Розділ 18
Рефлексія

Pharo – рефлексивна мова програмування. Рефлексія передбачає можливості для того, щоб і досліджувати структуру та поведінку програмної системи під час її виконання, і змінювати їх. Засоби дослідження часто називають *інтроспекцією* або самоаналізом, а можливості змін – *адаптивністю* або заступництвом[1]. У цьому розділі представлено кілька аспектів щодо засобів інтроспекції Pharo: як отримати доступ і змінити значення змінної екземпляра, як переміщуватися системою або як виконувати перехресні посилання. Також описано деякі аспекти поведінкової рефлексії, тобто, можливості змінити систему та розширити її.

18.1. Суть рефлексії

Рефлексія втілює ідею, що програма здатна *аналізувати* свої власні структуру та хід виконання. З рефлексією внутрішні механізми, які підтримують виконання програм (класи, методи, стек викликів), доступні для розробника так само, як і його звичайні програми. У Pharo це означає, що ці внутрішні механізми описані як звичайні об'єкти, і що розробник може надсилати їм повідомлення. Об'єкти, які підтримують виконання програм, часто називають *метаоб'єктами*, щоб наголосити на тому факті, що вони перебувають на іншому рівні, ніж звичайні об'єкти.

Точніше кажучи, *метаоб'єкти* програмної системи на етапі виконання *втілюються* в звичайні об'єкти, які можна запитувати та інспектувати. Метаоб'єкти Pharo – це класи, метакласи, словники методів, скомпільовані методи, а також стек часу виконання, процеси тощо. Таку форму рефлексії також називають *інтроспекцією*, її підтримує багато сучасних мов програмування (чому активно сприяв Smalltalk-80, попередник Pharo).

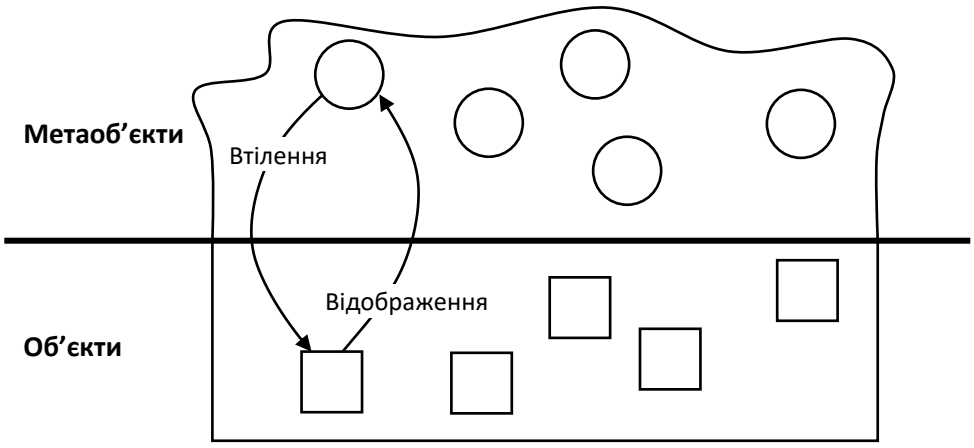

Рис. 18.1. Втілення та відображення

[1] В оригіналі – *introspection* and *intercession* (*прим. – Ярошко С.*).

З іншого боку, у Pharo можна модифікувати втілені метаоб'єкти та *відобразити* ці зміни назад у виконувану програму (рис. 18.1). Така здатність називається *заступництвом* і підтримується в здебільшого динамічними мовами програмування, і лише дуже обмежено – статичними мовами. Тому, зверніть увагу, якщо люди кажуть, що Java рефлексивна мова, то насправді це інтроспективна, а не рефлексивна мова.

Програма, яка керує іншими програмами (або навіть сама собою), є метапрограмою. Щоб бути рефлексивною, мова програмування має підтримувати і самоаналіз, і заступництво. Інтроспекція – це здатність *досліджувати* структури даних, які визначають мову: об'єкти, класи, методи та стек виконання. Адаптивність – це здатність *модифікувати* ці структури, іншими словами, змінювати семантику мови та поведінку програми зсередини самої програми. *Структурна рефлексія* стосується дослідження та модифікації структур середовища виконання, а *поведінкова рефлексія* стосується зміни інтерпретації цих структур.

У розділі йдеться головно про структурну рефлексію. Розглянуто багато практичних прикладів, які ілюструють, як Pharo підтримує самоаналіз і метапрограмування.

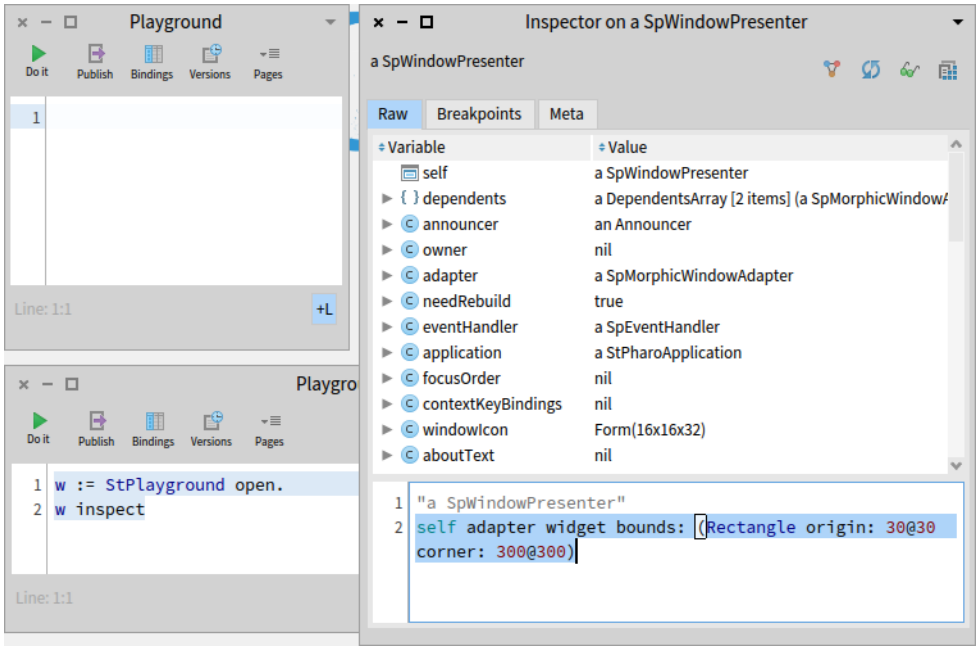

Рис. 18.2. Інспектування екземпляра *StPlayground*

18.2. Інтроспекція

За допомогою інспектора можна переглядати об'єкт, змінювати значення його змінних екземпляра та навіть надсилати йому повідомлення.

Виконайте в Робочому вікні наведений код.

```
w := StPlayground open.
w inspect
```

Він відкриє ще одне Робоче вікно й інспектор. Інспектор показує внутрішній стан нового вікна як список його змінних екземпляра: у лівому стовпці – імена змінних (*announcer*, *owner*, ...), а в правому – значення змінної навпроти відповідного імені.

Тепер виберіть вікно інспектора і клацніть на його панелі редагування коду (вона містить коментар угорі). Введіть у ній повідомлення «*self adapter widget bounds: (Rectangle*

origin: 30@30 corner: 300@300)», як зображено на рис. 18.2, і виконайте його командою «*Do it*», як у Робочому вікні.

Ви побачите, що створене Робоче вікно відразу переміститься та змінить розмір.

18.3. Доступ до змінних екземпляра

Як працює інспектор? У Pharo всі змінні екземпляра захищені. Теоретично неможливо доступитися до них з іншого об'єкта, якщо в класі не визначені методи доступу. На практиці інспектор отримує доступ до змінних екземпляра без таких методів, бо він використовує рефлексивні можливості Pharo. Класи визначають або іменовані змінні екземпляра, або індексовані. Для доступу до них інспектор використовує визначені в класі *Object* методи: *instVarAt: index* та *instVarNamed: aString* – для отримання значення змінної екземпляра з номером *index* або з ідентифікатором *aString*, відповідно. Так само, щоб призначити нові значення цим змінним екземпляра, він використовує *instVarAt: index put: value* та *instVarNamed: aString put: value*.

Продовжимо експерименти з Робочими вікнами, створеними в попередньому параграфі. Доповніть код у першому вікні наведеним нижче рядком і виконайте його.

```
w adapter widget
   instVarNamed:'bounds' put: (Rectangle origin: 150@30 corner: 450@300)
```

Клацніть на другому вікні, щоб актуалізувати оновлення та побачити результат. Щойно ви змінили стан об'єкта без допомоги методів доступу.

