Розділ 6

Створення невеликої гри

У цьому розділі ми створимо просту гру Lights Out (https://en.wikipedia.org/wiki/Lights_Out_(game)). У процесі роботи ми розширимо наше знайомство з Оглядачем класів, Інспектором об'єктів, Налагоджувачем і системою контролю версій Iceberg. Важливо добре оволодіти цими основними інструментами. У Pharo можна програмувати у звичній манері: визначити клас, потім його поля та методи. Однак у Pharo процес розробки може бути більш продуктивним! Ви можете визначати змінні екземпляра та методи на льоту, писати код у Налагоджувачі, використовуючи точний контекст об'єктів, що існують на поточний момент. Тому ми знову заохочуватимемо вас використовувати розробку, керовану тестуванням, під час написання цієї гри.

Кілька слів попередження: цей розділ містить кілька навмисних помилок, зроблених для того, щоб продемонструвати, як обробляти помилки, що трапилися під час виконання, та знаходити їх в коді. Вибачте, якщо це вас трохи розчарує, але мусимо побачити ці важливі технічні прийоми в дії, тому намагайтеся дотримуватися наших інструкцій.

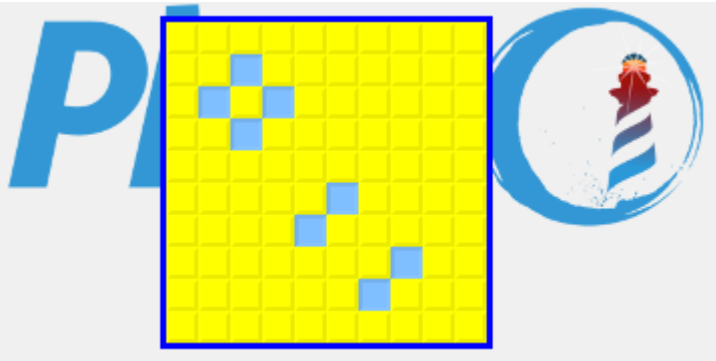

Рис. 6.1. Поле гри Lights Out

6.1. Гра Lights Out

Ігрове поле – це прямокутник, заповнений жовтими клітинками. Натискання на одну з них перемикає колір чотирьох сусідніх, і вони стають синіми (рис. 6.1). Повторне натискання на клітинку знову змінить колір сусідів: вони стануть жовтими. Мета гри – отримати якомога більше синіх клітинок.

«Lights Out» має два типи об'єктів: ігрове поле та 100 окремих клітинок. Програмний код, що реалізує гру, міститиме два класи: один для гри, інший – для клітинок.

6.2. Створення нового пакета класів

Нам потрібно створити новий пакет. Як і раніше, зробимо це в Оглядачі. Якщо не пригадуєте, як створюють пакети, перегляньте ще раз розділ 3 «Швидкий огляд Pharo» та розділ 5 «Розробка простого лічильника».

Ми назвемо пакет *PBE-LightsOut*. Так ви зможете швидко знайти його серед усіх інших: просто надрукуйте «*PBE*» в рядку фільтра, і бачитимете тільки його.

