejection or a behavior-preserving warning. It combines project-test and engine outcomes with replay and classification to identify transformation defects and missing, misplaced, or overly conservative checks. Parameter-level generation is not new: Li and Thompson generate refactoring arguments over real programs [LT07], and Gligoric *et al.* vary method signature parameters through a fixed task set [GBL+13]. The contribution of Refazz is the combination of real-project inputs, parameter strategies, forced execution, and replay-based classification. Table 6.2 summarizes the input, execution-control (precondition handling), and outcome-analysis dimensions.

6.5 Summary

Refazz tests its core metamorphic relation for accepted refactoring executions and uses forced modes to diagnose applicability rejections and behavior-preserving warnings. Parameter strategies generate concrete configurations, while the implementation records and restores

each attempt. The analysis pipeline groups anomalous records and inspects a representative of each group. Only confirmed unique defects enter the catalog reported in Chapter 7.

Chapter 7

Refazz Evaluation

This chapter evaluates Refazz on Pharo's refactoring engine. It describes the subject projects, selected refactorings, run configuration, and analysis procedure, then reports the results for **RQ3** and **RQ4**. Across seven studied refactorings and five projects, Refazz executed 255,699 attempts and confirmed 19 unique refactoring engine defects: 12 transformation defects and 7 precondition defects.

7.1 Experimental Setup

The engine under test was the refactoring engine shipped with *Pharo* 13, which also served as the experimental environment.

7.1.1 Project Selection

We used purposeful sampling rather than random project selection. A subject had to be an established Pharo project, load reproducibly in the pinned environment, have a passing non-trivial test suite, and provide refactoring targets across classes and methods. It also had to exercise relevant features such as inheritance, class-side behavior, blocks, non-local returns, cascades, assignments, literals, and name binding. We selected projects from different domains and of different sizes to avoid tuning the method to one codebase. Table 7.1 reports the selected production packages, excluding tests and dependencies. *Project coverage* is the portion of a subject project's code exercised by its own tests and therefore bounds the behavior observable to the semantic oracle. Method and node coverage count covered methods and AST nodes, respectively.

Coverage ranged from 60% to 87%. The study examines whether the available tests expose behavior changes.

7.1.2 Selected Refactorings

We selected common refactoring families [CHLN06]: adding arguments, extracting and inlining methods, and renaming program elements. The seven studied refactorings were

Table 7.1: Subject projects and coverage of the production packages included in the evaluation.

Project	Method	Node	Packages/classes	Domain
Artefact	66%	64%	4/109	PDF generation
AVL	63%	70%	4/6	Data structure
Graph algorithms	85%	85%	4/34	Graph algorithms
Microdown	60%	63%	2/90	Markdown
Soup	87%	82%	6/12	HTML scraping

Table 7.2: Execution summary across tested projects.

Project	Attempts	Time
Soup	39,720	52m 58s
Graph algorithms	56,877	52m 38s
Artefact	73,934	1h 08m 43s
AVL	24,291	19m 19s
Microdown	60,877	1h 24m 21s
Total	**255,699**	

Add Argument, Composite Extract Method, Extract Method, Inline Method, Rename Variable, Rename Class, and Rename Method. Composite Extract Method is the composite implementation of Extract Method introduced in Section 5.2. It realizes Extract Method as a sequence of reusable transformations. At the time of the evaluation it had not replaced the legacy implementation, which we evaluated separately as Extract Method.

7.1.3 Run Configuration and Test Selection

We ran all studied refactorings and applicable strategies on the five projects, for 255,699 attempts in total. Each refactoring had a five-minute timeout per parameter strategy, execution mode, and project. A timeout could reflect either a large configuration space or expensive transformation logic, not necessarily a defect. Thirty timeouts occurred among the 180 combinations of refactoring, parameter strategy, execution mode, and project, none of them on AVL. Each fell in a different one of the 140 combinations of refactoring, mode, and project. Twenty involved Extract Method and Composite Extract Method, which enumerate AST nodes and statement subsequences. Eight involved Inline Method and Rename Variable under MR3. The remaining two occurred on Microdown, for Rename Method under MR4 and Add Argument under MR3. Full-project runs took between approximately 19 minutes and 1 hour 24 minutes (Table 7.2).

The evaluation uses the MuTalk-based implementation described in Chapter 6, including its coverage-guided test selection. During triage, we reran the full project test suite for attempts with observed behavior changes and for attempts that continued after behavior-preserving warnings. These reruns did not change the reported classifications.

Table 7.3: Technical taxonomy of the 19 confirmed defects by affected concern.

Technical concern	Count	Affected refactorings	Primary outcomes
Name, scope, and binding	1	Composite Extract Method	Terminated execution
Control flow and returns	5	Composite Extract Method, Extract Method	Three test failures; two terminated executions
Structural and syntactic validity	9	Add Argument, Composite Extract Method, Extract Method	Three test failures; six terminated executions
Precondition precision and placement	4	Extract Method, Inline Method, Rename Class	Test failure, terminated execution, or completed forced execution
Total	**19**		

7.1.4 Analysis Procedure

We applied the analysis pipeline of Section 6.3.1. MuTalk's *alive*, *killed*, and *terminated* labels describe raw attempts: tests passed, tests failed, or execution did not complete. We interpreted these labels relative to the execution mode and grouped anomalies by refactoring, mode, strategy family, failure signature, relevant input characteristics, and, for terminated attempts, exception type.

A representative configuration from every group was inspected, and MuTalk replay was used for most groups to execute the retained refactoring mutation again. Replay supported diagnosis, construction of reproducing examples, and deduplication. We then assigned one primary root-cause class to each defect. Multiple manifestations of one cause count once. Secondary characteristics are reported where they help explain a defect, but they do not create another catalog row or enter the subtype totals.

The modes do not assign defect classes automatically. In particular, an MR3 rejection is classified as overly conservative only when forced execution completes and inspection confirms that the structural change is feasible. An MR2 execution can also expose an overly conservative check. Normal checking accepts the configuration, and execution then fails on a hard restriction that should instead be an overridable behavior-preserving warning. Appendix C provides the 19-row audit trail used for all counts.

7.2 Results

Refazz confirmed 19 unique defects across five of the seven studied refactorings. Rename Variable and Rename Method yielded none. Six confirmed defects were found in Extract Method, ten in Composite Extract Method, and three in the other studied refactorings.

7.2.1 RQ3: Project Tests and Outcome Checks

We first classify each confirmed defect by the technical concern it affects. Table 7.3 reports this taxonomy together with the primary observed outcomes. The taxonomy is orthogonal to whether the root cause is a transformation or precondition defect.

Name, scope, and binding. One defect concerned extraction in an incorrect class-side binding context. Its original execution terminated. Replay and inspection established the binding error.

Control flow and returns. Five extraction defects concerned blocks, non-local returns, return propagation, or statement replacement. Project-test failures exposed three of these defects. Two others first appeared as terminated executions. Replay then made the failures attributable to specific transformation steps.

Structural and syntactic validity. Nine defects affected structural input handling or change construction. Six produced invalid ASTs, unformattable source, parser failures, or exceptions and terminated before producing a completed result. Project-test failures exposed the other three: extracting the left side of an assignment, changing the contents of a literal array, and introducing recursion after extracting `super`.

Precondition precision and placement. Four defects concerned checks that rejected feasible transformations or were placed in execution-time logic. The misplaced check in Extract Method, an override check, was exposed by a project-test failure. It was hidden in optional name handling rather than evaluated as a behavior-preserving precondition. The misplaced check in Inline Method first appeared through a terminated execution and a late warning, and has been fixed in Pharo. The overly conservative Extract Method check was attributed to MR3 because forced execution completed and inspection confirmed that the transformation was feasible. Rename Class issue #18240 was exposed under MR2 because its shared-variable configuration passed normal checking before execution failed. Appendix C explains why the catalog counts it as overly conservative.

Three of the seven project-test failures occurred in Composite Extract Method (#16882, #18401, #18402) and four in Extract Method (#16173, #16174, #16175, #17235).

Answer to RQ3. Project-test failures exposed seven of the 19 confirmed defects (37%). Of the remaining twelve defects, eleven first appeared through terminated executions, and one through completed forced execution followed by manual inspection. This supports **H3 (Semantic Oracles)** within the observed scope: existing project suites acted as partial executable behavioral specifications in seven cases, while other outcome checks exposed the remaining twelve defects.

7.2.2 RQ4: Parameter Exploration and Forced Execution

Table 7.4 reports the execution mode to which each unique confirmed defect was attributed under the same projects, refactorings, and triage process. MR1 represents valid-input testing. MR2 adds invalid, boundary, and context-dependent configurations under normal checks, while MR3 forces configurations past applicability rejections. MR4 forces execution after behavior-preserving warnings.

Valid-input testing surfaced 13 defects: 12 transformation defects and one misplaced behavior-preserving check. Parameter exploration found three missing preconditions, one

Table 7.4: Mode attribution of unique confirmed defects.

Mode	Transformation	Precondition	Total
MR1: valid-input	12	1	13
MR2: input exploration	0	5	5
MR3: forced execution after rejection	0	1	1
MR4: forced execution after warning	0	0	0
Total	**12**	**7**	**19**

overly conservative behavior-preserving restriction, and one misplaced behavior-preserving check. Forced execution after applicability rejection found one overly conservative check. MR4 continued execution after a behavior-preserving warning. It recorded whether the structural change could be completed and whether project tests exposed the behavior risk reported by the warning. Its 3,450 attempts produced 898 test failures and 83 terminated executions (Appendix D).

In these runs, the anomalies were linked to defects that had already been attributed to other modes. An applicability rejection can be refuted by a single completed forced execution. A warning instead reports a behavior risk that a passing run cannot rule out, as the tests cover only part of the program's behavior. In the warning continuations examined here, replay reconstructed the reported risk and, in most cases, identified a scenario that justified it. As a result, MR4 did not contribute any confirmed defects to the catalog.