Змінні екземпляра

Метод *allInstVarNames* повертає імена всіх змінних екземпляра заданого класу.

```
StPlayground allInstVarNames
>>> #(#dependents #announcer #owner #adapter #needRebuild #eventHandler
    #application #focusOrder #contextKeyBindings #windowIcon #aboutText
    #askOkToClose #titleHolder #additionalSubpresentersMap #layout
    #visible #extent #styles #millerList #model #lastPageSelectedTabName
    #withHeaderBar)
```

Наступний фрагмент коду показує, як зібрати значення змінних довільного екземпляра класу *StPlayground*. Виконайте його командою «*Print it*».

```
w := StPlayground someInstance.
w class allInstVarNames collect: [:each | each -> (w instVarNamed: each)]
```

Подібним способом можна відібрати екземпляри класу, які мають певні властивості, перебираючи їх за допомогою ітератора *select:*. Наприклад, щоб отримати всі дочірні об'єкти морфи *world* (кореневої морфи відображених графічних елементів), спробуйте виконати наведений нижче вираз.

```
Morph allSubInstances
   select: [ :each |
      | own |
      own := (each instVarNamed: 'owner').
      own isNotNil and: [ own isWorldMorph ] ]
```

18.4. Про застосування рефлексії

Такі вирази зручні для налагодження коду або створення інструментів розробки, але використання їх для розробки звичайних програм є поганою ідеєю: рефлексивні методи порушують інкапсуляцію об'єктів. Вони порушують правила доступу. Вони роблять ваш код набагато складнішим для розуміння та обслуговування. Розумна теза, яку варто запам'ятати: «Разом із суперсилою приходить супервідповідальність». Тому не варто застосовувати рефлексію тільки тому, що це можна зробити.

18.5. Про примітиви

Обидва методи: і *instVarAt:*, і *instVarAt:put:* – примітивні, тобто реалізовані як вбудовані операції віртуальної машини Pharo. Якщо ви ознайомитеся з їхнім кодом, то побачите прагму *<primitive: N>*, де *N* – ціле число. Вона позначає метод, який потрібно опрацьовувати не так, як інші.

```
Object >> instVarAt: index
  "Primitive. Answer a fixed variable in an object. ..."

  <primitive: 173 error: ec>
  self primitiveFailed
```

Будь-який код Pharo розташований після оголошення примітиву виконується лише тоді, коли примітив зазнає невдачі. У цьому конкретному випадку не існує способу реалізувати бажану дію поза примітивом, тому метод запускає виняток.

Багато методів реалізовано примітивами віртуальної машини для швидшого виконання. Наприклад, деякі арифметичні операції над *SmallInteger*.

```
* aNumber
"Primitive. Multiply the receiver by the argument and answer with
 the result if it is a SmallInteger. Fail if the argument or the
 result is not a SmallInteger. Essential. No Lookup. See Object
 documentation whatIsAPrimitive."

<primitive: 9>
^ super * aNumber
```

Якщо віртуальна машина не вміє обробляти тип аргументу, то примітив зазнає невдачі, і керування перейде до коду Pharo. Код методів з примітивами можна змінювати, як і будь-яких інших, але варто пам'ятати, що це може бути ризикованою справою для стабільності цілої системи Pharo.

18.6. Опитування класів та інтерфейсів

Інструменти розробки в Pharo: системний оглядач, налагоджувач, інспектор та інші – всі використовують згадані рефлексивні функції.

Ось ще кілька повідомлень, які можуть бути корисними для створення інструментів розробки.

Повідомлення *isKindOf: aClass* повертає *true*, якщо отримувач є екземпляром *aClass* або одного з його підкласів.

```
1.5 class
>>> SmallFloat64

1.5 isKindOf: Float
>>> true

1.5 isKindOf: Number
>>> true

1.5 isKindOf: Integer
>>> false
```

Повідомлення *respondsTo: aSymbol* повертає *true*, якщо отримувачу доступний метод з селектором *aSymbol*.

```
1.5 respondsTo: #floor
>>> true "бо клас Number реалізує метод floor"

1.5 floor
>>> 1

Exception respondsTo: #,
>>> true "класи винятків можна групувати"
```

Стережіться!

Хоча рефлективні методи особливо корисні для створення інструментів розробки, зазвичай вони не підходять для типових програм. Запит до об'єкта щодо його належності до конкретного класу або щодо розуміння певних повідомлень є типовими ознаками проблем проєктування, бо вони порушують принцип інкапсуляції. Водночас інструменти розробки не є звичайними програмами, бо є частиною самого програмного забезпечення. Саме тому вони мають право глибоко досліджувати внутрішні деталі коду.

18.7. Прості метрики коду

Давайте подивимося, як можна використовувати функції інтроспекції Pharo для швидкого обчислення деяких метрик коду. Метрики коду вимірюють такі аспекти – глибина ієрархії успадкування, кількість безпосередніх або опосередкованих підкласів, кількість методів або змінних екземпляра у кожному класі, або кількість локально визначених методів чи змінних екземпляра. Нижче наведено кілька показників для класу *Morph*, який є надкласом усіх графічних об'єктів у Pharo. Вони засвідчують, що це дуже великий клас, і що він є коренем величезної ієрархії. Такі показники наштовхують на думку, що він потребує деякого рефакторингу!

```
"глибина наслідування"
Morph allSuperclasses size.
>>> 2

"кількість методів"
Morph allSelectors size.
>>> 1436

"кількість змінних екземпляра"
Morph allInstVarNames size.
>>> 6
```

```
"кількість нових методів"
Morph selectors size.
>>> 931

"кількість нових змінних"
Morph instVarNames size.
>>> 6

"безпосередні підкласи"
Morph subclasses size.
>>> 73

"загальна кількість підкласів"
Morph allSubclasses size.
>>> 448

"кількість рядків коду"
Morph linesOfCode.
>>> 5088
```

Однією з найцікавіших метрик в області об’єктно-орієнтованих мов є кількість методів, які розширюють методи, успадковані від надкласу. Це інформує нас про відношення між класом і його надкласами. У наступних параграфах ми з’ясуємо, як використати наші знання про структуру часу виконання, щоб відповісти на такі запитання.

18.8. Дослідження екземплярів

У Pharo все є об'єктом. Зокрема, класи – це об’єкти, які надають корисні функції для відшукання їхніх екземплярів. Більшість повідомлень, які ми зараз розглянемо, реалізовані в *Behavior*, тому їх розуміють усі класи.

Наприклад, можна отримати випадковий екземпляр певного класу, надіславши йому повідомлення *someInstance*.

```
Point someInstance
>>> (-1@-1)
```

Також можна зібрати всі екземпляри за допомогою *allInstances* або довідатися кількість активних екземплярів у пам’яті за допомогою *instanceCount*.

```
ByteString allInstances
>>> #('collection' 'position' ...)

ByteString instanceCount
>>> 54837

String allSubInstances size
>>> 147289
```

Згадані методи отримують доступ до екземплярів, які зберігаються всередині методів. Оскільки Pharo має близько 130 000 методів, то такі цифри не видаються божевільними.

18.9. Від методів до змінних екземпляра

Описані нижче функції можуть бути дуже корисними для налагодження програми, тому що можна попросити клас перерахувати ті його методи, які демонструють певні властивості. Ось ще кілька цікавих і корисних методів для дослідження коду через рефлексію.

whichSelectorsAccess: повертає список усіх селекторів методів, які читають або записують змінну екземпляра, ім'я якої задано аргументом.

whichSelectorsStoreInto: повертає селектори методів, які змінюють значення змінної екземпляра.

whichSelectorsReferTo: повертає селектори методів, які надсилають задане аргументом повідомлення.

```
Point whichSelectorsAccess: 'x'
>>> #(#octantOf: #roundDownTo: #+ #asIntegerPoint #transposed ...)

Point whichSelectorsStoreInto: 'x'
>>> #(#fromSton: #setX:setY: #setR:degrees: #bitShiftPoint:)

Point whichSelectorsReferTo: #+
>>> #(#+)
```

Наведені нижче методи переглядають ланцюжок наслідування.

whichClassIncludesSelector: повертає надклас, який реалізує задане аргументом повідомлення.

unreferencedInstanceVariables повертає список змінних екземпляра, які не використовуються ні в класі отримувача, ні в будь-якому з його підкласів.

```
Rectangle whichClassIncludesSelector: #inspect
>>> Object

Rectangle unreferencedInstanceVariables
>>> #()
```

18.10. Про *SystemNavigation*

SystemNavigation – це фасад, який підтримує різні корисні методи для запитів до вихідного коду системи і відшукання методів за заданим критерієм. *SystemNavigation default* повертає екземпляр, який можна використовувати для навігації системою.

```
SystemNavigation default allClassesImplementing: #yourself
>>> an OrderedCollection(Object)
```

Призначення наведених нижче повідомлень описано їхніми селекторами.

```
SystemNavigation default allSentMessages size
>>>43985

(SystemNavigation default allUnsentMessagesIn: Object selectors) size
>>> 44
```

```
SystemNavigation default allUnimplementedCalls size
>>> 335
```

Зауважимо, що реалізовані методи, для яких не надіслані повідомлення, не обов'язково надлишкові, оскільки повідомлення можуть бути надіслані неявно (наприклад, за допомогою *perform:*). Проблематичніші повідомлення, надіслані, але не реалізовані, бо методи, які надсилають ці повідомлення, не будуть виконані. Це може бути ознакою незавершеної реалізації, застарілих API або відсутності бібліотек. Вони також часто трапляються в тестах щодо реалізації інструментів.