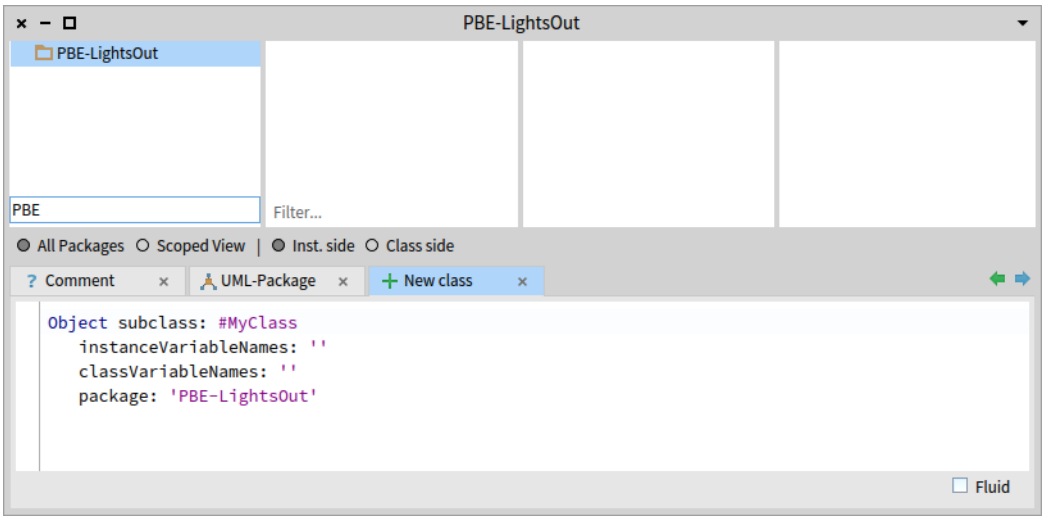

Рис. 6.2. Відфільтровування пакетів для ефективнішої роботи

6.3. Визначення класу *LOCell*

Зараз у новому пакеті, звісно, немає класів. Проте в нижній панелі Оглядача, панелі редагування, відображається шаблон для створення нового класу (див. рис. 6.2). Давайте заповнимо його необхідними даними.

Лістинг 6.1. Визначення класу *LOCell*

```
SimpleSwitchMorph subclass: #LOCell
   instanceVariableNames: 'mouseAction'
   classVariableNames: ''
   package: 'PBE-LightsOut'
```

6.4. Створення нового класу

Лістинг 6.1 містить визначення нового класу, яке ми плануємо використовувати.

Давайте на хвилину замислимося, що ми бачимо у шаблоні визначення класу. Чи це є якась спеціальна форма, яку потрібно заповнити, щоб створити новий клас? Чи це є новий синтаксис? Ні, це просто повідомлення, яке надсилають об'єктові! Визначення класу – це вираз Pharo, який надсилає повідомлення до існуючого класу *SimpleSwitchMorph* з проханням створити підклас *LOCell*. Саме повідомленням є «*subclass:instanceVariableNames:classVariableNames:package:*». Воно дещо багатослівне. Усі аргументи є рядками, крім імені підкласу, який створюємо – його задано символом *#LOCell*. Ім'я пакета Оглядач вказує автоматично – це наш новий пакет, в якому оголошуємо клас (див. рис. 6.2). І '*mouseAction*' задає ім'я змінної екземпляра, яку ми використаємо, щоб вказати, яка дія відбудеться, коли хтось клацне на клітинці.

То чому ми наслідуємо від *SimpleSwitchMorph*, а не від *Object*? Дуже скоро ми побачимо переваги наслідування спеціалізованих класів і довідаємося, для чого потрібна змінна екземпляра *mouseAction*.

Щоб повідомлення надійшло до класу, підтвердьте його зміни командою контекстного меню або комбінацією [*Cmd* + *S*]. Повідомлення буде надіслано, клас відкомпільовано, і ми отримаємо щось таке, як на рис. 6.3.