Strategy contribution. The name/shared-variable strategy exposed the overly conservative Rename Class restriction described above. The inherited-selector strategy exposed the misplaced override check in Extract Method, whose configuration is a valid selector taken from an ancestor. Parameter-permutation strategies exposed the missing Add Argument check. Source-interval boundary strategies exposed missing extraction checks and rejected extraction cases that forced execution showed to be feasible. The confirmed-defect catalog maps every issue to its mode, strategy family, detection signal, defect class, and taxonomy.

Answer to RQ4. Parameter exploration exercised precondition paths that valid configurations did not reach. Three missing checks, one overly conservative check, and one misplaced behavior-preserving check were uniquely attributed to MR2, and one overly conservative check was uniquely attributed to MR3. Together, these account for 6 of the 19 confirmed defects (32%). This supports **H4 (Boundary Verification)** within the studied modes and subjects: invalid, boundary, and forced-execution configurations complement valid-input testing. MR4 added no confirmed defect within the studied warning continuations.

7.3 Infrastructure Findings

The 19 counted defects are refactoring engine defects. Refazz also exposed problems in neighboring infrastructure. In issue #17259, Refazz forced Rename Variable to execute after

its preconditions rejected an invalid argument name. The compiler accepted the resulting method programmatically. We reported this compiler regression but did not count it as a refactoring defect.

During tool development, the apply-test-undo cycle exposed a defect in the undo mechanism. Because a faulty undo could affect later attempts, we fixed it before collecting the reported data. It is therefore an infrastructure finding rather than one of the 19 experimental defects.

7.4 Discussion

Transformation and precondition testing. A refactoring engine can accept a configuration and transform it incorrectly, accept an unsafe configuration, or reject a feasible one. The results include all three cases. Parameter exploration reaches precondition and transformation paths, project tests and terminated executions expose different manifestations, and MuTalk replay supports diagnosis after either signal.

Extraction defect concentration. Extraction-style refactorings account for 16 of the 19 defects in this sample. Their failures span binding, control flow, returns, and structural validity. This concentration motivates broader study, but the present sample cannot establish a correlation between implementation complexity and defect density.

The two extraction implementations. Composite Extract Method accounts for more of these defects than Extract Method, although it is the smaller implementation (Section 5.2.3). The two had different levels of exposure. The legacy operation has been in daily use since the Refactoring Browser [RBJO96, RBJ97], while the composite had not yet replaced it when the study ran. The defects of the composite concern source-interval handling, structural validity, and control flow (Appendix C). These are extraction cases that the legacy implementation had come to handle over time, and none pertains to the composition mechanism itself.

Relation to prior testing approaches. Section 6.4 compares the method's design with prior testing approaches. The evaluation shows that the outcome checks play complementary roles in the studied engine. Project-test failures reveal covered behavior changes, terminated executions reveal failures before the test oracle can run, and completed forced execution can enable inspection of a rejected configuration's feasibility. None of these signals alone characterizes refactoring correctness.

Practical integration. Pharo already runs a dedicated refactoring test suite and performs manual UI testing during releases. Refazz is intended for on-demand release hardening or targeted runs after a refactoring implementation changes. Maintainers can restrict the refactoring and strategy set, then replay and deduplicate anomalous configurations. Six integrated fixes show that maintainers acted on the reports.

7.5 Threats to Validity

Manual classification. Grouping raw attempts and assigning root causes requires human judgment, and different analysts might partition the attempts in various ways.

Attempts were therefore grouped by stable execution characteristics. A representative of each group was inspected, and replay or equivalent retained evidence was used to confirm the observed behavior. Every counted defect is linked to a public tracker issue, and a row-by-row confirmed-defect catalog accompanies the results. Misclassification remains possible, but the catalog lets the reader re-derive or contest every count.

Nondeterministic tests. An intermittently failing test would appear as a change in observed behavior.

However, triage would catch such failures. Each attempt begins with a passing baseline. Every counted defect is reproduced through replay or equivalent retained evidence and is reported with a concrete failing scenario. The raw failure counts in Appendix D are documented as attempt outcomes rather than as defects. No intermittent failure was observed during triage.

Incomplete oracle and false negatives. Passing tests do not prove behavior preservation. Coverage-guided selection, weak assertions, timeouts, and failures that do not reach the test oracle can all hide defects.

We reran the full test suite for every confirmed behavior-changing case, so the positive findings do not depend on the reduced oracle used during exploration.

Project and input selection. The projects were chosen because they load reproducibly, have non-trivial passing test suites, and exercise the language features the studied refactorings touch. Other projects may expose a different defect distribution.

We therefore diversified the sample: the five projects come from different domains and vary in size and test coverage. Results are reported per project, so project-specific effects remain visible.

Some invalid configurations at the API level would not arise in an IDE workflow. They remain relevant because scripts and tools call the engine directly. The claims are about API-level behavior, not about how often failures occur in interactive use.

Language and engine transfer. The implementation and evaluation focus on Pharo. Other languages offer different static checks and require different parameter strategies.

We separate the transferable workflow from the Pharo-specific results. The method is defined in terms of engine capabilities: a programmatic refactoring API, a source-preserving program and change representation, control over parameters and precondition handling, isolation or reliable undo, and projects with executable tests. Section 5.6 discusses how these capabilities correspond to other engines. The observed defect distribution is specific to the studied engine and should not be generalized directly.

Comparison baseline. There is no independent automated baseline for testing Pharo refactorings, so Refazz cannot be compared with an external tool.

The comparison is therefore internal. The mode-attribution analysis evaluates the modes within Refazz on the same projects and refactorings, with the same procedure and oracle. Each mode combines a set of configurations with a precondition policy, so the results are reported as attribution within this study, not as the separate effect of either. The effect of the configurations is separable in one case. MR1 and MR2 run under the same policy π and differ only in the configurations they supply, so the five defects attributed to MR2 follow from parameter exploration alone. MR3 and MR4 cannot separate the configurations from the policy, because forcing execution past a rejection or a warning requires a configuration that triggers it.

7.6 Evaluation Record

The evaluation used Refazz source commit `40f1b1f6` from github.com/guillep/refactoring-mutation and MuTalk commit `2ba13136` in the pinned Pharo 13 environment. The aggregate CSV exports of the attempts and the analysis scripts are part of the replication package archived on Zenodo [Šar26]. Appendix C provides the canonical 19-row defect catalog, and Appendix D reports aggregate attempt outcomes by mode for all five subject projects. Reported issues are tracked in Pharo's issue tracker under the *mutation impact* label.

7.7 Summary

In 255,699 attempts, Refazz confirmed 19 defects: 12 transformation defects and 7 precondition defects. Project-test failures exposed seven. Eleven other defects first appeared through terminated executions, and one was the completed MR3 over-conservatism case. Parameter exploration and forced execution accounted for six of the 19. Six fixes were integrated.

Chapter 8

Conclusion

The goal of this thesis was to improve reliability and reuse in refactoring engines for dynamically typed object-oriented languages.

The reference architecture addresses reuse by letting the caller select the behavioral level. Its roles and contracts were realized over two different program and change models. In Pharo, which works on a live image, the architecture ships with Pharo 14 for the migrated refactoring families, and the migration continues in the development branch. In *rope*, which works on source files, a scoped proof of concept introduced separation, decoration, and composition without replacing the host engine.

Refazz addresses reliability. Applied to seven refactorings over five projects, it confirmed 19 refactoring engine defects, and fixes for six of them have been integrated into the Pharo development branch.

The precondition split connects the two parts. It lets a caller set aside a behavior-preserving check while keeping the applicability checks, and it lets Refazz force execution past either an applicability rejection or a behavior-preserving warning, one at a time.

8.1 Answers to the Research Questions

RQ1. *How can a refactoring engine be architected to unify behavior-agnostic transformations and behavior-preserving refactorings into reusable, composable building blocks?*

The architecture promotes reuse by keeping behavior-preserving checks out of transformations. Each transformation includes only its structural change and the necessary applicability checks, so it can be reused as a building block. Refactorings decorate those transformations with behavior-preserving diagnostics. Composite operations sequence transformations and refactorings over intermediate program states.

The counts and clone analysis show that duplication fell where the migration introduced a reusable operation. Add Method and Remove Method are reused in the Pharo migration by 28 and 14 distinct refactorings and transformations, respectively. The duplicated-line ratio in the analyzed packages decreased from 6.02% to 1.52%. The case studies show that the resulting operations compose to form a simplified Extract Method refactoring and domain-specific

variants that replace or extend composed sequences. The gains concentrate on transformations that represent checked program operations, and not on the finer-grained AST wrappers, which remain internal helpers. These results support H1 within the scope of the migration.

RQ2. *How can user interaction be isolated from core refactoring logic while preserving both interactive guidance and headless batch execution?*

Interaction drivers separate user interaction from core refactoring execution. They collect parameters, resolve warnings, present previews, and let the user decide whether to apply a change. In the studied migration, the number of methods that send refactoringWarning: decreased from 28 to 15. The combined number for refactoringError: and the deprecated refactoringFailure: decreased from 163 to 98. The same configured core runs from both interactive and scripted entry points, and it provides the headless execution that Refazz uses. These results support H2 within the implementation scope.

RQ3. *To what extent can existing test suites from real-world projects act as partial executable behavioral specifications that expose behavior changes caused by refactorings?*

Applied to seven refactorings across five projects, Refazz executed 255,699 attempts and confirmed 19 defects. Project-test failures exposed seven of them (37%). Of the remaining twelve defects, eleven first appeared through terminated executions, and one through completed forced execution followed by manual inspection. Existing project suites therefore acted as partial executable behavioral specifications in seven cases, while other outcome checks exposed the remaining twelve defects. These results support H3 within the observed scope.

RQ4. *How do parameter exploration and forced execution past an applicability rejection or a behavior-preserving warning reveal missing or overly conservative refactoring preconditions that valid-input testing does not expose?*

Parameter exploration exercised precondition paths that valid configurations did not reach. Three missing checks, one overly conservative check, and one misplaced behavior-preserving check were uniquely attributed to MR2, and one more overly conservative check to MR3. These six of the 19 confirmed defects (32%) are all precondition defects. Valid-input testing uncovered the remaining 13. Parameter-permutation strategies exposed the missing Add Argument check. The name/shared-variable strategy revealed the overly conservative Rename Class restriction, and the inherited-selector strategy the misplaced override check in Extract Method. Source-interval boundary strategies exposed missing extraction checks and rejected extraction cases that forced execution showed to be feasible. Under MR4, the anomalies were linked to defects that had already been attributed to other modes. Passing runs could not rule out the behavior risk reported by a warning. MR4 therefore did not contribute any confirmed defects to the catalog. These results support H4 within the studied modes and subjects.