Point allCallsOn повертає колекцію всіх випадків явного надсилання повідомлень класу *Point*. Екземпляри такої колекції мають вигляд «*Ім'яКласу>>селекторМетоду*».

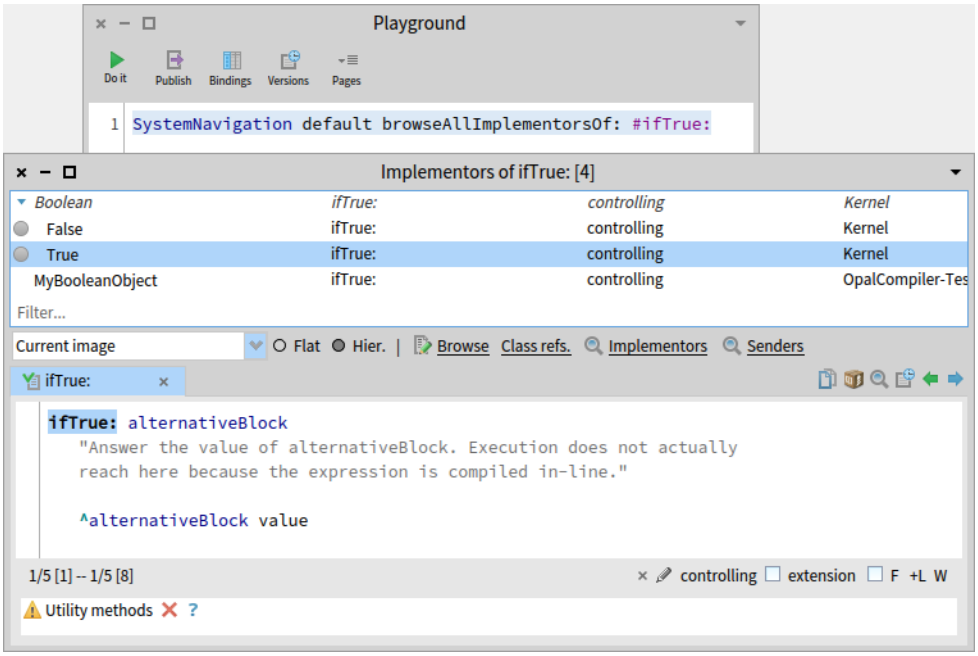

Рис. 18.3. Відшукання всіх реалізацій методу *ifTrue:*

Усі згадані методи інтегровані в середовище програмування Pharo, зокрема, в Оглядач коду. Як вже згадували раніше, існують зручні комбінації клавіш для перегляду всіх реалізаторів [*Cmd-M*] і відправників [*Cmd-N*] кожного повідомлення. Можливо, не так широко відомо, що багато таких вбудованих запитів, реалізовано у вигляді методів класу *SystemNavigation* у протоколі *query*. Наприклад, усі реалізатори повідомлення *ifTrue:* можна переглянути програмно, надіславши відповідне повідомлення (див. рис. 18.3).

```
SystemNavigation default browseAllImplementorsOf: #ifTrue:
```

Особливо корисними є методи *browseAllSelect:* і *browseMethodsWithSourceString:matchCase:*. Ось два різні способи перегляду всіх методів у системі, які надсилають повідомлення до *super* (перший спосіб – це скоріше груба сила, другий – кращий, усуває деякі помилкові спрацьовування).

```
SystemNavigation default
   browseMethodsWithSourceString: 'super'
   matchCase: true

SystemNavigation default
   browseAllSelect: [:method | method sendsToSuper ]
```

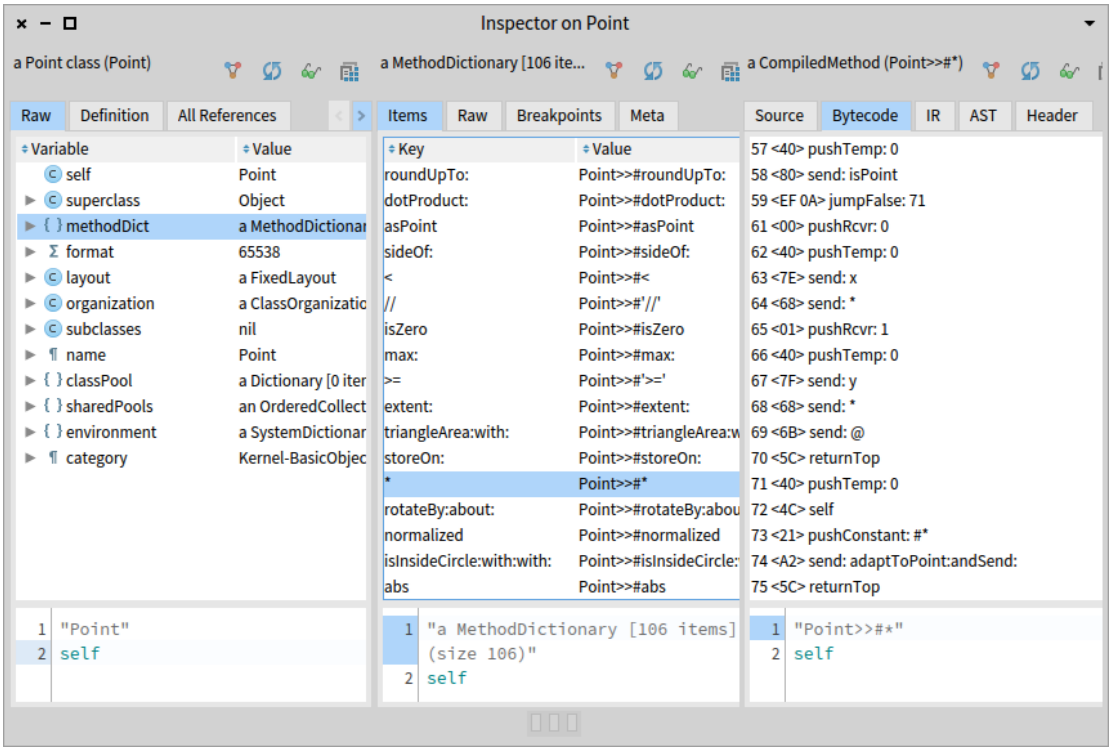

Рис. 18.4. Інспектування класу *Point* і байт-коду його методу #*

18.11. Класи, словники методів і методи

Класи є об'єктами, тому їх можна інспектувати або досліджувати, як і будь-який інший об'єкт.

Виконайте вираз *Point inspect*. На рис. 18.4 інспектор показує структуру класу *Point*. Видно, що клас зберігає свої методи в словнику, ключами якого є селектори. Селектор *#** вказує на декомпільований байт-код методу *Point >> **.

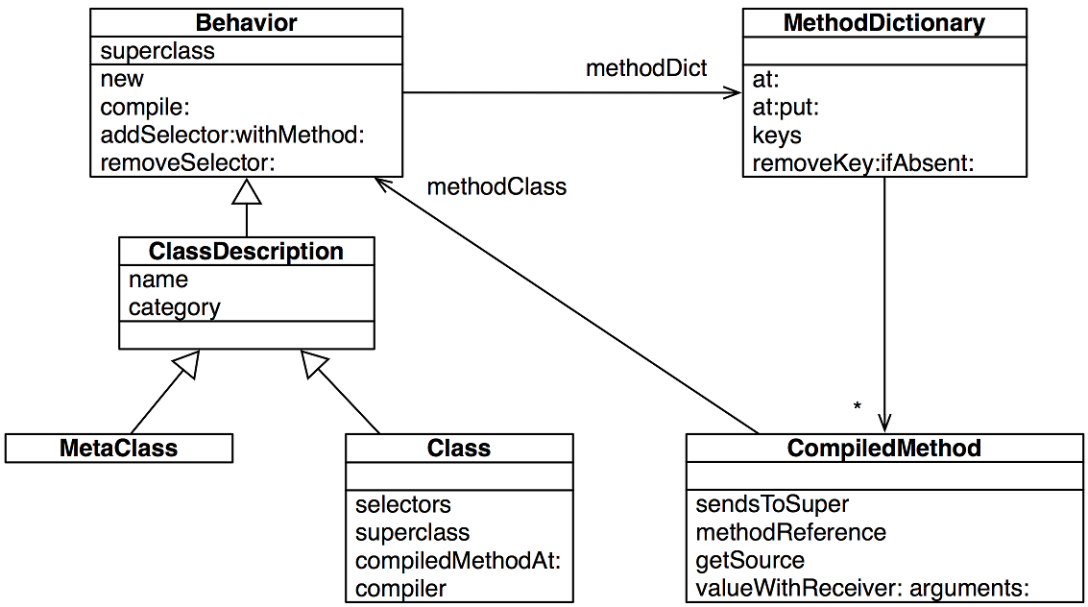

Рис. 18.5. Класи, словники методів і компільовані методи

Розглянемо зв'язок між класами та методами. На рис. 18.5 видно, що класи та метакласи мають спільний надклас *Behavior*. Тут визначено *new* серед інших ключових методів класів. Кожен клас має словник методів, який відображає селектори методів на

скомпільовані методи. Кожен скомпільований метод знає клас, якому він належить. На рис. 18.4 навіть можна побачити декомпільовані байт-коди методу.