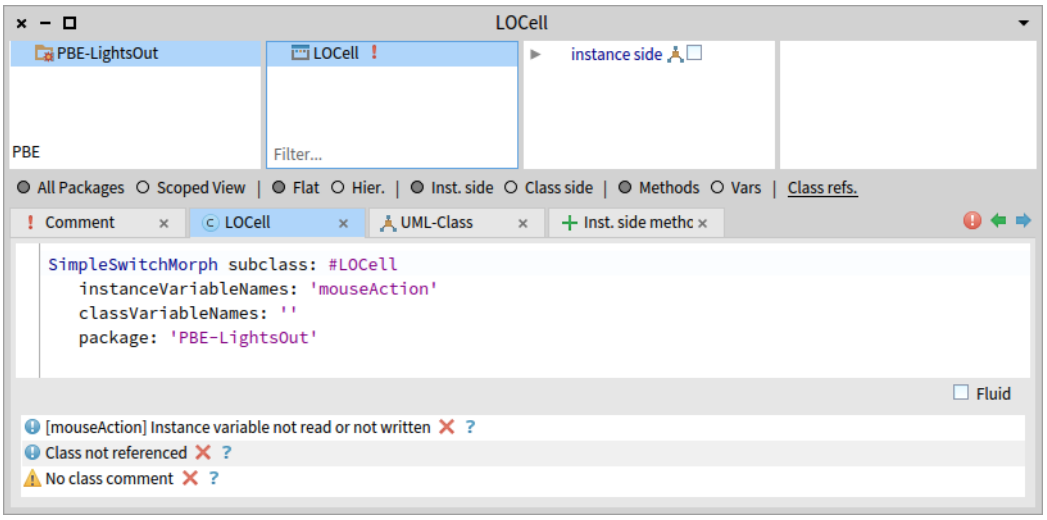

Рис. 6.3. Новостворений клас *LOCell*

Новий клас з'явився в панелі класів Оглядача, а панель редагування тепер показує визначення класу. Унизу вікна видно відгук Помічника з якості: він автоматично запускає перевірку правил якості на вашому коді та звітує про результат. Можете не звертати на нього увагу: трохи забагато для початку.

6.5. Про коментарі

Pharo-розробники високо цінують не тільки читабельність їхнього коду, а також і хороші якісні коментарі.

Коментарі до методів

Людям властиво вірити, що добре написані методи не обов'язково коментувати. Ніби призначення та зміст методу мають бути очевидними після прочитання. Це хибна думка, яка заохочує неохайність. Звісно, погано написаний код, крім коментування, потрібно виправити та перебудувати. Хороший коментар не виправдовує складний для читання код.

Очевидно, що немає змісту коментувати тривіальні методи. Коментар не мав би бути перекладом коду людською мовою, натомість мав би пояснювати, що цей метод робить, контекст його виконання або резон створення. Коментар має розвіяти можливі сумніви читача щодо призначення коду та сприяти його правильному розумінню.

Коментарі до класів

Ви вже знайомі з вкладкою коментарів Оглядача класів. Перейдіть на неї, і побачите шаблон якісного коментаря, наданий розробниками Pharo. Прочитайте його! Зразок побудовано згідно з CRC-дизайном[1], який розробили Кент Бек і Уорд Канінгем, коли працювали над Smalltalk у 80-х роках минулого століття (перегляньте їхню статтю[2],

[1] Class Responsibility Collaborators – «Клас Відповідальність Взаємодія», метод аналізу при проектуванні об'єктно-орієнтованого програмного забезпечення
[2] Kent Beck and Ward Cunningham «A Laboratory For Teaching Object-Oriented Thinking», https://dl.acm.org/doi/pdf/10.1145/74878.74879

щоб довідатися більше). Коротко кажучи, коментар кількома реченнями описує *відповідальність* класу, позаяк він *взаємодіє* з іншими класами, щоб реалізувати цю відповідальність. Додатково можна зазначити інтерфейс класу (основні повідомлення, які розуміє екземпляр класу), навести приклад використання (зазвичай у Pharo визначають приклади як методи класу) та деякі деталі внутрішнього влаштування класу чи обґрунтування реалізації.

6.6. Додавання методів до класу

Давайте додамо кілька методів до нашого класу. Найперше додамо метод екземпляра з лістингу 6.2.

Лістинг 6.2. Ініціалізація екземпляра класу *LOCell*

```
LOCell >> initialize
  super initialize.
  self label: ''.
  self borderWidth: 2.
  bounds := 0 @ 0 corner: 16 @ 16.
  offColor := Color yellow.
  onColor := Color r:0 g:0.4 b:1.
  self useSquareCorners.
  self turnOff
```

Нагадаємо, що ми використовуємо запис «*ClassName* >> *methodName*» лише для того, щоб вказати, в якому класі визначено метод.

Зауважте, що символи '' у третьому рядку – це дві окремі одинарні лапки без розділювача між ними, а не одна подвійна лапка! Так позначають порожній рядок. Інший спосіб створити порожній рядок – *String new*. Не забудьте зберегти визначення методу.

У цьому методі багато чого відбувається, давайте розберемося з усім.