Table 8.1 lists the four hypotheses, the scope within which each is supported, and the sections that report the evidence.

Table 8.1: Assessment of the hypotheses and the location of the supporting evidence.

Hypothesis	Assessment	Evidence
H1 (Architecture & Reuse)	Supported within the scope of the migration	Section 5.1: reuse counts and clone analysis; Sections 5.2 and 5.3: the composite Extract Method and two domain-specific variants
H2 (Execution & Decoupling)	Supported within the implementation scope	Section 5.5: signaling counts, and one core operation invoked from a driver and from a script
H3 (Semantic Oracles)	Supported within the observed scope	Section 7.2.1: seven of 19 confirmed defects exposed by project-test failures
H4 (Boundary Verification)	Supported within the studied modes and subjects	Section 7.2.2: six confirmed defects attributed to parameter exploration and forced execution

8.2 Portability to Other Dynamic Environments

Section 5.6 lists the engine capabilities that the architectural roles require in the Pharo implementation and the Python proof of concept. The proof of concept is realized in *rope* for Rename Method and Change Signature.

The Refazz workflow introduces its own requirements: a programmatic refactoring API, a source-preserving program and change representation, control over parameters and precondition handling, isolation or reliable undo, and projects with executable tests. Python tools may fulfill parts of these requirements, but the strategies and oracles would need language-specific adaptation.

8.3 Scope and Limitations

The architectural results are claimed for the migrated families, not for the Pharo engine as a whole. Because the catalog migration is ongoing, the reported counts describe the state at the measured Pharo 14 commit.

The testing results are claimed for the studied configuration, which includes five projects, seven refactorings, the implemented strategies, and the behaviors covered by the project test suites. The 19 confirmed defects represent a lower bound for this configuration, and the observed defect distribution is specific to the engine under study.

The cross-language claim is formulated at the level of roles and contracts for the two transferred operations, Rename Method and Change Signature. The *rope* proof of concept demonstrates that separation, decoration, and composition can be introduced into an independently developed engine without replacing its program and change models. It does not establish comparable reuse across the Python refactoring catalog.

Chapters 5 and 7 discuss the threats to the validity of each study, along with the measures taken to mitigate them.

8.4 Future Work

We plan to complete the migration of the refactoring catalog under the new architecture. The migration also presents an opportunity to document the decisions that recur as each refactoring is migrated. These decisions can be compiled into a migration catalog for developers working on similar engines. One such decision breaks down a compound precondition into its applicability and behavior-preserving components. Another extracts a driver from an interactive flow.

We also plan to make Refazz a permanent quality gate. Running it on a fixed set of projects before each Pharo release would catch regressions in the engine, and expanding its strategies to include additional refactorings would catch more of them.

Beyond Python, a class-based dynamically typed language like Ruby is the next direct target, while multi-paradigm languages such as JavaScript would expose a broader range of constructs.

Exploring statically typed engines is a further direction. We expect the roles to remain unchanged. More behavior-preserving facts are decidable before execution, so fewer configurations would end with warnings. Warnings would still be necessary in cases where static facts are insufficient, such as with reflection or framework-driven dispatch. The compiler would introduce an outcome check to the execution modes of Refazz without replacing project tests, as a program that compiles can still behave differently.

Bibliography

[AC11] Akram Ajouli and Julien Cohen. Refactoring composite to visitor and inverse transformation in java. *ArXiv*, abs/1112.4271, 2011.

[ACD+22] Nicolas Anquetil, Miguel Campero, Stéphane Ducasse, Juan-Pablo Sandoval, and Pablo Tesone. Transformation-based refactorings: a first analysis. In *International Workshop of Smalltalk Technologies, IWST'22*, 2022.

[AMO24] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Behind the intent of extract method refactoring: A systematic literature review. *IEEE Transactions on Software Engineering*, 50(4):668–694, 2024.

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Lulu - Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[BG98] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):51–56, 1998.

[BGH07] Marat Boshernitsan, L. Susan Graham, and A. Marti Hearst. Aligning development tools with the way programmers think about code changes. In *Conference on Human Factors in Computing Systems (CHI '07)*, April 2007.

[BHH+11] István Bozó, Dániel Horpácsi, Zoltán Horváth, Róbert Kitlei, Judit Koszegi, M. Tejfel, and Melinda Tóth. Refactorerl - source code analysis and refactoring in erlang. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools, ISBN 978-9949-23-178-2*, pages 138–148, Tallin, Estonia, October 2011.

[BR98] John Brant and Don Roberts. "Good Enough" Analysis for Refactoring. In *Object-Oriented Technology Ecoop '98 Workshop Reader*, LNCS, pages 81–82. Springer-Verlag, 1998.

[BUA+23] Ana Carla Bibiano, Anderson Gonçalves Uchôa, Wesley Klewerton Guez Assunção, Daniel Oliveira, Thelma Elita Colanzi, Silvia Regina Vergilio, and Alessandro F. Garcia. Composite refactoring: Representations, characteristics and effects on software projects. *Information and Software Technology*, 156:107134, 2023.

[CA13] Julien Cohen and Akram Ajouli. Practical use of static composition of refactoring operations. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1700–1705, New York, NY, USA, 2013.

[CCY98] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic Testing: A New Approach for Generating Next Test Cases. *ArXiv*, 1998.

[CHLN06] Steve Counsell, Youssef Hassoun, George Lozou, and Rajaa Najjar. Common refactorings, a dependency graph and some code smells: An empirical study of java oss. In *International Symposium on Empirical Software Engineering*, volume 2006, pages 288–296, September 2006.

[CKL+18] Tsong Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T.H. Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys*, 51:4:1–4:27, January 2018.

[DB13] Stéphane Ducasse and Clément Bera. Blocks: a detailed analysis. In *Deep Into Pharo*, page 25. Square Bracket Associates, September 2013.

[DDGM07] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 185–194, New York, NY, USA, 2007. Association for Computing Machinery.

[DHT10] Dániel Drienyovszky, Dániel Horpácsi, and Simon Thompson. Quickchecking refactoring tools. In *Proceedings of the 9th ACM SIGPLAN Workshop on Erlang*, pages 75–80, 2010.

[DMP16] Stéphane Ducasse, Eliot Miranda, and Alain Plantec. Pragmas: Literal messages as powerful method annotations. In *International Workshop on Smalltalk Technologies IWST'16*, Prague, Czech Republic, August 2016.

[DPS23] Stéphane Ducasse, Guillermo Polito, and Juan Pablo Sandoval. *Testing in Pharo*. Book on Demand – Keepers of the lighthouse, 2023.

[dSS17] Gustavo Jansen de Souza Santos. *Assessing and Improving Code Transformations to Support Software Evolution*. PhD thesis, University Lille 1 - Sciences et Technologies - France, February 2017.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[FGL12] Stephen R. Foster, William G. Griswold, and Sorin Lerner. Witchdoctor: Ide support for real-time auto-completion of refactorings. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 222–232, Piscataway, NJ, USA, 2012. IEEE Press.

[FMM+11] Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. Refactoring towards the good parts of javascript. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 189–190, New York, NY, USA, 2011. Association for Computing Machinery.

[Fow18] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, Boston, second edition, November 2018.

[GBL+13] Milos Gligoric, Farnaz Behrang, Yilong Li, Jeffrey Overbey, Munawar Hafiz, and Darko Marinov. Systematic testing of refactoring engines on real software projects. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 629 – 653, Berlin, Heidelberg, 2013. Springer-Verlag.

[GCA22] Levi Gomes, Cassio Cordeiro, and Everton L. G. Alves. Evaluating the effectiveness of regression test suites for extract method validation. In *Proceedings of the 7th Brazilian Symposium on Systematic and Automated Software Testing*, SAST '22, pages 1–8, New York, NY, USA, 2022. Association for Computing Machinery.

[GDMH12] Xi Ge, Quinton L. DuBose, and Emerson Murphy-Hill. Reconciling manual and automatic refactoring. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 211–221, Piscataway, NJ, USA, 2012. IEEE Press.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GJ05] Alejandra Garrido and Ralph Johnson. Analyzing multiple configurations of a c program. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 379–388. IEEE, 2005.

[GN93] William G. Griswold and David Notkin. Automated assistance for program restructuring. *ACM Transaction of Software Engineering Methodologies*, 2(3):228–269, 1993.

[GRO25] Rohit Gheyi, Marcio Ribeiro, and Jonhnanthan Oliveira. Evaluating the effectiveness of small language models in detecting refactoring bugs, 2025.

[HKH17] Dániel Horpácsi, Judit Köszegi, and Zoltán Horváth. Trustworthy refactoring via decomposition and schemes: A complex case study. In *VPT@ETAPS*, 2017.

[HKN21] Daniel Horpácsi, Judit Köszegi, and Dávid J. Németh. Towards a generic framework for trustworthy program refactoring. *Acta Cybernetica*, 25(4):753–779, March 2021.

[HKT16] Dániel Horpácsi, Judit Kőszegi, and Simon Thompson. Towards trustworthy refactoring in erlang. *arXiv preprint arXiv:1607.02228*, 2016.

[HKV12] Mark Hills, Paul Klint, and Jurgen J. Vinju. Scripting a refactoring with rascal and eclipse. In *5th Workshop on Refactoring Tools*, pages 40–49, 2012.

[HLK+08] Zoltán Horváth, László Lövei, Tamás Kozsik, Róbert Kitlei, Anikó Nagyné Víg, Tamás Nagy, Melinda Tóth, and Roland Király. Building a refactoring tool for erlang. In *Workshop on Advanced Software Development Tools and Techniques, WASDETT*, volume 2008, 2008.

[HO15] Munawar Hafiz and Jeffrey Overbey. Refactoring myths. *IEEE Software*, 32(6):39–43, 2015.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications conference (OOPSLA'97)*, pages 318–326. ACM Press, November 1997.