Зв'язки між класами та методами можна використовувати для формування запитів щодо системи. Наприклад, щоб дізнатися, які методи класу не перевизначають методи надкласу, можна перейти від класу до словника методів, як зображено нижче, і вибрати потрібне.

```
| aClass |
aClass := SmallInteger.
aClass methodDict keys select: [ :aMethod |
   (aClass superclass canUnderstand: aMethod) not ]
>>> an IdentitySet(#threeDigitName #printStringBase:nDigits: ...)
```

Скомпільований метод – об'єкт, який не тільки зберігає байт-код методу, а також надає численні корисні методи для запитів до системи. Одним із таких методів є *isAbstract* (який повідомляє, чи надсилає метод повідомлення *subclassResponsibility*). Його можна використати для ідентифікації всіх абстрактних методів класу.

```
| aClass |
aClass := Number.
aClass methodDict keys select: [ :aMethod |
   (aClass >> aMethod) isAbstract ]
>>> #(#* #asFloat #storeOn:base: #printOn:base: #+ #round: #/ #-
    #adaptToInteger:andSend: #nthRoot: #sqrt #adaptToFraction:andSend:)
```

Зверніть увагу, що цей код надсилає класові повідомлення >>, щоб за заданим селектором отримати скомпільований метод.

Щоб переглянути всі надсилання повідомлень до *super* в межах певної ієрархії, наприклад, в межах ієрархії колекцій, можна сформувати складніший запит. Для зручного перегляду результатів повідомлення *browseMessageList:name:* відкриє спеціальне вікно (екземпляр класу *ClyOldMessageBrowserAdapter*).

```
class := Collection.
SystemNavigation default
   browseMessageList: (class withAllSubclasses gather: [ :each |
      each methodDict associations
         select: [:assoc | assoc value sendsToSuper ]
         thenCollect: [:assoc |
            RGMethodDefinition realClass: each selector: assoc key]])
   name: 'Supersends of ', class name, ' and its subclasses'
```

Зверніть увагу на те, як перейшли від класів до словників методів, а потім до скомпільованих методів, щоб відібрати ті з них, які нас цікавлять. RGMethodDefinition – це легкий проксі-сервер для скомпільованого методу, який використовується багатьма інструментами. З екземпляра скомпільованого методу легко отримати посилання на сам метод за допомогою *CompiledMethod >> methodReference*.

```
(Object >> #=) methodReference selector
>>> #=
```

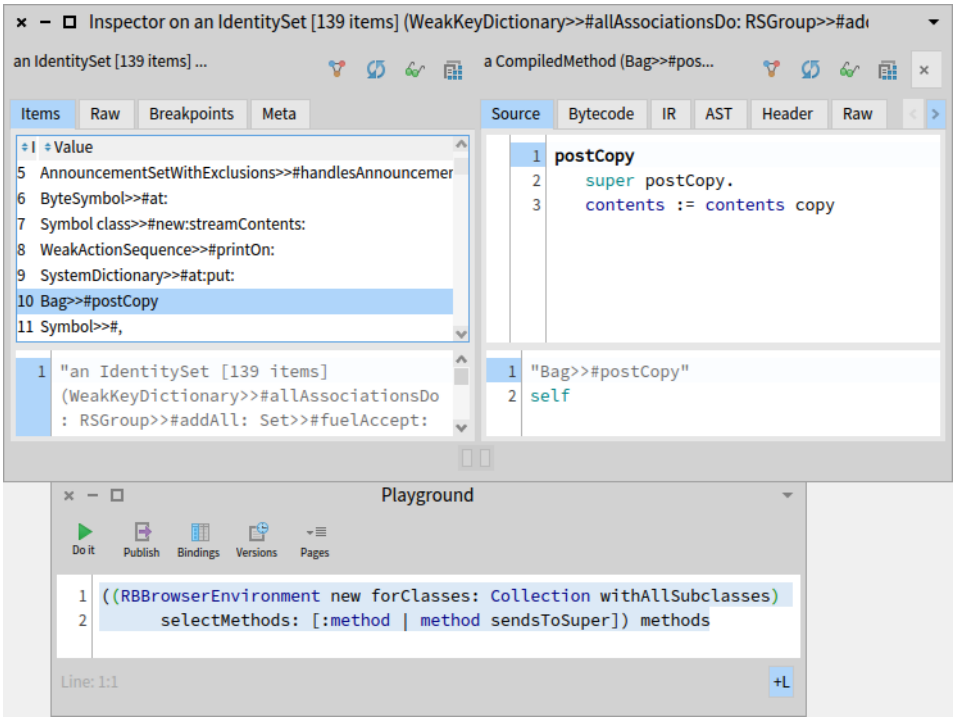

Рис. 18.6. Інспектування методів, які надсилають повідомлення до *super*

18.12. Середовища перегляду

SystemNavigation надає зручні засоби програмного формування запитів і перегляду системного коду, проте існують й інші способи. Інтегрований у Pharo Системний оглядач дає змогу обмежити середовище для виконання пошуку.

Припустимо, нам потрібно дізнатися, які класи посилаються на клас *Point*, але тільки з його власного пакета.

Відкрийте Оглядача на класі *Point*. Клацніть на пакеті верхнього рівня *Kernel* у панелі пакетів і увімкніть перемикач ***Scoped View***. Оглядач покаже лише пакет *Kernel* і всі класи в ньому (і деякі класи, які мають методи розширення з цього пакета). Тоді відкрийте контекстне меню класу *Point* і виберіть команду «*Class refs. Cmd+N*». Вона покаже всі методи, які мають посилання на клас *Point*, точніше, ті з них, які належать до пакета *Kernel*. Порівняйте отримане з результатом пошуку в Оглядачі без обмежень.

Таку ділянку пошуку називають *середовищем перегляду* (клас *RBBrowserEnvironment* – базовий для ієрархії класів, які реалізують середовища перегляду). Усі інші пошуки, як-от пошук *відправників повідомлення* або *реалізацій методу* в Оглядачі, також будуть обмежені цим середовищем.

Середовище перегляду можна створити також програмно. Ось, наприклад, як створити нове середовище *RBBrowserEnvironment* для класу *Collection* та його підкласів, вибрати методи, що надсилають повідомлення до *super*, і переглянути отримане середовище.

```
((RBBrowserEnvironment new forClasses: Collection withAllSubclasses)
      selectMethods: [:method | method sendsToSuper]) methods
```

Зверніть увагу, що цей фрагмент значно компактніший, ніж попередній еквівалентний приклад із використанням *SystemNavigation*. Щоб побачити результат запиту, відкрийте його в інспекторі: виконайте його командою «*Cmd+I*».