Методи ініціалізації

Зауважимо, що цей метод *initialize* відрізняється від того, який ми бачили у лічильника в попередньому розділі. Нагадаємо, що це спеціальний метод, і за домовленістю його буде викликано одразу після створення об'єкта. Отже, якщо ми виконаємо *LOCell new*, то повідомлення *initialize* буде автоматично відправлене до новоствореного об'єкта. Методи ініціалізації використовують для налаштування стану об'єктів зазвичай, щоб встановити їхні поля – це саме те, що ми тут зробили.

Виклик ініціалізації надкласу

Перше, що робить цей метод (у другому рядку коду) – викликає метод *initialize* свого надкласу *SimpleSwitchMorph*. Можете бути певні, що будь-який успадкований стан буде правильно ініціалізовано методом *initialize* надкласу. Підклас наслідує набір полів надкласу, тому завжди варто надсилати *super initialize* та ініціалізувати успадкований стан, перш ніж виконувати будь-які інші дії. Ми не знаємо точно, що зробить метод *initialize* класу *SimpleSwitchMorph* (і можемо про це не турбуватися), але з великою імовірністю він встановить певні доцільні початкові значення успадкованим полям. Тому нам краще викликати його, щоб не зіткнутись з якимось невідомим станом морфи.

Далі метод налаштовує стан екземпляра. Наприклад, надсилаючи *self label: ''*, він робить порожній рядок написом цього об'єкта.

Про створення точки та прямокутника

Вираз «*0 @ 0 corner: 16 @ 16*», здається, варто пояснити докладніше. *0 @ 0* створює об’єкт класу *Point* з обома координатами, *x* та *y*, які дорівнюють нулю. Точніше, *0 @ 0* надсилає повідомлення @ числу *0* з аргументом *0*. У результаті число *0* просить клас *Point* створити новий екземпляр з координатами (0; 0). Тепер ми відправляємо цій новоствореній точці повідомлення *corner: 16 @ 16*, що змушує її створити екземпляр *Rectangle* з вершинами (0; 0) та (16; 16). Цей новостворений прямокутник буде присвоєно змінній *bounds*, успадкованій від надкласу. Змінна *bounds* визначає, якого розміру буде наша морфа. По суті, ми сказали: «Будь квадратом 16 на 16 пікселів».

Зауважимо, що початком системи координат вікна Pharo є лівий верхній кут, координата *x* зростає зліва направо, координата *y* – згори донизу.

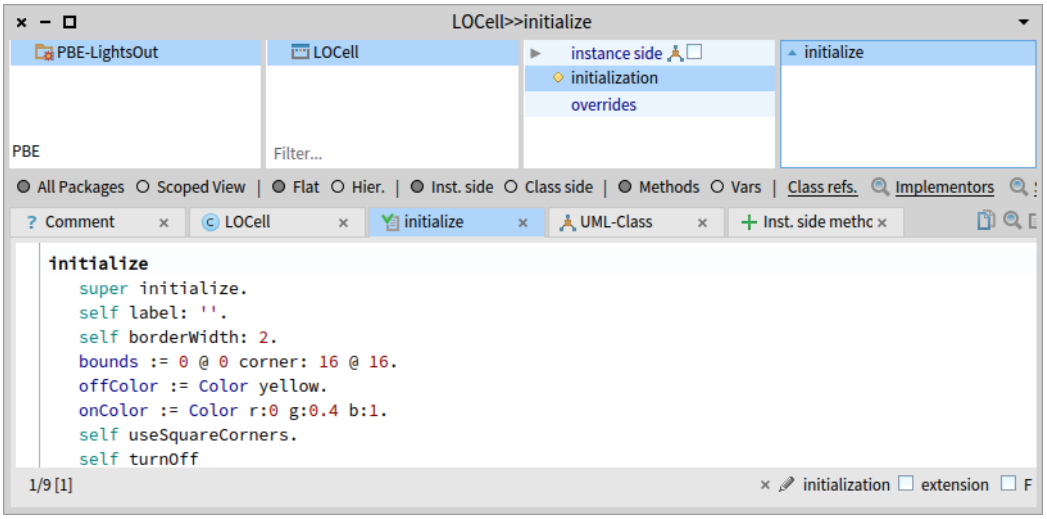

Рис. 6.4. Новостворений метод *initialize*

Про решту

Решта методу мала б «говорити» сама за себе. Частиною мистецтва написання хорошого коду Pharo є вибір хороших імен методів, щоб код можна було читати як ламану англійську. Ви могли б уявити об’єкт, який розмовляє сам з собою і каже: «Наказую собі: використовуй квадратні кути!», «Наказую собі: вимкнися!».