[JLDM09] Vilas Jagannath, Yun Young Lee, Brett Daniel, and Darko Marinov. Reducing the costs of bounded-exhaustive testing. In *Fundamental Approaches to Software Engineering*, pages 171–185, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[KBDA16] Jongwook Kim, Don Batory, Danny Dig, and Maider Azanza. Improving refactoring speed by 10x. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1145 – 1156, 2016.

[KCH+08] Tamás Kozsik, Zoltán Csörnyei, Zoltán Horváth, Roland Király, Róbert Kitlei, László Lövei, Tamás Nagy, Melinda Tóth, and Anikó Víg. Use cases for refactoring in erlang. *Central European Functional Programming School: Second Summer School, CEFP 2007, Cluj-Napoca, Romania, June 23-30, 2007, Revised Selected Lectures 2*, pages 250–285, 2008.

[Kes18] Pascal Kesseli. *Semantic Refactorings.* PhD thesis, University of Oxford, 2018.

[KWK04] Günter Kniesel-Wünsche and Helge Koch. Static composition of refactorings. *Science of Computer Programming*, 52:9–51, August 2004.

[LCJ13] Yun Young Lee, Nicholas Chen, and Ralph E. Johnson. Drag-and-drop refactoring: intuitive and efficient program transformation. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 23–32. IEEE Press, 2013.

[LT07] H. Li and S. Thompson. Testing erlang refactorings with quickcheck. In *IFL*, pages 19 – 36. Springer-Verlag, 2007.

[LT11] Huiqing Li and Simon Thompson. A user-extensible refactoring tool for erlang programs. Technical report, University of Kent, October 2011.

[LT12a] Huiqing Li and Simon Thompson. A domain-specific language for scripting refactorings in erlang. In *FASE*, 2012.

[LT12b] Huiqing Li and Simon Thompson. Let's make refactoring tools user-extensible! In *Proceedings of the fifth workshop on refactoring tools*, pages 32–39, 2012.

[LTOT08] Huiqing Li, Simon Thompson, György Orosz, and Melinda Tóth. Refactoring with wrangler, updated: Data and process refactorings, and integration with eclipse. In *Workshop on ERLANG*, pages 61–72, New York, NY, USA, 2008. Association for Computing Machinery.

[MAP+08] Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. A case study on the impact of refactoring on quality and productivity in an agile team. In *Balancing Agility and Formalism in Software Engineering*, pages 252–266. Springer Berlin Heidelberg, 2008.

[MBBI18] Eliot Miranda, Clément Béra, Elisa Gonzalez Boix, and Dan Ingalls. Two decades of Smalltalk VM development: live VM development through simulation tools. In *Proceedings of International Workshop on Virtual Machines and Intermediate Languages (VMIL'18)*, pages 57–66. ACM, 2018.

[MGS+14] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Leopoldo Teixeira, and Paulo Borba. Making refactoring safer through impact analysis. *Science of Computer Programming*, 93:39 – 64, 2014.

[MGS+18] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Marcio Ribeiro, Paulo Borba, and Leopoldo Teixeira. Detecting overly strong preconditions in refactoring engines. *IEEE Transactions on Software Engineering*, 44(5):429 – 452, 2018.

[MHPB09] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In *International Conference on Software Engineering (ICSE)*, pages 287–297, 2009.

[MHPB11] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2011.

[MMG+14] Melina Mongiovi, Gustavo Mendes, Rohit Gheyi, Gustavo Soares, and Marcio Ribeiro. Scaling testing of refactoring engines. In *International Conference on Software Maintenance and Evolution*, pages 371–380, 2014.

[Mon16] Melina Mongiovi. Scaling testing of refactoring engines. In *International Conference on Software Engineering Companion*, ICSE '16, pages 674 – 676, New York, NY, USA, 2016. Association for Computing Machinery.

[MSL12] Philip Mayer, Andreas Schroeder, and Welf Löwe. Cross-language code analysis and refactoring. In *In Proceedings of the International Workshop on Source Code Analysis and Manipulation*, 2012.

[MT04] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transaction on Software Engineering*, 30(2):126–139, 2004.

[MY05] Katsuhisa Maruyama and Shinichiro Yamamoto. Design and implementation of an extensible and modifiable refactoring tool. In *Proceedings of the 13th*

International Workshop on Program Comprehension, IWPC '05, pages 195–204, USA, 2005. IEEE Computer Society.

[NCV+13] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. A comparative study of manual and automated refactorings. In *27th European Conference on Object-Oriented Programming*, pages 552–576, 2013.

[OC00] Mel Ò Cinnéide. Composite refactorings for java programs. In *Proceedings of the Workshop on Formal Techniques for Java Programs, European Conference on Object-Oriented Programming*, April 2000.

[OCN99] M. Ò Cinnéide and P. Nixon. A methodology for the automated introduction of design patterns. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, page 463, USA, 1999. IEEE Computer Society.

[OGRG25] Jonhnanthan Oliveira, Rohit Gheyi, Márcio Ribeiro, and Alessandro Garcia. Bugs in the shadows: Static detection of faulty python refactorings. In *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software (SBES 2025)*, pages 182–192. Sociedade Brasileira de Computação, September 2025.

[OJ93] William F. Opdyke and Ralph E. Johnson. Creating abstract superclasses by refactoring. In *Proceedings of the 1993 ACM Conference on Computer Science*, pages 66–73. ACM Press, 1993.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.

[Ove11] Jeffrey Overbey. *A Toolkit for Constructing Refactoring Engines*. Phd dissertation, University of Illinois at Urbana-Champaign, 2011.

[Ove13] Jeffrey L. Overbey. Immutable source-mapped abstract syntax tree: a design pattern for refactoring engine apis. In *Proceedings of the 20th Conference on Pattern Languages of Programs*, PLoP '13, USA, 2013. The Hillside Group.

[PMP+16] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Softw. Pract. Exper.*, 46(9):1155–1179, September 2016.

[RBD15] Markiyan Rizun, Jean-Christophe Bach, and Stéphane Ducasse. Code transformation by direct transformation of asts. In *International Workshop on Smalltalk Technologies (IWST)*, 2015.

[RBJ97] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[RBJO96] Don Roberts, John Brant, Ralph E. Johnson, and Bill Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96*, April 1996.

[Rob99] Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, 1999.

[Ros19] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill Education, New York, eighth edition, 2019.

[SADT24] Balsa Sarenac, Nicolas Anquetil, Stéphane Ducasse, and Pablo Tesone. A new architecture reconciling refactorings and transformations. *Journal of Computer Languages*, page 101273, 2024.

[SAM18] Indy P. S. C. Silva, Everton L. G. Alves, and Patrícia D. L. Machado. Can automated test case generation cope with extract method validation? In *Proceedings of the XXXII Brazilian Symposium on Software Engineering*, SBES '18, pages 152–161, New York, NY, USA, 2018. Association for Computing Machinery.

[Šar26] Balša Šarenac. Replication package: Improving refactorings in dynamically typed object-oriented languages. Zenodo, 2026. Version 1.0.0, https://doi.org/10.5281/zenodo.22857548.

[Sch10] Max Schäfer. *Specification, Implementation and Verification of Refactorings*. PhD thesis, Oxford University Computing Laboratory, 2010.

[SdM10] Max Schäfer and Oege de Moor. Specifying and implementing refactorings. In *Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'10)*, 2010.

[SDPR24] Balsa Sarenac, Stéphane Ducasse, Guillermo Polito, and Gordana Rakic. Modular and extensible extract method. In *International Workshop on Smalltalk Technologies*, 2024.

[SFSRC16] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortes. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824, 2016.

[SGM13] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, 39(2):147–162, February 2013.

[SGSM10] Gustavo Soares, Rohit Gheyi, D. Serey, and Tiago Massoni. Making program refactoring safer. *IEEE Software*, 27(4):52–57, 2010.

[SHT24] Bendegúz Seres, Dániel Horpácsi, and Simon Thompson. Is this really a refactoring? automated equivalence checking for erlang projects. In *Proceedings of the 23rd ACM SIGPLAN International Workshop on Erlang*, Erlang 2024, pages 55–66, New York, NY, USA, 2024. Association for Computing Machinery.

[SK09] Emmad Saadeh and Derrick G. Kourie. Composite refactoring using fine-grained transformations. In *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, SAICSIT '09, pages 22–29, New York, NY, USA, 2009. Association for Computing Machinery.

[SKL06] Dennis Strein, Hans Kratz, and Welf Lowe. Cross-language program analysis and refactoring. In *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 207–216, 2006.

[SMG11] Gustavo Soares, Melina Mongiovi, and Rohit Gheyi. Identifying overly strong conditions in refactoring implementations. In *IEEE International Conference on Software Maintenance (ICSM)*, pages 173–182, 2011.

[Spi10] Diomidis Spinellis. Cscout: A refactoring browser for c. *Science of Computer Programming*, 75(4):216–231, 2010.

[Ste18] Friedrich Steimann. Constraint-based refactoring. *ACM Trans. Program. Lang. Syst.*, 40(1), January 2018.

[STST12] Max Schäfer, Andreas Thies, Friedrich Steimann, and Frank Tip. A comprehensive approach to naming and accessibility in refactoring java programs. *IEEE Transactions on Software Engineering*, 38(6):1233–1257, 2012.

[SVEdM09] Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege de Moor. Stepping stones over the refactoring rubicon – lightweight language extensions to easily realise refactorings. In Sophia Drossopoulou, editor, *European Conference on Object-Oriented Programming (ECOOP)*, pages 369–393. Springer-Verlag, 2009.

[SvP12] F. Steimann and J. von Pilgrim. Constraint-based refactoring with foresight. In *ECOOP*, 2012.

[TCGS23] Sewen Thy, Andreea Costea, Kiran Gopinathan, and Ilya Sergey. Adventure of a lifetime: Extract method refactoring for rust. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023.

[TDTP23] Iona Thomas, Stéphane Ducasse, Pablo Tesone, and Guillermo Polito. Pharo: a reflective language - A first systematic analysis of reflective APIs. In *IWST 23 - International Workshop on Smalltalk Technologies*, Lyon, France, August 2023.