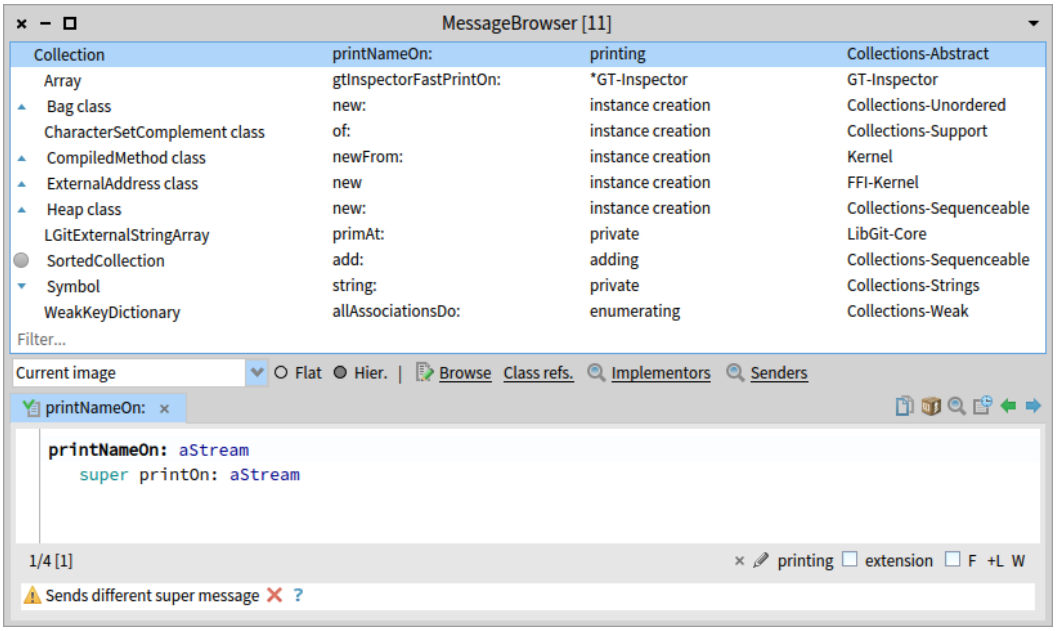

Рис. 18.7. Інспектування методів, які надсилають повідомлення до *super*

Нарешті, програмно можна знайти ті методи, які надсилають до *super* повідомлення, що відрізняються від селектора самого методу.

```
Smalltalk tools messageList browse:
((RBBrowserEnvironment new forClasses: (Collection withAllSubclasses))
   selectMethods: [:method | method sendsToSuper and:
      [(method parseTree superMessages includes: method selector) not]
   ]) methods
```

Тут у кожного скомпільованого методу запитують його дерево аналізу (*Refactoring Browser*), щоб з'ясувати, чи повідомлення до *super* відрізняються від селектора методу. Для перегляду результатів використано оглядача повідомлень. На рис. 18.7 зображено, що в ієрархії *Collection* було знайдено одинадцять таких методів, враховуючи *Collection >> printNameOn:*, який надсилає *super printOn:*.

Перегляньте протокол *querying* класу *RBProgramNode*, щоб побачити, що можна запитати в дерева аналізу.

18.13. Прагми – анотації методів

Прагма – це анотація методу, яка зазначає дані про програму, але не бере участі у виконанні програми. Вона не має прямого впливу на роботу методу, який анотує. Прагми застосовують, зокрема, для постачання інформації компіляторові та для опрацювання на етапі виконання.

Інформація для компілятора. Компілятор може використати прагму, щоб зробити виклик методу примітивною функцією. Така функція має бути визначена віртуальною машиною або зовнішнім плагіном.

Опрацювання на етапі виконання. Деякі прагми доступні для перевірки під час виконання.

У методі можна оголосити одну або кілька прагм, оголошення прагм розташовують перед першим виразом Pharo. Внутрішньо прагма – це статичне повідомлення, надіслане з літеральними аргументами.

Ми вже бачили в цьому розділі окремі прагми, коли говорили про примітиви. Примітив – це не що інше, як оголошення прагми. Розглянемо вираз *<primitive: 173 error: ec>*, визначений у методі *Object >> #instVarAt:*. Селектор прагми – *primitive:error:*, а його аргументи – буквальне значення літерала *173* і змінна *ec*, код помилки, який заповнює віртуальна машина, якщо виконання реалізації зазнає невдачі.

Компілятор, ймовірно, найбільший користувач прагм. SUnit – ще один інструмент, який використовує анотації. SUnit може оцінити охоплення програми тестами за кодом модульних тестів. Окремі методи можна виключити з оцінки охоплення за допомогою відповідної прагми, як у методі *documentation* метакласу *SplitJointTest class*:

```
SplitJointTest class >> documentation
  <ignoreForCoverage>
  "self showDocumentation"

  ^ 'This package provides function.... "
```

Анотуючи метод прагмою *<ignoreForCoverage>*, можна керувати сферою охоплення.

Прагми – це об'єкти першого класу, екземпляри класу *Pragma*. Скомпільований метод відповідає на повідомлення *pragmas*, повертаючи масив прагм.

```
(SplitJoinTest class >> #documentation) pragmas.
>>> an Array(<ignoreForCoverage>)
```

```
(SmallFloat64>>#+) pragmas
>>> an Array(<primitive: 541>)
```

За допомогою повідомлення *allNamed:in:* класові *Pragma* можна перебрати методи, що містять певну прагму. Наведений нижче приклад демонструє, що на стороні класу *SplitJoinTest* є два методи, анотованих *<ignoreForCoverage>*.

```
Pragma allNamed: #ignoreForCoverage in: SplitJoinTest class
>>> an Array(<ignoreForCoverage> <ignoreForCoverage>)
```

18.14. Доступ до контексту етапу виконання

Ми побачили, як інтроспективні можливості Pharo дають змогу формувати запити та досліджувати об'єкти, класи та методи. Але як щодо середовища виконання?

Контексти методу

Контекст етапу виконання запущеного (на виконання) методу фактично перебуває у віртуальній машині – його взагалі немає в образі! З іншого боку, налагоджувач, вочевидь, має доступ до цієї інформації, і ми можемо з задоволенням і користю досліджувати контекст виконання, як і будь-який інший об'єкт. Як таке можливо?

Насправді нічого чарівного в налагоджувачі немає. Секрет полягає в псевдозмінній *thisContext*, з якою ми раніше ознайомилися лише побіжно. Щоразу, коли у запущеному

методі трапляється звертання до *thisContext*, увесь контекст виконання цього методу втілюється та стає доступним образові як ряд пов'язаних екземплярів класу *Context*.

Ми можемо легко поекспериментувати з цим механізмом самі.

Змініть визначення *Integer >> slowFactorial*, вставивши вираз «*thisContext inspect. self halt.*», як зображено нижче.

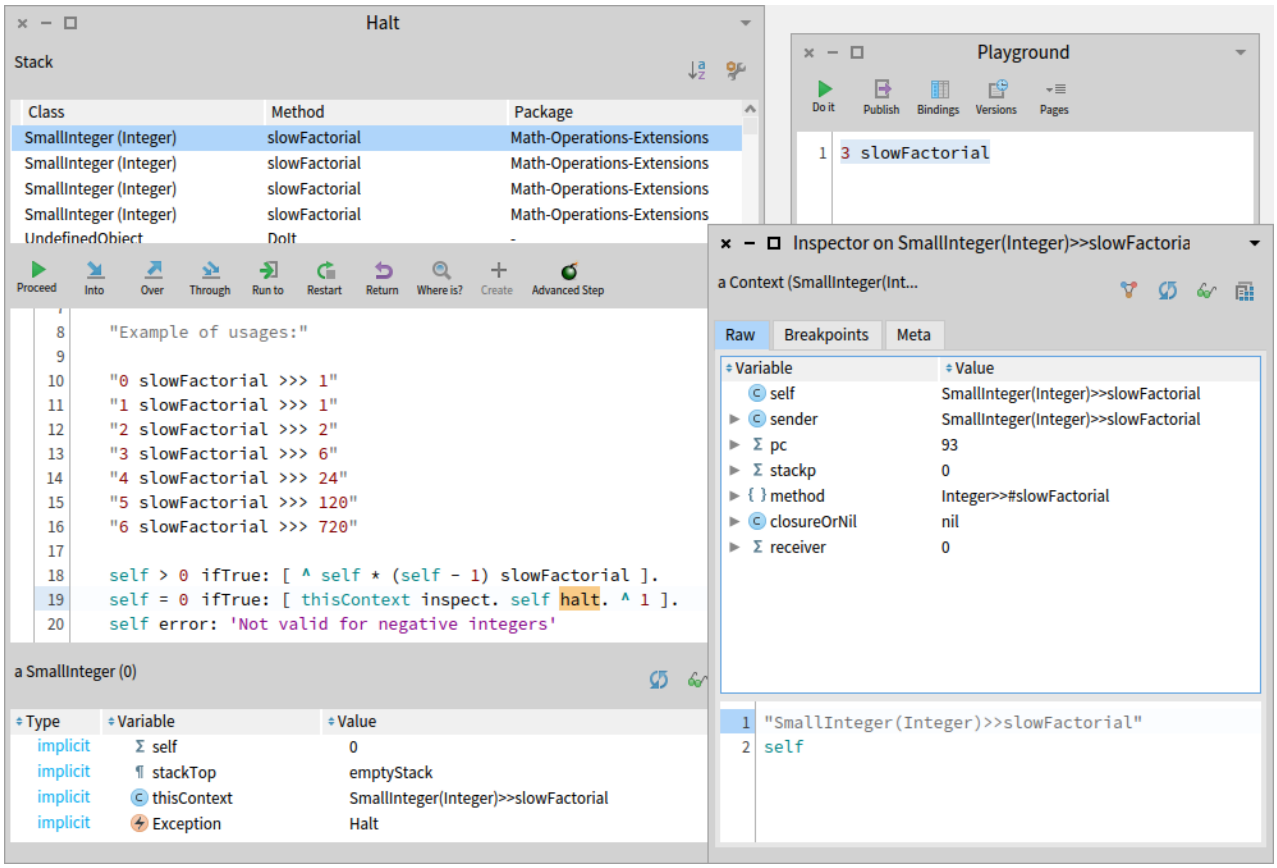

Рис. 18.8. Інспектування *thisContext*

```
Integer >> slowFactorial
  "Повертає факторіал отримувача."

  self > 0 ifTrue: [^ self * (self - 1) slowFactorial].
  self = 0 ifTrue: [thisContext inspect. self halt. ^ 1].
  self error: 'Not valid for negative integers'
```

Тепер виконайте *3 slowFactorial* у Робочому вікні. Ви мали б отримати і вікно налагоджувача, і вікно інспектора, як показано на рис. 18.8.