Зверніть увагу на маленьку синю стрілку біля імені методу (див. рис. 6.4). Вона означає, що *initialize* визначений у надкласі і перевизначений у вашому класі. Ще одна дія відбулася без нашої участі: середовище автоматично зачислило наш метод до протоколу *initialization*.

6.7. Інспектування об’єкта

Ви можете невідкладно випробувати дію написаного коду, створивши новий об’єкт класу *LOCell* та проінспектувавши його. Відкрийте Робоче вікно або Пісочницю, введіть вираз «*LOCell new*» і оберіть «*Inspect it*» з контекстного меню (чи натисніть [*Cmd* + *I*]).

Лівий стовпець вкладки *Raw* інспектора показує список полів, а правий – їхні значення (див. рис. 6.5). Інші вкладки демонструють інші сторони *LOCell*. Ми переглянемо їх та поекспериментуємо.

Якщо ви клацнете на одному з полів, то Інспектор відкриє нову панель з деталями обраного поля (див. рис. 6.6). Закрити її можна кнопкою з хрестиком (вгорі праворуч).

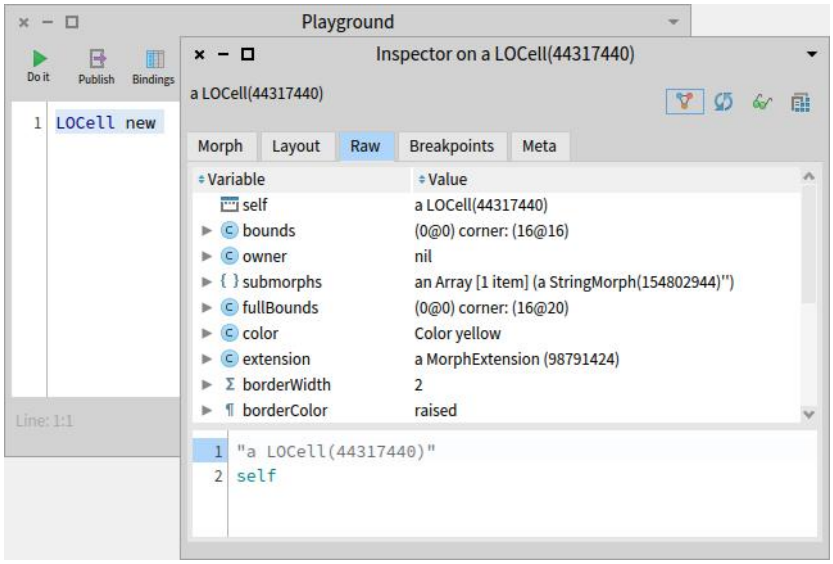

Рис. 6.5. Дослідження екземпляра *LOCell* за допомогою Інспектора

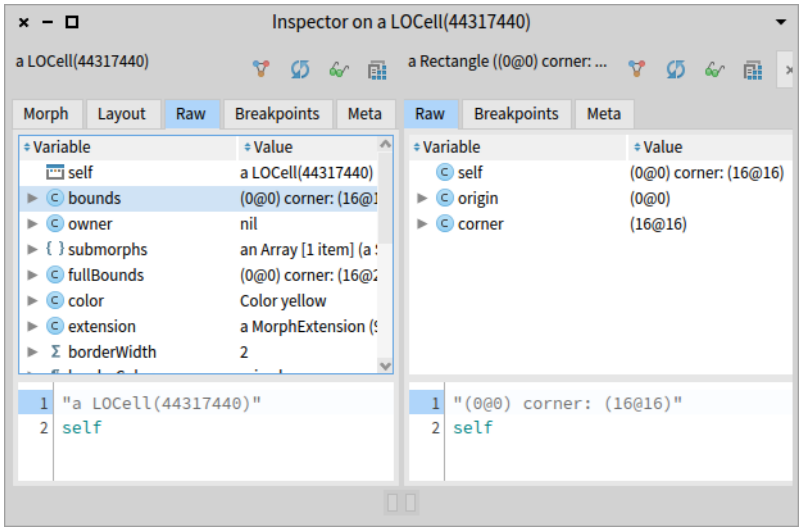

Рис. 6.6. Щоб проінспектувати значення змінної екземпляра (інший об'єкт), достатньо на ній клацнути