[Tic01] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern, December 2001.

[TPF+20] Pablo Tesone, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, and Stéphane Ducasse. Preserving instance state during refactorings in live environments. *Future Generation Computer Systems*, 110:1–17, 2020.

[TS10] Andreas Thies and Friedrich Steimann. Systematic testing of refactoring tools. In *AST'10 Proceedings*. ACM, 2010. poster.

[VCM+13] M. Vakilian, N. Chen, R. Z. Moghaddam, S. Negara, and R. E. Johnson. A compositional paradigm of automating refactorings. In *European Conference on Object-Oriented Programming*, pages 527–551, 2013.

[VCN+12] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. Use, disuse, and misuse of automated refactorings. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 233–243, Piscataway, NJ, USA, 2012. IEEE Press.

[VEdM06] M. Verbaere, R. Ettinger, and O. de Moor. Jungl: a scripting language for refactoring. In *Proceedings of International Conference on Software Engineering*, 2006.

[Vis01] Eelco Visser. A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, 57:109–143, 2001.

[VRS22] Philippe Voinov, Manuel Rigger, and Zhendong Su. Forest: Structural code editing with multiple cursors. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2022, pages 137–152, New York, NY, USA, 2022. Association for Computing Machinery.

[WXT25] Haibo Wang, Zhuolin Xu, and Shin Hwei Tan. Testing refactoring engine via historical bug report driven llm. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (FORGE)*, pages 113–124. IEEE, 2025.

[WXZ+26] Haibo Wang, Zhuolin Xu, Huaien Zhang, Nikolaos Tsantalis, and Shin Hwei Tan. Towards understanding refactoring engine bugs. *ACM Transactions on Software Engineering and Methodology*, 35(5):1–55, 2026.

[WY05] Daniel G. Waddington and Bin Yao. High-fidelity c/c++ code transformation. *Electronic Notes in Theoretical Computer Science*, 141(4):35–56, 2005. Proceedings of the Fifth Workshop on Language Descriptions, Tools, and Applications (LDTA 2005).

Appendix A

Program Model Interface Map

This appendix records the interface between refactoring operations and the existing Pharo program model. It is not general API documentation: its purpose is to show the kinds of queries and change-producing messages that preconditions and transformations use.

Scope. The map reflects the Pharo 14 evaluation snapshot at build commit c08e24c4937114b024e218b76880f435c1387de1. It groups representative selectors by receiver class and responsibility.

Chapter 4 uses this interface, while Chapter 5 reports how many direct calls were replaced by operation-level reuse.

Table A.1: Categorized summary of the Pharo program-model interface used by refactoring operations.

Responsibility	Primary receivers	Representative selectors	Architectural use
Class and hierarchy queries	RBAbstractClass, RBClass, RBMetaclass, RBTrait	allSubclasses, allSuperclasses, definesVariable:, hierarchyDefinesMethod:, traitComposition	Preconditions determine whether an operation is applicable and whether it carries a behavior risk.
Binding and reference queries	RBAbstractClass, RBNamespace, RBMethod	bindingOf:, allReferencesTo:, allImplementorsOf:, refersToVariable:	Rename, move, extract, and warning preconditions locate affected program entities.
Source and AST access	RBAbstractClass, RBMethod	parseTreeFor:, sourceCodeFor:, ast, source	Transformations inspect and rewrite source while retaining source-level information.
Method and variable changes	RBAbstractClass, RBClass, RBNamespace	addMethod:, removeMethod:, addInstanceVariable:, removeClassVariable:from:	Program-level transformations construct checked entity changes.
Class and package changes	RBNamespace	defineClass:, renameClass:to:around:, removePackageNamed:, reparentClasses:to:	Class and package transformations construct changes through the same program-model interface.
Change accumulation and lookup	RBNamespace, RBPackage	changes, classNamed:, packageNamed:, hasRemoved:	Composite steps resolve intermediate entities and accumulate deferred changes.

Appendix B

Extract Method in the Legacy Engine

This appendix documents the legacy Extract Method refactoring referenced in Section 5.2.1. It records the monolithic algorithm's execution flow and analyzes two language-specific concerns that the composite implementation in Section 5.2.2 must reproduce: return propagation (including multiple assignments) and non-local returns. The material is a technical reference for engine developers and is kept separate from the evaluation argument in Chapter 5.

B.1 Legacy Extraction Algorithm

B.1.1 Purpose and invariants

Extract Method moves a selected sequence from a source method to a new method and replaces the selection with an invocation. It can also reuse an equivalent method already present in the hierarchy, a common step when introducing a hook in the Template Method pattern [GHJV95]. The implementation must establish three properties:

- the selection denotes a supported sequence;
- the source and extracted methods can be constructed; and
- control flow and values used after the selection are preserved.

B.1.2 Preparation and checks

The implementation examined here is the monolithic Pharo 12 Extract Method, inherited from the original Refactoring Browser [RBJO96, RBJ97]. Pharo 12 is used because this implementation was the starting point for the composite rewrite. The broader migration comparison in Chapter 5 uses Pharo 11. Roberts' thesis does not document this particular operation [Rob99], so the relevant algorithm is summarized here.

The preparation stage combines validation, analysis, interaction, and construction:

1. Resolve the source method and parse both the source and the selected sequence. The selected AST is embedded in a temporary RBMethodNode.
2. Locate the selection in the source tree and replace it with a placeholder that will later become the extracted-method invocation.
3. Determine whether a trailing assignment remains in the source method. If so, the placeholder becomes the right-hand side of that assignment and the extracted method must return the assigned value.
4. Reject unsupported positions, including a selection on the left-hand side of an assignment or at the start of a cascade.
5. Analyze return propagation, non-local returns, and temporary-variable flow. At most one variable assigned inside the selection may still be required by the remaining source method. That variable becomes the extracted method's result.
6. Remove temporaries no longer needed by the source method and add the required arguments and temporaries to the extracted method.

In the legacy implementation, this stage is called precondition checking, but several of its steps are preparation rather than checks. They determine which variable the extracted method returns and which arguments and temporaries it needs. The composite design (Section 5.2) performs this preparation in a separate stage, before the checks and the transformation steps.

B.1.3 Change construction and interaction

After preparation, the implementation asks whether to search the hierarchy for an equivalent method. If a match is selected, its selector and parameter mapping are reused. Otherwise, the user supplies a new selector. The implementation then replaces the placeholder with the configured invocation and assigns the selector to the extracted method.

These choices are semantically useful, but they are requested inside the core algorithm. A script must therefore reproduce an interactive protocol even when all parameters are already known.

B.1.4 Consequences for the composite design

Existing capabilities. The legacy operation handles selection parsing, cascades, assignment flow, temporary variables, return propagation, non-local returns, and reuse of an equivalent hierarchy method. The composite implementation must preserve all of these cases instead of replacing them with simpler operations.

Coupling and reuse limitations. Preparation data, precondition decisions, user interaction, and source changes are produced by the same object. The operation also performs model and AST changes directly even when equivalent checked transformations already

exist [ACD+22]. Chapter 5 evaluates how the composite implementation separates and reuses these responsibilities.

The following sections describe the two control-flow cases that most directly constrain that decomposition: return propagation and non-local returns.

B.2 Return Propagation and Assigned Values

Extraction raises two questions: (1) whether the source method must return the extracted-method invocation, and (2) which value the extracted method must return.

Return from the source method. The source method must return the extracted-method invocation when the selection includes the source method's explicit return as its final statement. Listing B.1 illustrates this case. Selecting the final three statements moves ^ self isOk into the extracted method, so the replacement invocation also becomes the source method's return. If the explicit return is outside the selection, the invocation remains an ordinary statement.

```
ExampleClass >> exampleMethod
    | x |
    x := self setUpEnvironment.
    self perform: x.
    self postProcess.
    ^ self isOk

"After refactoring:"
ExampleClass >> exampleMethod
    | x |
    x := self setUpEnvironment.
    ^ self extractedMethod: x.

ExampleClass >> extractedMethod: x
    self perform: x.
    self postProcess.
    ^ self isOk
```

Listing B.1: A selection containing the source method's return requires the source method to return the extracted invocation.

Return from the extracted method. The extracted method must return a value when the replacement expression is used by its surrounding source context. The engine derives this requirement from the selected AST and its parent. For example, extracting the expression that supplies a source-method return requires the extracted method to return the same value, as shown in Listings B.2 and B.3.

```
ExampleClass >> foo: aString
    ^ self validate: aString
```

Listing B.2: A method returning the result of a message send.

```
ExampleClass >> foo: aString
    ^ self extractedMethod: aString
ExampleClass >> extractedMethod: aString
    ^ self validate: aString
```

Listing B.3: Extracting the return expression from Listing B.2.

Multiple assignments. Pharo methods return one value. When an assigned variable is read after the selection, the extracted method must return that variable so that the source method can restore the assignment. Listings B.4 and B.5 show the single-variable case.

```
ExampleClass >> foo
    | a |
    a := 3 + (2 sqrt - 4) - (4 + 2 sqrt).
    ^ self validate: a
```

Listing B.4: Example method with a single assignment.

```
ExampleClass >> foo
    | a |
    a := self extractedMethod.
    ^ self validate: a
ExampleClass >> extractedMethod
    | a |
    a := 3 + (2 sqrt - 4) - (4 + 2 sqrt).
    ^ a
```

Listing B.5: Extracting the assignment from Listing B.4.

Several assignments may be extracted when at most one assigned variable is read after the selection. That variable becomes the single returned value. Listing B.6 provides the source example.

```
ExampleClass >> foo
    | a b c d |
    a := 3.
    b := self bar: a.
    c := self baz: b.
    d := self doSomething.
    ^ self validate: c and: b
```

Listing B.6: Example method with multiple assignments.

If the assignments to a and b are extracted, only b is read by the remaining source method. The expected result is shown in Listing B.7.

```
ExampleClass >> foo
    | b c d |
    b := self extractedMethod.
    c := self baz: b.
    d := self doSomething.
    ^ self validate: c and: b
```

```
7
8  ExampleClass >> extractedMethod
9      | a b |
10     a := 3.
11     b := self bar: a.
12     ^ b
```

Listing B.7: Extracting the assignments to a and b from Listing B.6.