Інспектування *thisContext* дає повний доступ до поточного контексту виконання, стеку викликів, локальних змінних і аргументів, ланцюжка відправників і отримувача. Ласкаво просимо до налагоджувача бідної людини! Якщо тепер переглянути клас досліджуваного об'єкта (тобто, виконати «*self browse*» у нижній панелі інспектора), то виявиться, що це екземпляр класу *Context*, як і кожен відправник у ланцюжку.

Змінну *thisContext* не призначено для щоденного програмування, але вона важлива для створення таких інструментів, як налагоджувачі, і для доступу до інформації про стек викликів. Щоб дізнатися, які методи використовують *thisContext*, виконайте такий вираз (його виконання може зайняти трохи часу).

```
SystemNavigation default
   browseMethodsWithSourceString: 'thisContext' matchCase: true
```

Як з'ясувалося, одним із найпоширеніших застосувань є виявлення відправника повідомлення. Типове застосування полягає в наданні повнішої інформації розробнику. Розглянемо приклад. За домовленістю метод, який надсилає *self subclassResponsibility*, вважається абстрактним. Але як *Object >> subclassResponsibility* надає вичерпне повідомлення про помилку і зазначає, який абстрактний метод було викликано? Дуже просто, запитавши *thisContext* про відправника.

```
subclassResponsibility
"This message sets up a framework for the behavior of the class'
 subclasses. Announce that the subclass should have implemented this
 message."

SubclassResponsibility signalFor: thisContext sender selector
```

18.15. Інтелектуальні контекстні точки переривання

Щоб задати точку переривання у Pharo, в потрібному місці методу записують «*self halt*». Виконання такого виразу призводить до втілення *thisContext*, і відкривання вікна налагоджувача в точці переривання. На жаль, це створює проблеми для методів, які інтенсивно використовуються в системі.

Припустимо, наприклад, що потрібно дослідити виконання *Morph >> openInWorld*. Визначити точку переривання в цьому методі досить проблематично.

Будьте уважні, наступний експеримент зруйнує все! Створіть *новий* образ системи та задайте таку точку переривання, як подано нижче.

```
Morph >> openInWorld
   "Add this morph to the world."
   self halt.
   self openInWorld: self currentWorld
```

Зверніть увагу, як система відразу зависає, щойно ви намагаєтеся відкрити будь-яку нову морфу (меню/вікно/...)! Ви навіть не отримаєте вікно налагоджувача. Проблема стане зрозумілою, коли ми додумаємося, що, по-перше, *Morph >> openInWorld* використовується багатьма частинами системи, тому точка переривання спрацьовує дуже скоро після взаємодії з інтерфейсом користувача; по-друге, *сам налагоджувач* надсилає *openInWorld*, як тільки пробує відкрити вікно, запобігаючи відкриттю налагоджувача! Потрібен спосіб умовної зупинки, який перериватиме виконання лише за умови перебування в певному контексті. Це саме те, що надає *Object >> haltIf:*.

Припустимо тепер, що потрібно зупинитися, лише якщо *openInWorld* надсилається, скажімо, з контексту *MorphTest >> testOpenInWorld*.

Знову запустіть новий образ та встановіть таку точку зупинки:

```
Morph >> openInWorld
   "Add this morph to the world."
   self haltIf: #testOpenInWorld.
   self openInWorld: self currentWorld
```

Цього разу образ не зависає. Спробуйте запустити *MorphTest*. Він зупиниться та відкриє налагоджувача.

```
MorphTest run: #testOpenInWorld.
```

Як це працює? Давайте розглянемо *Object >> haltIf:*. Він надсилає повідомлення *if:* з умовою класу винятків *Halt*. Умовою може бути логічне значення, блок або символ. Метод *Halt class >> if:* сам перевірить, чи умова є символом, а в нашому прикладі так і є, і тому виконає «*self haltIfCallChain: thisContext home sender contains: condition*». Текст відповідного методу наведено нижче.

```
Object >> haltIf: condition
  <debuggerCompleteToSender>
  Halt if: condition.
```

```
Halt class >> haltIfCallChain: haltSenderContext contains: aSelector
  | cntxt |
  cntxt := haltSenderContext.
  [ cntxt isNil ] whileFalse: [
     cntxt selector = aSelector ifTrue: [ self signalIn:
        haltSenderContext ]. cntxt := cntxt sender ]
```

Починаючи з *thisContext*, *haltIfCallChain:contains:* проходить стеком виконання і перевіряє, чи ім'я викликаного методу збігається з переданим аргументом. Якщо це так, то він сам сигналізує про виняток, який за замовчуванням відкриває налагоджувача.

Аргументом для *haltIf:* можна також надати логічне значення або логічний блок, але це прості випадки, які не використовують *thisContext*.

18.16. Перехоплення незрозумілих повідомлень

У попередніх параграфах рефлексивні функції Pharo використовували здебільшого для формування запитів і дослідження об'єктів, класів, методів і стеку на етапі виконання. Тепер розглянемо, як використати знання про структуру системи для перехоплення повідомлень і зміни поведінки під час виконання.

Коли об'єкт отримує повідомлення, то для його опрацювання шукає відповідний метод спочатку в словнику методів свого класу, а якщо такого методу немає, то продовжує пошук ієрархією класів догори, доки не досягне *Object*. Якщо метод для цього повідомлення так і не знайдеться, то об'єкт *надішле сам собі* повідомлення *doesNotUnderstand:* і передасть селектор незрозумілого повідомлення як аргумент. Потім процес пошуку розпочнеться заново, доки не буде знайдено *Object >> doesNotUnderstand:* і не буде запущено налагоджувач.

Але що, якщо *doesNotUnderstand:* перевизначено одним із підкласів *Object* на шляху пошуку? Виявляється, це зручний спосіб реалізації певних видів дуже динамічної поведінки. Перевизначивши *doesNotUnderstand:*, об'єкт, який не розуміє повідомлення, може вдатися до альтернативної стратегії, щоб відповісти на нього.

Є два дуже поширених застосування цієї техніки: по-перше, реалізація легких проксі-серверів (обгорток) для об'єктів; по-друге, динамічна компіляція або завантаження коду, якого бракує. Це те, про що йтиметься в наступних параграфах.

18.17. Легкий проксі-сервер

Щоб реалізувати легкий проксі, вводять *мінімальний об'єкт*, який діятиме як обгортка існуючого об'єкта. Оскільки обгортка практично не реалізує власних методів, будь-яке надіслане їй повідомлення перехоплює функція *doesNotUnderstand:*. Реалізуючи це повідомлення, обгортка може виконувати спеціальні дії перед тим, як делегувати повідомлення справжньому суб'єкту, для якого вона є проксі-сервером.