Виконання виразів

Нижня панель інспектора працює як маленьке робоче вікно. Вона корисна тим, що у ній псевдозмінна *self* прив'язана до вибраного об'єкта.

Перейдіть до цієї панелі внизу вікна Інспектора, наберіть такий текст: «*self bounds: (100@50 corner: 150@100)*» та виконайте «*Do it*». Значення змінної *bounds* має змінитися в Інспекторі автоматично. Якщо цього не відбулося, то натисніть на кнопку *Refresh* (пара синіх закручених стрілок) у правому верхньому куті вікна, щоб примусово оновити відображення в Інспекторі. Тепер наберіть текст «*self openInWorld*» у робочій панелі та виконайте його.

Меню-ореол

У лівому верхньому куті екрана має з'явитись клітинка, як показано на рис. 6.7. Вона з'явилась саме у тій позиції і того розміру, що вказані у змінній *bounds*: нижче від верхнього краю вікна на 50 пікселів та правіше від лівого на 100, ширина та висота клітинки – по 50 пікселів.

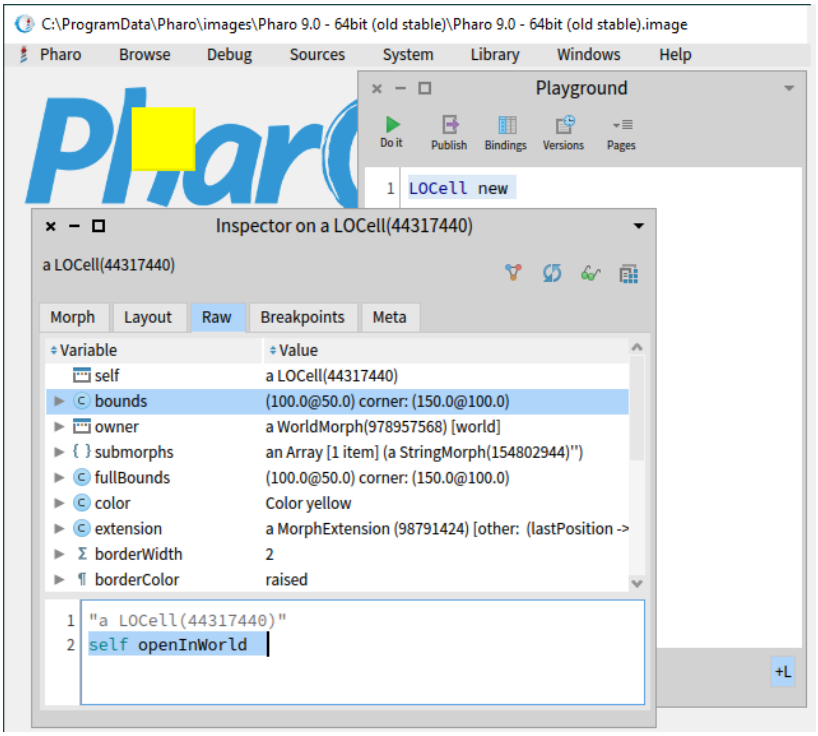

Рис. 6.7. Екземпляр *LOCell* відкрито у *World*

Метаклацніть на клітинці, щоб відкрити її меню-ореол. Це меню надає візуальний спосіб взаємодії з екземпляром класу *Morph* за допомогою маніпуляторів, що зараз оточують морф. Перетягніть клітинку по екрану чорним маніпулятором (вгорі посередині), змініть розмір клітинки жовтим маніпулятором (внизу праворуч). Простежте, як змінюються значення *bounds* в Інспекторі. (Можливо, вам доведеться натиснути кнопку *Refresh*). Зверніть увагу на те, що щойно створений морф уже має поведінку: клацніть на ньому, і колір клітинки зміниться! Закрийте клітинку, клацнувши на рожевому маніпуляторі з хрестиком.