By contrast, extracting the assignments to b and c would require two values because both variables are read after the selection. That case is rejected by the implementation examined here.

Scoped implementation rule. For selections accepted by the preceding control-flow and variable-flow checks, the composite implementation always gives the extracted method an explicit return. The returned expression is either the single assigned variable required by the remaining source method or the selected sequence's final value. The rule may produce an unused result when the source context ignores the invocation's value. Listing B.8 shows the rule for the assignments to c and d from Listing B.6. The extracted method returns c, the only assigned variable read after the selection.

```
1  ExampleClass >> foo
2      | a b c |
3      a := 3.
4      b := self bar: a.
5      c := self extractedMethod: b.
6      ^ self validate: c and: b
7
8  ExampleClass >> extractedMethod: b
9      | c d |
10     c := self baz: b.
11     d := self doSomething.
12     ^ c
```

Listing B.8: Returning the assigned value required after a multiple-assignment selection.

In the composite design, selection and variable-flow preconditions choose the returned value, an add-return step constructs the extracted method's result, and the source-replacement step decides whether the invocation itself must be wrapped in a return. In this way, the legacy return logic is split into reusable preconditions and transformation steps.

B.3 Non-Local Returns

Pharo blocks are lexical closures. An explicit return inside a block is non-local: it exits the activation of the method in which the block was created, not merely the block invocation [DB13]. In Listing B.9, the block is created by run. If x is true, ^ self evaluate terminates run and supplies its result to the caller. exampleMethod receives that result in result and then continues with self conclude.

```
ExampleClass >> run
    x ifTrue: [ ^ self evaluate ].
    ^ self withoutEvaluation.

ExampleClass >> exampleMethod
    result := self run.
    self conclude.
```

Listing B.9: A block whose non-local return exits the home-method activation.

Moving a block containing a non-local return into a new method changes the block's home method. Extraction is therefore safe only when the replacement invocation reproduces the exits of the selected region in the source method. For the restricted rule used by the composite implementation, a selection containing a non-local return must extend through the source method's final explicit return. The replacement invocation is then itself returned from the source method.

In Listing B.10, selecting the sequence from c := self computeValue through the final return includes both exits. Selecting only the conditional return, or stopping before the final return, would leave an exit in the source method and change the control-flow relation.

```
ExampleClass >> foo
    | c |
    c := self computeValue.
    c ifOdd: [ ^ true ].
    ^ self validate: c
```

Listing B.10: A method containing a block with a non-local return.

Scoped acceptance rule. For the supported cases, a selection containing a non-local return is accepted only when its last statement is the source method's explicit return. Otherwise, the selection must contain no non-local return. This rule is sufficient for the control-flow cases handled by the composite implementation. It is not a general proof for every Pharo block, in particular not for an ensure/unwind interaction (a non-local return that unwinds through an ensure: block).

In the composite design, the non-local-return precondition enforces this rule. The source-replacement step then wraps the replacement invocation in a return, and the extracted method keeps the selected final return.

Appendix C

Confirmed Refactoring Engine Defects

Table C.1 lists the 19 unique refactoring engine defects used throughout this thesis. Duplicate tracker entries and issue #17259, which records a compiler regression rather than a refactoring engine defect, are excluded. "fixed" means that the correction has been integrated into the Pharo development branch.

Table C.2 maps each issue to its execution mode, detection signal, supporting evidence, and the two classifications used in Chapter 7. The strategy family names the parameter strategy that produced the failing configuration. *Any* means that every parameter strategy triggers the defect. The defect class records the primary root cause used in the totals, while the taxonomy identifies the affected technical concern. A defect may concern precondition precision in the taxonomy even when its first failing execution occurred under MR1.

The mapping reconciles to 12 transformation defects and 7 precondition defects (3 missing, 2 overly conservative, and 2 misplaced): 13 defects found under MR1, three MR2 missing preconditions, one MR2 overly conservative check, one MR2 misplaced behavior-preserving check, and one MR3 overly conservative check. Each issue contributes to one primary defect class. Secondary characteristics are not counted separately. For Rename Class issue #18240, the fixing PR #19125 establishes the primary root-cause classification: it removes the late hard error and reifies the collision as a behavior-preserving precondition that the driver presents as an overridable warning. The catalog therefore counts the defect primarily as overly conservative. It is also misplaced because the hard restriction was implemented in the execution path rather than in the behavior-preserving preconditions. The taxonomy totals are one name/scope/binding, five control-flow/return, nine structural/syntactic, and four precondition-precision/placement defects.

Table C.1: Confirmed defects in the studied Pharo refactoring engine.

Area	Description	Status	Issue
Add Argument	Missing check for the number of keywords, arguments, and permutations	fixed	#18259
Composite Extract Method	Allows extraction of the left side of an assignment	confirmed	#16882
Composite Extract Method	Cannot extract expression with parentheses	confirmed	#16883
Composite Extract Method	Should not extract a temporary declaration	confirmed	#16885
Composite Extract Method	Invalid parse tree when extracting a temporary usage in an expression	confirmed	#16888
Composite Extract Method	Cannot extract method from the class side	fixed	#16890
Composite Extract Method	Missing precondition for invalid interval cases	fixed	#18152
Composite Extract Method	Issues when extracting blocks that are arguments	confirmed	#18401
Composite Extract Method	Infinite recursion when extracting the receiver block of whileTrue:	confirmed	#18402
Composite Extract Method	Fails when replacing code with non-local return	confirmed	#18403
Composite Extract Method	Fails to extract a statement	confirmed	#18404
Extract Method	Override check hidden in optional name handling	confirmed	#17235
Extract Method	Extracting from a literal array element breaks code	confirmed	#16173
Extract Method	Extracting super breaks code	confirmed	#16174
Extract Method	Extract method with non-local returns breaks behavior	confirmed	#16175
Extract Method	Overly conservative precondition	confirmed	#17254
Extract Method	Missing handling when statement list is empty	fixed	#18151
Inline Method	Behavior-preserving preconditions checked during execution	fixed	#18154
Rename Class	Overly conservative shared-variable restriction placed in the execution path	fixed	#18240

Table C.2: Execution mode, detection signal, supporting evidence, and classification of the confirmed defects.

Issue	Refactoring	Mode	Strategy family	Detection signal	Supporting evidence	Defect class	Taxonomy
#18259	Add Argument	MR2	Parameter permutation	Terminated execution	Exception	Missing applicability precondition	Structural validity
#16882	Composite Extract Method	MR1	Source interval	Project-test failure	Incorrect generated code	Transformation defect	Structural validity
#16883	Composite Extract Method	MR1	Source interval	Terminated execution	Parse failure	Transformation defect	Structural validity
#16885	Composite Extract Method	MR1	Source interval	Terminated execution	Parse failure	Transformation defect	Structural validity
#16888	Composite Extract Method	MR1	Source interval	Terminated execution	Unformattable parse tree	Transformation defect	Structural validity
#16890	Composite Extract Method	MR1	Any	Terminated execution	Binding and source inspection	Transformation defect	Name/scope/binding
#18152	Composite Extract Method	MR2	Source interval	Terminated execution	AST and source inspection	Missing applicability precondition	Structural validity
#18401	Composite Extract Method	MR1	Source interval	Project-test failure	Incorrect generated code	Transformation defect	Control flow/returns
#18402	Composite Extract Method	MR1	Source interval	Project-test failure	Infinite recursion	Transformation defect	Control flow/returns
#18403	Composite Extract Method	MR1	Source interval	Terminated execution	Non-local-return replacement failure	Transformation defect	Control flow/returns
#18404	Composite Extract Method	MR1	Source interval	Terminated execution	Statement-replacement failure	Transformation defect	Control flow/returns
#17235	Extract Method	MR2	Name/inherited selector	Project-test failure	Late option check	Misplaced behavior-preserving precondition	Precondition precision
#16173	Extract Method	MR1	Source interval	Project-test failure	Changed literal-array contents	Transformation defect	Structural validity
#16174	Extract Method	MR1	Source interval	Project-test failure	Infinite recursion after extracting `super`	Transformation defect	Structural validity
#16175	Extract Method	MR1	Source interval	Project-test failure	Non-local-return inspection	Transformation defect	Control flow/returns
#17254	Extract Method	MR3	Source interval	Completed forced execution	Manual inspection	Overly conservative applicability precondition	Precondition precision
#18151	Extract Method	MR2	Source interval	Terminated execution	Empty statement-list failure	Missing applicability precondition	Structural validity
#18154	Inline Method	MR1	Any	Terminated execution	Late warning during execution	Misplaced behavior-preserving precondition	Precondition precision
#18240	Rename Class	MR2	Name/shared variable	Terminated execution	Debugger exception and fixing PR #19125	Overly conservative behavior-preserving precondition	Precondition precision

Appendix D

Project-Specific Refazz Outcomes

This appendix reports aggregate outcomes of refactoring attempts by project, refactoring, and execution mode for Soup, Graph algorithms, Artefact, AVL, and Microdown. Each table uses the four execution modes: MR1 (valid-input), MR2 (input exploration), MR3 (forced after applicability rejection), and MR4 (forced after behavior-preserving warning). The outcome columns report *tests failed*, *tests passed*, and *did not complete* (terminated), corresponding to MuTalk's raw *killed*, *alive*, and *terminated* outcomes. The tables and the attempt totals in Chapter 7 count only attempts whose execution started. A configuration that is rejected or stops after a warning is not included. These counts describe attempts, so repeated attempts can expose the same underlying defect. Appendix C provides the 19-row audit of unique confirmed defects.

Table D.1: Aggregate Refazz attempt outcomes by execution mode for the Soup project.