Давайте розглянемо, як це можна зробити. Визначаємо *LoggingProxy*, як подано нижче.

```
ProtoObject subclass: #LoggingProxy
   instanceVariableNames: 'subject invocationCount'
   classVariableNames: ''
   package: 'PBE-Reflection'
```

Зауважте, що оголошено підклас *ProtoObject*, а не *Object*, щоб запобігти успадкуванню від *Object* більше 470 (!) методів.

```
Object methodDict size
>>> 473
```

Така обгортка має дві змінні екземпляра: об'єкт *subject*, для якого вона проксі, і *count* – кількість перехоплених повідомлень. Потрібно ініціалізувати обидві змінні екземпляра та надати метод читання лічильника повідомлень. Спочатку змінна *subject* вказує на сам проксі-об'єкт.

```
LoggingProxy >> initialize
  invocationCount := 0.
  subject := self.

LoggingProxy >> invocationCount
  ^ invocationCount
```

Завдання обгортки – перехоплювати всі повідомлення, друкувати їх у *Transcript*, оновлювати лічильник повідомлень і пересилати повідомлення справжньому об'єктові.

```
LoggingProxy >> doesNotUnderstand: aMessage
  Transcript show: 'performing ', aMessage printString; cr.
  invocationCount := invocationCount + 1.
  ^ aMessage sendTo: subject
```

А тепер трохи магії. Створимо новий об'єкт *Point* і новий об'єкт *LoggingProxy*, а тоді попросимо проксі стати точкою!

```
point := 1@2.
LoggingProxy new become: point.
```

Виконання повідомлення *become:* спричиняє обмін посилань: заміни всіх посилань на об'єкт, на який вказує змінна *point*, на посилання на екземпляр проксі та навпаки. Найважливіше те, що змінна *subject* екземпляра проксі тепер посилається на точку!

```
point invocationCount
>>> 0
```

```
point + (3@4)
>>> (4@6)

point invocationCount
>>> 1
```

В консолі з'явиться рядок «*performing + (3@4)*».

У більшості випадків це працює добре, але є деякі недоліки.

```
point class
>>> LoggingProxy
```

Насправді метод *class* реалізовано в *ProtoObject*, але навіть якби він був реалізований в *Object*, від якого *LoggingProxy* не наслідує, повідомлення *class* фактично не надсилається до обгортки або до її суб'єкта. На повідомлення безпосередньо відповідає віртуальна машина. *yourself* також ніколи не надсилають по-справжньому.

Нижче перелічено інші повідомлення, які залежно від отримувача може інтерпретувати безпосередньо віртуальна машина.

```
+- < > <= >= = ~= * / \\ =\=
@ bitShift: // bitAnd: bitOr:
at: at:put: size
next nextPut: attend
blockCopy: value value: do: new new: x y
```

Ніколи не надсилаються повідомлення, методи яких компілятор перетворює на вбудовані байт-коди порівняння та переходу. Нижче перелічено їхні селектори.

```
ifTrue: ifFalse: ifTrue:ifFalse: ifFalse:ifTrue:
and: or:
whileFalse: whileTrue: whileFalse whileTrue
to:do: to:by:do:
caseOf: caseOf:otherwise:
ifNil: ifNotNil: ifNil:ifNotNil: ifNotNil:ifNil:
```

Спроби надіслати ці повідомлення об'єктові невідповідного типу зазвичай призводять до винятку з боку віртуальної машини, оскільки вона не може використовувати вбудовану диспетчеризацію для неправильних отримувачів. Можна перехопити виняток і визначити належну поведінку, наприклад, перевизначивши *mustBeBoolean* в отримувачі, або перехопивши виняток *NonBooleanReceiver*.

Навіть якщо можемо не брати до уваги надсилання таких спеціальних повідомлень, то існує ще одна фундаментальна проблема, яку неможливо подолати за допомогою описаного підходу: не вдасться перехопити надсилання *self*.

```
point := 1@2.
LoggingProxy new become: point.
point invocationCount
>>> 0

point rectangle: (3@4)
>>> (1@2) corner: (3@4)
```

```
point invocationCount
>>> 1
```

У консолі з'явиться рядок «*performing rectangle: (3@4)*», а інформації про доступ до полів точки немає. Використання *self* аргументом повідомлення в методі *rectangle:* обмануло обгортку.

```
Point >> rectangle: aPoint
"Answer a Rectangle that encompasses the receiver and aPoint.
 This is the most general infix way to create a rectangle."

^ Rectangle
   point: self
   point: aPoint
```

Порівняйте отримане раніше з результатом створення такого самого прямокутника без використання *self*.

```
point := 1@2.
LoggingProxy new become: point.
point invocationCount
>>> 0

Rectangle point: point point: (3@4)
>>> (1@2) corner: (3@4)

point invocationCount
>>> 4
```

У консолі – чотири повідомлення про доступ до полів: два до *x*, і два до *y*.

За допомогою описаної техніки можна перехоплювати повідомлення, але потрібно пам'ятати про природні обмеження використання проксі-серверів. У параграфі 18.19 описано інший, загальніший підхід до перехоплення повідомлень.

Від перекладача. Автори книги описали одну загальну обгортку, яку можна використовувати з об'єктами різних типів. Зрозуміло, що на стільки загальна сутність однаково реагує на всі повідомлення: просто заносить їх до протоколу. Поміркуйте, як за допомогою перевизначення певних методів у проксі задати спеціальну поведінку об'єкта-обгортки. Наприклад, як убезпечитися від використання точок з від'ємними координатами під час створення прямокутників.

18.18. Генерування методів, яких бракує

Іншим поширеним застосуванням перехоплення незрозумілих повідомлень є динамічне завантаження або створення методів, яких бракує. Уявіть дуже велику бібліотеку класів з багатьма методами. Замість того, щоб завантажувати всю бібліотеку, ми могли б завантажити заглушку для кожного класу бібліотеки. Заглушки знають, де знайти вихідний код усіх своїх методів, перехоплюють усі незрозумілі повідомлення та на вимогу динамічно генерують методи, яких немає. У якийсь момент цю поведінку можна деактивувати, а завантажений код зберегти як мінімально необхідну для клієнтської програми підмножину.

Давайте розглянемо простий варіант цієї методики на прикладі класу, який на вимогу автоматично додає методи читання своїх змінних екземпляра. Логіка поведінки така: перехоплюємо кожне незрозуміле повідомлення, якщо існує змінна екземпляра з таким іменем, як селектор перехопленого повідомлення, та просимо клас скомпілювати метод доступу до неї і повторно надсилаємо те саме повідомлення.

```
Object subclass: #DynamicAccessors
   instanceVariableNames: 'x'
   classVariableNames: ''
   package: 'PBE-Reflection'
```

```
DynamicAcccessors >> doesNotUnderstand: aMessage
  | messageName |
  messageName := aMessage selector asString.
  (self class instVarNames includes: messageName)
     ifTrue: [
        self class compile: messageName, String cr, ' ^ ', messageName.
        ^ aMessage sendTo: self ].
  ^ super doesNotUnderstand: aMessage

DynamicAcccessors >> initialize
  x := 55
```

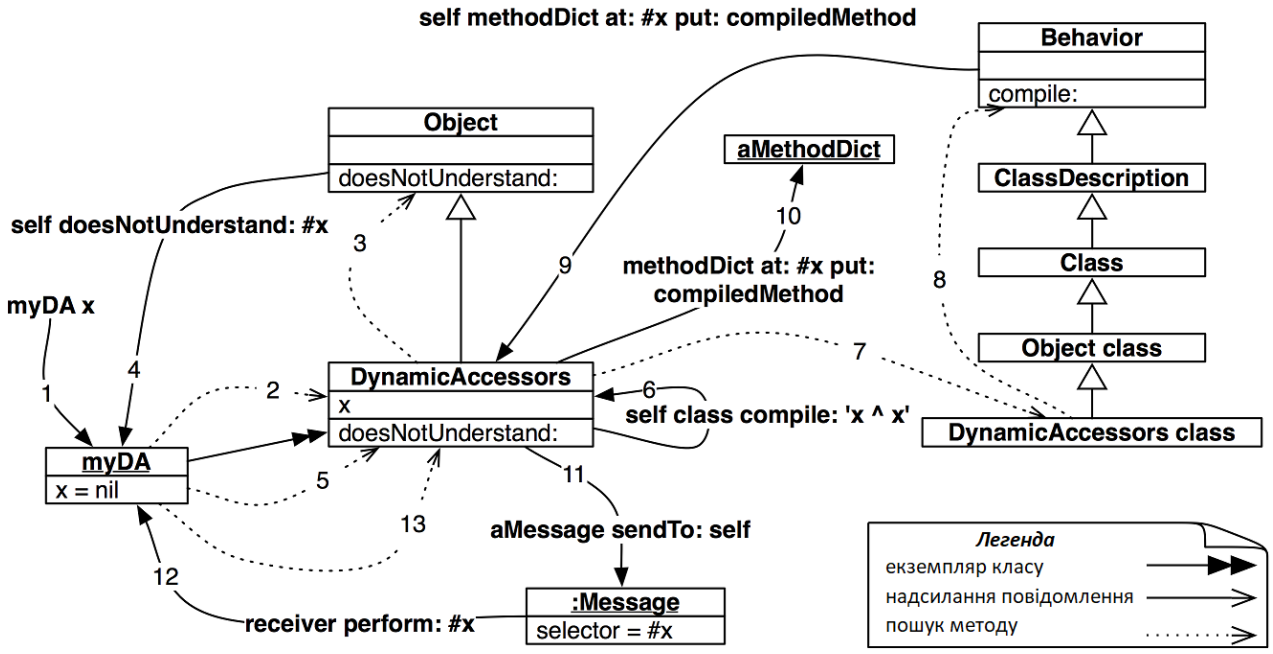

Рис. 18.9. Динамічне створення методів доступу

Припустимо, що клас *DynamicAccessors* має змінну екземпляра *x*, але не має попередньо визначеного методу доступу. Наведений нижче код динамічно згенерує такий метод і отримає значення змінної.

```
myDA := DynamicAccessors new.
myDA x
>>> 55
```

Давайте розглянемо, що відбувається, коли повідомлення *x* вперше надсилають екземплярові *DynamicAccessors* (див. рис. 18.9).