6.8. Визначення класу *LOGame*

Давайте створимо ще один потрібний для гри клас. Назвемо його *LOGame*.

Зробіть видимим шаблон створення класу у вікні редагування коду Оглядача класів. Для цього клацніть на назві пакета (або виберіть команду «*Add Class*» з контекстного меню панелі класів). Відредагуйте код, щоб він виглядав, як показано в лістингу 6.3, і збережіть його.

Лістинг 6.3. Визначення класу *LOGame*

```
BorderedMorph subclass: #LOGame
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'PBE-LightsOut'
```

Тут ми наслідуємо клас *BorderedMorph*. Клас *Morph* є базовим для всіх графічних фігур у Pharo. Ми вже бачили *SimpleSwitchMorph*, такий *Morph*, який можна увімкнути та вимкнути. Закономірно, що *BorderedMorph* – це *Morph*, який має межу. Ми також могли б вставити імена змінних екземпляра між лапками в другому рядку, але, поки що, залишимо цей список порожнім.

Лістинг 6.4. Метод ініціалізації гри

```
LOGame >> initialize
  | sampleCell width height n |
  super initialize.
  n := self cellsPerSide.
  sampleCell := LOCell new.
  width := sampleCell width.
  height := sampleCell height.
  self bounds: (5 @ 5 extent: (width * n) @ (height * n) + (2 * self
        borderWidth)).
  cells := Array2D new: n tabulate: [ :i :j | self newCellAt: i at: j ]
```

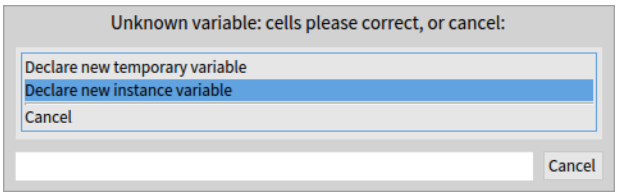

Рис. 6.8. Визначення нової змінної екземпляра «на льоту»