Refactoring	Mode	Attempts	Tests failed	Tests passed	Did not complete
Add Argument	MR1: valid-input	488	36	450	2
	MR2: input exploration	166	3	161	2
	MR3: forced after applicability rejection	1,021	36	759	226
	MR4: forced after behavior-preserving warning	62	56	4	2
Composite Extract Method	MR1: valid-input	1,918	16	1,662	240
	MR2: input exploration	1,431	15	1,214	202
	MR3: forced after applicability rejection	63	57	6	0
	MR4: forced after behavior-preserving warning	134	68	55	11
Extract Method	MR1: valid-input	1,589	6	1,583	0
	MR2: input exploration	1,694	3	1,689	2
	MR3: forced after applicability rejection	8,458	43	41	8,374
	MR4: forced after behavior-preserving warning	0	0	0	0
Inline Method	MR1: valid-input	250	0	250	0
	MR2: input exploration	458	0	458	0
	MR3: forced after applicability rejection	8,537	2	14	8,521
	MR4: forced after behavior-preserving warning	17	1	16	0
Rename Variable	MR1: valid-input	2,024	0	2,024	0
	MR2: input exploration	1,114	0	1,114	0
	MR3: forced after applicability rejection	3,617	254	1,203	2,160
	MR4: forced after behavior-preserving warning	0	0	0	0
Rename Class	MR1: valid-input	11	0	11	0
	MR2: input exploration	11	0	11	0
	MR3: forced after applicability rejection	5,156	0	0	5,156
	MR4: forced after behavior-preserving warning	0	0	0	0
Rename Method	MR1: valid-input	488	36	450	2
	MR2: input exploration	200	3	195	2
	MR3: forced after applicability rejection	494	0	0	494
	MR4: forced after behavior-preserving warning	319	54	259	6

Table D.2: Aggregate Refazz attempt outcomes by execution mode for the Graph algorithms project.

Refactoring	Mode	Attempts	Tests failed	Tests passed	Did not complete
Add Argument	MR1: valid-input	364	16	348	0
	MR2: input exploration	24	3	21	0
	MR3: forced after applicability rejection	754	16	582	156
	MR4: forced after behavior-preserving warning	152	64	88	0
Composite Extract Method	MR1: valid-input	2,424	43	1,993	388
	MR2: input exploration	2,375	35	1,942	398
	MR3: forced after applicability rejection	99	75	20	4
	MR4: forced after behavior-preserving warning	37	16	19	2
Extract Method	MR1: valid-input	2,031	31	2,000	0
	MR2: input exploration	2,011	27	1,984	0
	MR3: forced after applicability rejection	15,084	24	37	15,023
	MR4: forced after behavior-preserving warning	0	0	0	0
Inline Method	MR1: valid-input	147	0	147	0
	MR2: input exploration	259	0	259	0
	MR3: forced after applicability rejection	16,789	6	29	16,754
	MR4: forced after behavior-preserving warning	7	3	4	0
Rename Variable	MR1: valid-input	2,010	0	2,010	0
	MR2: input exploration	1,112	0	1,112	0
	MR3: forced after applicability rejection	5,593	546	1,916	3,131
	MR4: forced after behavior-preserving warning	0	0	0	0
Rename Class	MR1: valid-input	10	0	7	3
	MR2: input exploration	10	0	0	10
	MR3: forced after applicability rejection	4,469	0	0	4,469
	MR4: forced after behavior-preserving warning	0	0	0	0
Rename Method	MR1: valid-input	364	16	348	0
	MR2: input exploration	51	3	48	0
	MR3: forced after applicability rejection	364	0	0	364
	MR4: forced after behavior-preserving warning	337	104	233	0

Table D.3: Aggregate Refazz attempt outcomes by execution mode for the Artefact project.

Refactoring	Mode	Attempts	Tests failed	Tests passed	Did not complete
Add Argument	MR1: valid-input	1,288	71	1,205	12
	MR2: input exploration	304	16	252	36
	MR3: forced after applicability rejection	2,639	65	1,985	589
	MR4: forced after behavior-preserving warning	216	63	153	0
Composite Extract Method	MR1: valid-input	2,675	5	2,492	178
	MR2: input exploration	2,422	4	2,233	185
	MR3: forced after applicability rejection	211	24	184	3
	MR4: forced after behavior-preserving warning	9	0	8	1
Extract Method	MR1: valid-input	2,284	4	2,280	0
	MR2: input exploration	1,934	4	1,930	0
	MR3: forced after applicability rejection	14,965	0	2	14,963
	MR4: forced after behavior-preserving warning	0	0	0	0
Inline Method	MR1: valid-input	532	0	532	0
	MR2: input exploration	905	0	905	0
	MR3: forced after applicability rejection	14,802	0	12	14,790
	MR4: forced after behavior-preserving warning	106	26	80	0
Rename Variable	MR1: valid-input	4,006	0	4,006	0
	MR2: input exploration	2,503	0	2,503	0
	MR3: forced after applicability rejection	7,564	67	2,817	4,680
	MR4: forced after behavior-preserving warning	0	0	0	0
Rename Class	MR1: valid-input	305	0	305	0
	MR2: input exploration	305	0	305	0
	MR3: forced after applicability rejection	10,096	0	0	10,096
	MR4: forced after behavior-preserving warning	0	0	0	0
Rename Method	MR1: valid-input	1,288	56	1,176	56
	MR2: input exploration	363	19	329	15
	MR3: forced after applicability rejection	1,290	0	0	1,290
	MR4: forced after behavior-preserving warning	922	70	810	42

Table D.4: Aggregate Refazz attempt outcomes by execution mode for the AVL project.

Refactoring	Mode	Attempts	Tests failed	Tests passed	Did not complete
Add Argument	MR1: valid-input	144	2	142	0
	MR2: input exploration	27	1	26	0
	MR3: forced after applicability rejection	299	2	236	61
	MR4: forced after behavior-preserving warning	35	10	25	0
Composite Extract Method	MR1: valid-input	1,095	1	845	249
	MR2: input exploration	1,538	1	1,083	454
	MR3: forced after applicability rejection	10	0	10	0
	MR4: forced after behavior-preserving warning	17	0	16	1
Extract Method	MR1: valid-input	933	1	932	0
	MR2: input exploration	1,218	1	1,217	0
	MR3: forced after applicability rejection	4,937	2	24	4,911
	MR4: forced after behavior-preserving warning	0	0	0	0
Inline Method	MR1: valid-input	33	0	33	0
	MR2: input exploration	78	0	78	0
	MR3: forced after applicability rejection	6,077	0	6	6,071
	MR4: forced after behavior-preserving warning	0	0	0	0
Rename Variable	MR1: valid-input	720	0	720	0
	MR2: input exploration	633	0	633	0
	MR3: forced after applicability rejection	4,508	14	2,054	2,440
	MR4: forced after behavior-preserving warning	0	0	0	0
Rename Class	MR1: valid-input	7	0	7	0
	MR2: input exploration	7	0	7	0
	MR3: forced after applicability rejection	1,540	0	0	1,540
	MR4: forced after behavior-preserving warning	0	0	0	0
Rename Method	MR1: valid-input	144	2	142	0
	MR2: input exploration	38	1	37	0
	MR3: forced after applicability rejection	144	0	0	144
	MR4: forced after behavior-preserving warning	109	0	109	0

Table D.5: Aggregate Refazz attempt outcomes by execution mode for the Microdown project.

Refactoring	Mode	Attempts	Tests failed	Tests passed	Did not complete
Add Argument	MR1: valid-input	1,548	88	1,458	2
	MR2: input exploration	232	3	115	114
	MR3: forced after applicability rejection	3,089	75	2,296	718
	MR4: forced after behavior-preserving warning	293	163	127	3
Composite Extract Method	MR1: valid-input	1,705	36	1,485	184
	MR2: input exploration	1,623	23	1,401	199
	MR3: forced after applicability rejection	140	79	58	3
	MR4: forced after behavior-preserving warning	92	47	33	12
Extract Method	MR1: valid-input	1,433	7	1,425	1
	MR2: input exploration	1,480	8	1,471	1
	MR3: forced after applicability rejection	11,640	22	13	11,605
	MR4: forced after behavior-preserving warning	0	0	0	0
Inline Method	MR1: valid-input	752	0	752	0
	MR2: input exploration	1,367	0	1,367	0
	MR3: forced after applicability rejection	8,351	6	7	8,338
	MR4: forced after behavior-preserving warning	186	26	160	0
Rename Variable	MR1: valid-input	3,788	0	3,784	4
	MR2: input exploration	1,970	0	1,968	2
	MR3: forced after applicability rejection	6,256	349	1,634	4,273
	MR4: forced after behavior-preserving warning	0	0	0	0
Rename Class	MR1: valid-input	104	4	98	2
	MR2: input exploration	104	0	5	99
	MR3: forced after applicability rejection	10,924	0	0	10,924
	MR4: forced after behavior-preserving warning	0	0	0	0
Rename Method	MR1: valid-input	1,548	79	1,467	2
	MR2: input exploration	302	10	290	2
	MR3: forced after applicability rejection	1,550	0	0	1,550
	MR4: forced after behavior-preserving warning	400	127	270	3

Biography

Balša Šarenac was born in 1997 in Trebinje, where he finished Jovan Dučić Gymnasium in 2016. He then enrolled at the Faculty of Technical Sciences, University of Novi Sad, majoring in Computing and Control Engineering, and graduated in 2020. He continued with master's studies at the same faculty and in 2021 defended his master's thesis, "Identifying Cohesive Parts of a Class." He enrolled in doctoral studies the same year.

In 2020, he became a teaching assistant at the Faculty of Technical Sciences, where he combines teaching and research. In 2022, he began researching refactoring with the RMoD team (now Evref) at Inria and the University of Lille. He visited the team as a research guest several times between 2022 and 2025. That work led to the topic of this thesis: refactoring in dynamically typed object-oriented languages. He is a contributor to the Pharo open-source community and the maintainer of its refactoring engine, which this thesis describes. To date, he has co-authored five research papers, one of which was published in a journal indexed in the Science Citation Index Expanded.

He has also been a part-time contractor for Open Law Library since 2021, where he works on industrial software and maintains large code bases that change continuously. His research interests include refactoring, software evolution, and developer tooling.

Овај Образац чини саставни део докторске дисертације, односно докторског уметничког пројекта који се брани на Универзитету у Новом Саду. Попуњен Образац укоричити иза текста докторске дисертације, односно докторског уметничког пројекта.