На рисунку номерами позначено послідовні кроки опрацювання повідомлення. Пояснимо кожен з них. (1) Повідомлення *x* надсилаємо до *myDA*. (2) Відбувається пошук методу в класі та (3) в ієрархії класів, пошук зазнає невдачі. (4) Це призводить до того, що «*self doesNotUnderstand: #x*» повертається до об’єкта (5) і запускає новий пошук. Цього разу метод *doesNotUnderstand:* відразу знайдено в *DynamicAccessors*, (6) він просить свій клас скомпілювати рядок *'x ^ x'*. Метод *compile:* шукається (7) в ієрархії метакласів і (8) нарешті знайдено в *Behavior*. Він (9-10) додає новий скомпільований метод до словника методів *DynamicAccessors*. Нарешті, (11-13) повідомлення надсилається повторно (за допомогою *sendTo:*), і цього разу метод доступу знайдено.

Якщо початкове надіслане повідомлення не збігається з іменами змінним екземпляра, то за допомогою *super* керування передається методові *doesNotUnderstand:* за замовчуванням. Таку саму техніку можна використовувати для генерування модифікаторів змінних екземпляра або інших видів шаблонного коду.

Про повідомлення *sendTo:*

Метод *sendTo:* реалізовано в класі *Message*, як подано нижче.

```
Message >> sendTo: receiver
   "повертає результат надсилання себе до отримувача"

   ^ receiver perform: selector withArguments: args
```

Метод *Object >> perform:* можна використовувати для надсилання повідомлень, створених на етапі виконання:

```
5 perform: #factorial
>>> 120

6 perform: ('fac', 'torial') asSymbol
>>> 720

4 perform: #max: withArguments: (Array with: 6)
>>> 6
```

18.19. Об’єкти як обгортки методів

Ми вже бачили, що скомпільовані методи є об’єктами в Pharo. Вони підтримують низку методів, які дають змогу програмісту надсилати запити до системи виконання.

От що, напевно, може викликати здивування, так це те, що будь-який об’єкт може відігравати роль скомпільованого методу та бути значенням у словнику методів. Все, що такий об’єкт має зробити, це відповісти на повідомлення *run:with:in:* і кілька інших важливих повідомлень. За допомогою такого механізму можна створювати інші шпигунські інструменти. Підхід можна описати так: створити об’єкт, який посилається на оригінальний скомпільований метод, щоб можна було його виконати, та замінити метод в словнику методів таким об’єктом (так званою обгорткою методу). Об’єкт-обгортка може виконувати різну додаткову роботу, наприклад, реєструвати, чи було виконано метод, скільки разів, коли він був виконаний.

Випробуємо такий підхід обгортання методів. Pharo постачається з простим класом *ObjectsAsMethodsExample*, щоб можна було проілюструвати його.

Визначте порожній клас *Demo*. Виконайте «*Demo new answer42*» і зверніть увагу, що виникає звичайна помилка *Message Not Understood*.

```
Object subclass: #Demo
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'PBE-Reflection'
```

```
Demo new answer42
```

Тепер додайте до словника методів класу *Demo* звичайний об'єкт.

```
Demo methodDict
   at: #answer42
   put: ObjectsAsMethodsExample new
```

Спробуйте ще раз надрукувати результат виконання «*Demo new answer42*». Цього разу ви мали б отримати відповідь 42.

```
Demo new answer42
>>> 42
```

Якщо переглянути клас *ObjectsAsMethodsExample*, то можна знайти наведені нижче методи.

```
ObjectsAsMethodsExample >> run: oldSelector with: arguments in: aReceiver
  ^ self perform: oldSelector withArguments: arguments

ObjectsAsMethodsExample >> answer42
  ^ 42
```

Коли екземпляр *Demo* отримує повідомлення *answer42*, пошук методу відбувається як зазвичай, однак віртуальна машина виявить, що замість скомпільованого методу цю роль намагається відіграти звичайний об'єкт Pharo. Тоді віртуальна машина надішле йому нове повідомлення *run:with:in:* з аргументами – початковим селектором методу, його аргументами та отримувачем. Оскільки *ObjectsAsMethodsExample* реалізує такий метод, то він перехоплює повідомлення та делегує його собі.

Видалити підробку можна як зі звичайного словника.

```
Demo methodDict removeKey: #answer42 ifAbsent: []
```

Якщо уважніше придивитися до *ObjectsAsMethodsExample*, то виявиться, що його надклас також реалізує деякі методи: *flushcache*, *methodClass:* і *selector:*, але всі вони порожні. Скомпільованому методу можуть надсилати такі повідомлення, тому об'єкт, який прикидається скомпільованим методом, мусить бути реалізованим. *flushcache* є найважливішим методом, який потрібно реалізувати; інші можуть знадобитися деяким інструментам залежно від того, чи визначено метод за допомогою *Behavior >> addSelector:withMethod:* чи безпосередньо за допомогою *MethodDictionary >> at:put:*.

А для серйозного використання ідеї, поданої в цьому параграфі, наполегливо рекомендуємо застосовувати бібліотеку, що називається *MethodProxies*, доступну на http://github. com/pharo-contributions/MethodProxies, оскільки вона набагато надійніша і безпечніша.

18.20. Підсумки розділу

Рефлексія означає здатність запитувати, досліджувати та навіть змінювати метаоб'єкти системи виконання як звичайні об'єкти.

- Інспектор використовує *instVarAt:* і пов'язані методи для перегляду приватних змінних екземплярів об'єктів.
- Метод *Behavior >> allInstances* повертає всі екземпляри класу.
- Повідомлення *class*, *isKindOf:*, *respondsTo:* тощо корисні для збору показників або створення інструментів розробки, але їх треба уникати у звичайних програмах: вони порушують інкапсуляцію об'єктів і ускладнюють розуміння та підтримку коду.
- *SystemNavigation* – службовий клас, який містить багато корисних запитів для навігації та перегляду ієрархії класів. Наприклад, використовуйте *SystemNavigation default browseMethodsWithSourceString: 'pharo' matchCase: true*, щоб знайти та переглянути всі методи з заданим вихідним рядком. (Повільно, але ретельно!).
- Кожен клас Pharo вказує на екземпляр *MethodDictionary*, який відображає селектори на екземпляри *CompiledMethod*. Скомпільований метод вказує на свій клас, замикаючи коло.
- *RGMethodDefinition* – легкий проксі-сервер для скомпільованого методу, який надає додаткові зручні методи та використовується багатьма інструментами Pharo.
- *RBBrowserEnvironment*, частина інфраструктури Refactoring Browser, пропонує досконаліший ніж *SystemNavigation* інтерфейс для запитів до системи. Його перевага в тому, що результат запиту можна використовувати як область нового запиту. Доступні графічний і програмний інтерфейси.
- *thisContext* – псевдозмінна, яка втілює стек віртуальної машини на етапі виконання. Здебільшого використовується налагоджувачем для динамічної побудови інтерактивного перегляду стеку. Вона також особливо корисна для динамічного визначення відправника повідомлення.
- Контекстні точки переривання можна встановити за допомогою *haltIf:*, якщо вказати аргументом селектор певного методу. *haltIf:* перерве виконання лише тоді, коли названий метод буде у стеку викликів.
- Поширеним способом перехоплення повідомлень, надісланих певній сутності, є використання *мінімального об'єкта* як обгортки для неї. Такий проксі-сервер реалізує якомога менше методів і перехоплює всі надіслані повідомлення, реалізуючи *doesNotunderstand:*. Після перехоплення він може виконати деякі додаткові дії, а потім переслати початкове повідомлення за призначенням.

- Надішліть *become:*, щоб поміняти місцями посилання на два об'єкти, наприклад, на проксі та його суб'єкт.
- Будьте обережні, деякі повідомлення, як-от *class* і *yourself*, насправді ніколи не надсилаються, а інтерпретуються віртуальною машиною. Інші, як-от +, – та *ifTrue:* можуть бути безпосередньо інтерпретовані або вбудовані віртуальною машиною залежно від отримувача.
- Ще одне типове використання перевизначення *doesNotUnderstand:* – це ліниве завантаження або компіляція методів, яких бракує. Метод *doesNotUnderstand:* не може перехопити надсилання *self*.