План третмана података

Назив пројекта/истраживања
Унапређење рефакторисања у динамички типизираним објектно-оријентисаним језицима Improving Refactorings in Dynamically Typed Object-Oriented Languages
Назив институције/институција у оквиру којих се спроводи истраживање
а) Факултет техничких наука, Универзитет у Новом Саду б) Inria центар при Универзитету у Лилу в)
Назив програма у оквиру ког се реализује истраживање
Докторске академске студије, студијски програм Рачунарство и аутоматика
1. Опис података
1.1 Врста студије *Укратко описати тип студије у оквиру које се подаци прикупљају* **Емпиријско истраживање у софтверском инжењерству које комбинује аутоматизовани рачунарски експеримент, вишеструку студију случаја и анализу изворног кода.** 1.2 Врсте података а) **квантитативни** б) **квалитативни** 1.3. Начин прикупљања података а) анкете, упитници, тестови б) клиничке процене, медицински записи, електронски здравствени записи в) генотипови: навести врсту _____ г) административни подаци: навести врсту _____ д) узорци ткива: навести врсту_____

ђ) снимци, фотографије: навести врсту_____

е) **текст**, навести врсту _____

ж) мапа, навести врсту _____

з) **остало**: **Подаци су генерисани аутоматизованим извршавањем алата Refazz над пет Pharo пројеката, анализом изворног кода Pharo 11 и Pharo 14, анализом дуплираног кода и ручном валидацијом аномалија. Коришћени су и јавно доступни записи о проблемима и исправкама из Pharo репозиторијума. Скуп обухвата и изворни код и тестове доказа концепта архитектуре у Python алату rope. Истраживање не укључује испитанике.**

1.3 Формат података, употребљене скале, количина података

1.3.1 Употребљени софтвер и формат датотеке:

а) Excel фајл, датотека _____

b) SPSS фајл, датотека _____

c) PDF фајл, датотека _____

d) **Текст фајл**, датотека **TXT, TeX, изворни код (Smalltalk .st, Python .py, JavaScript .js, као и shell скрипте)**

е) JPG фајл, датотека _____

f) **Остало**, датотека **CSV, JSON, Markdown и ZIP архиве (изворни код Refazz, MuTalk и rope на фиксираним комитима, у оквиру репликационог пакета).**

1.3.2. Број записа (код квантитативних података)

а) број варијабли **11 у главним Refazz CSV датотекама; пратећи табеларни скупови имају 30, 14, 5 и 4 варијабле, зависно од скупа.**

б) број мерења (испитаника, процена, снимака и сл.) **255.699 покушаја рефакторисања представљених у 180 агрегатних редова у 35 CSV датотека; 19 потврђених дефеката; 68 записа о клијентима поновне употребе и 14 агрегатних записа; четири извештаја анализе дуплираног кода; 2.224 аутоматских тестова доказа концепта у алату rope, од којих 103 додата за архитектуру.**

1.3.3. Поновљена мерења

а) да

б) **не**

Уколико је одговор да, одговорити на следећа питања:

а) временски размак између поновљених мера је _____

б) варијабле које се више пута мере односе се на _____

в) нове верзије фајлова који садрже поновљена мерења су именоване као _____

Напомене: _____

Да ли формати и софтвер омогућавају дељење и дугорочну валидност података?

а) ***Да***

б) Не

Ако је одговор не, образложити _____

_____

2. Прикупљање података

2.1 Методологија за прикупљање/генерисање података

Аутоматизовани рачунарски експеримент, вишеструка студија случаја и анализа изворног кода. Све верзије софтвера су фиксиране: Pharo 13.1, Refazz на комиту 40f1b1f6, MuTalk на комиту 2ba13136, пет пројеката на фиксираним комитима, Pharo 11 (комит 963dc3a7de) и Pharo 14 (комит c08e24c493) за мере поновне употребе и дуплирања, jscpd 4.2.5, и rope на upstream комиту f005ac78ca доказом концепта на комиту 77b23b4d (грана architecture-poc, https://github.com/balsa-sarenac/rope).

2.1.1. У оквиру ког истраживачког нацрта су подаци прикупљени?

а) **експеримент**, **аутоматизовани рачунарски експеримент над фиксираним верзијама софтвера**

б) корелационо истраживање, навести тип _____

ц) анализа текста, навести тип _____

д) **остало**, **вишеструка студија случаја и анализа изворног кода**

2.1.2 Навести врсте мерних инструмената или стандарде података специфичних за одређену научну дисциплину (ако постоје).

Нису коришћени стандардизовани мерни инструменти специфични за дисциплину. Мере су оперативно дефинисане у документацији скупа, а генерисане су алатима Refazz и jscpd 4.2.5.

2.2 Квалитет података и стандарди

2.2.1. Третман недостајућих података

а) Да ли матрица садржи недостајуће податке? Да **Не**

Ако је одговор да, одговорити на следећа питања:

а) Колики је број недостајућих података? _____

б) Да ли се кориснику матрице препоручује замена недостајућих података? Да Не

в) Ако је одговор да, навести сугестије за третман замене недостајућих података

_____

2.2.2. На који начин је контролисан квалитет података? Описати

Верзије софтвера и изворног кода су фиксиране, а све депоноване датотеке имају SHA-256 контролне суме (датотека SHA256SUMS.txt). Аутоматске скрипте проверавају шему и укупне вредности и регенеришу табеле из канонских CSV датотека. Свих 19 потврђених дефеката је ручно ревидирано, а сваки ред носи везу ка јавном запису о проблему и, где постоји, ка исправци.

2.2.3. На који начин је извршена контрола уноса података у матрицу?

Подаци су претежно генерисани аутоматски. Скрипте за проверу одбацују дуплиране идентификаторе проблема, неусклађене укупне вредности и застареле бројеве у тексту дисертације.

3. Третман података и пратећа документација

3.1. Третман и чување података

3.1.1. Подаци ће бити депоновани у Zenodo репозиторијум.

3.1.2. URL адреса https://doi.org/10.5281/zenodo.22857548

*3.1.3. DOI **10.5281/zenodo.22857548***

3.1.4. Да ли ће подаци бити у отвореном приступу?

*а) **Да***

б) Да, али после ембарга који ће трајати до _____

в) Не

Ако је одговор не, навести разлог _____

3.1.5. Подаци неће бити депоновани у репозиторијум, али ће бити чувани.

Образложење

Није примењиво — подаци ће бити депоновани у репозиторијуму Zenodo

3.2 Метаподаци и документација података

3.2.1. Који стандард за метаподатке ће бити примењен? **DataCite Metadata Schema, у верзији коју Zenodo примењује у тренутку објављивања**
Пратећа документација обухвата README са описом скупа, пореклом података, фиксираним верзијама софтвера и комитима, контролним сумама (SHA256SUMS.txt) и упутством за репродукцију. Депозит чини једна архива репликационог пакета, која садржи и изворни код доказа концепта у алату rore.

3.2.1. Навести метаподатке на основу којих су подаци депоновани у репозиторијум.

Пратећа документација обухвата README са описом скупа, пореклом података, верзијама софтвера, контролним сумама и упутством за репродукцију.

Ако је потребно, навести методе које се користе за преузимање података, аналитичке и процедуралне информације, њихово кодирање, детаљне описе варијабли, записа итд.

_____

_____

_____

_____

3.3 Стратегија и стандарди за чување података

3.3.1. До ког периода ће подаци бити чувани у репозиторијуму? **Током животног века репозиторијума Zenodo, у складу са његовом политиком задржавања.**

3.3.2. Да ли ће подаци бити депоновани под шифром? Да **Не**

3.3.3. Да ли ће шифра бити доступна одређеном кругу истраживача? Да Не (**Није примењиво — подаци нису заштићени шифром**)

3.3.4. Да ли се подаци морају уклонити из отвореног приступа после извесног времена?

Да **Не**

Образложити

Не планира се уклањање података из отвореног приступа. Подаци ће се чувати у складу са политиком задржавања репозиторијума Zenodo.

4. Безбедност података и заштита поверљивих информација

Овај одељак МОРА бити попуњен ако ваши подаци укључују личне податке који се односе на учеснике у истраживању. За друга истраживања треба такође размотрити заштиту и сигурност

података.

4.1 Формални стандарди за сигурност информација/података

Истраживачи који спроводе испитивања с људима морају да се придржавају Закона о заштити података о личности *(https://www.paragraf.rs/propisi/zakon_o_zastiti_podataka_o_licnosti.html)* и одговарајућег институционалног кодекса о академском интегритету.

Није примењиво; истраживање не укључује људе као испитанике

4.1.2. Да ли је истраживање одобрено од стране етичке комисије? Да **Не**

Ако је одговор Да, навести датум и назив етичке комисије која је одобрила истраживање

_____

4.1.2. Да ли подаци укључују личне податке учесника у истраживању? Да **Не**

Ако је одговор да, наведите на који начин сте осигурали поверљивост и сигурност информација везаних за испитанике:

а) Подаци нису у отвореном приступу

б) Подаци су анонимизирани

ц) Остало, навести шта

_____

_____

5. Доступност података

5.1. Подаци ће бити

а) ***јавно доступни***

б) доступни само уском кругу истраживача у одређеној научној области

ц) затворени

Ако су подаци доступни само уском кругу истраживача, навести под којим условима могу да их користе:

_____

_____

Ако су подаци доступни само уском кругу истраживача, навести на који начин могу приступити подацима:

_____

_____

5.4. Навести лиценцу под којом ће прикупљени подаци бити архивирани.

Creative Commons Attribution 4.0 International (CC BY 4.0).

6. Улоге и одговорност

6.1. Навести име и презиме и мејл адресу власника (аутора) података

Балша Шаренац, balsasarenac@uns.ac.rs

6.2. Навести име и презиме и мејл адресу особе која одржава матрицу с подацима

Балша Шаренац, balsasarenac@uns.ac.rs

6.3. Навести име и презиме и мејл адресу особе која омогућује приступ подацима другим истраживачима

Балша Шаренац, balsasarenac@uns.ac.rs

Након објављивања, технички приступ омогућава репозиторијум Zenodo.