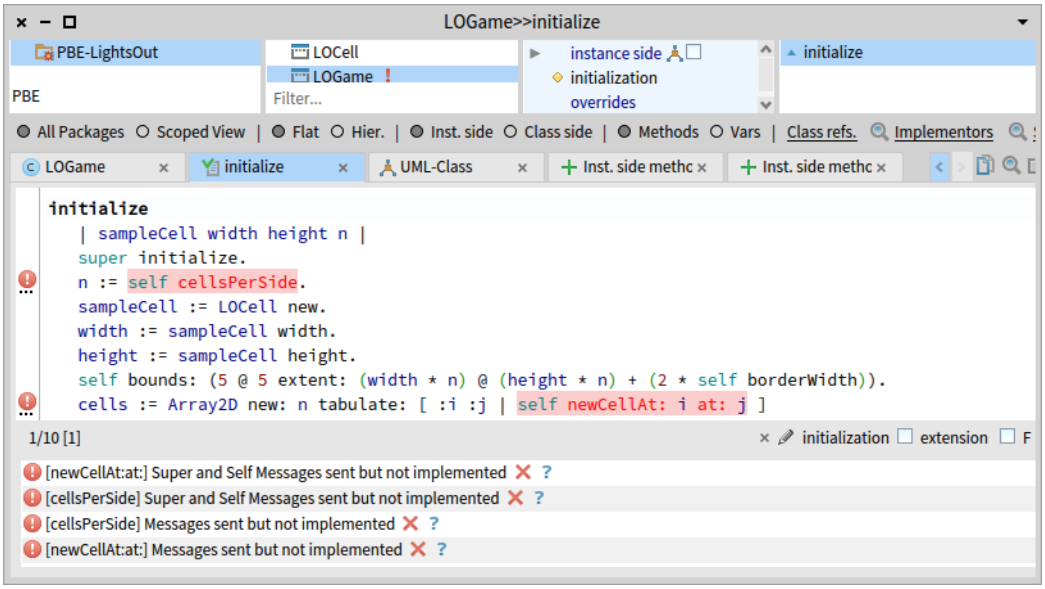

Рис. 6.9. Помилки, знайдені в тексті методу ініціалізації

6.9. Ініціалізація гри

Тепер визначимо метод *initialize* для *LOGame*. Введіть код з лістингу 6.4 в Оглядачі як метод екземпляра *LOGame*, і спробуйте зберегти його. Це досить об'ємний метод, проте не переживайте, ми детально пояснимо кожен його рядок у наступних параграфах.

Pharo поскаржиться, що він не знає, що означає *cells* (див. рис. 6.8), і запропонує вам кілька способів як виправити ситуацію. Оберіть «*Declare new instance variable*», оскільки нам потрібне поле даних екземпляра.

Відкрийте вкладку «*LOGame*» Оглядача і переконайтеся, що в оголошенні класу як за помахом чарівної палички з'явилося оголошення змінної *cells*. Поверніться до вкладки з оголошенням методу *initialize*, і ви побачите, що компілятор виявив ще дві проблеми в тексті методу (рис. 6.9). Наступний параграф розповість, яким незвичним способом можна їх залагодити: ми спробуємо *виконати* проблемний код!

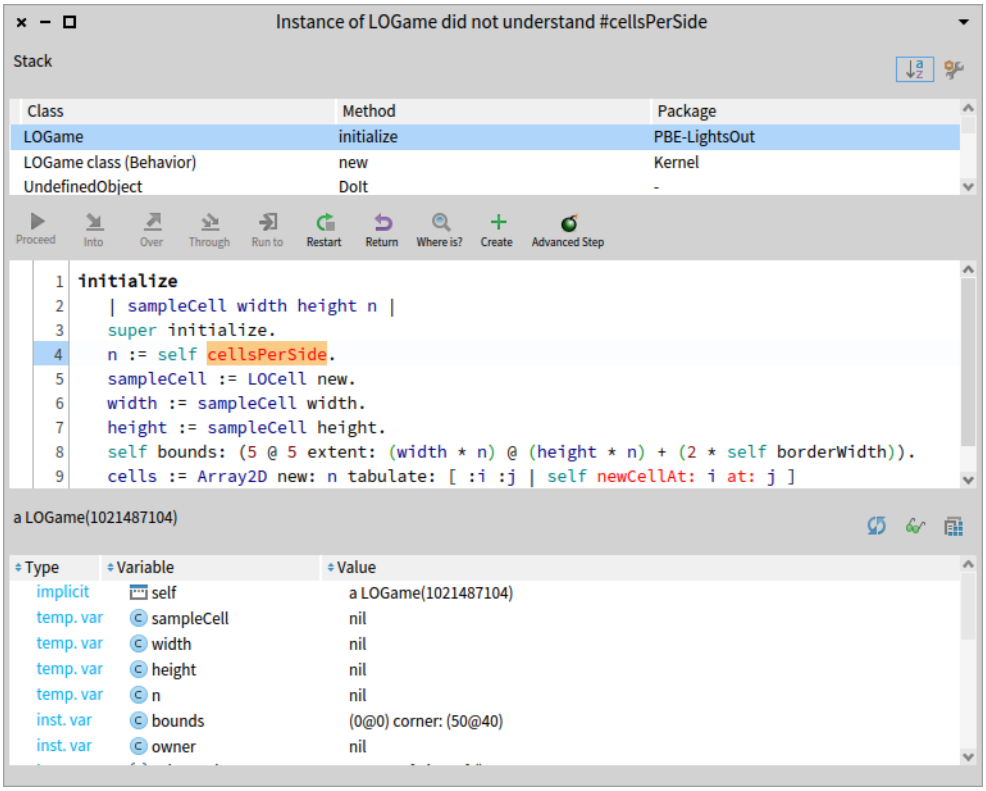

Рис. 6.10. Pharo знайшов невідомий селектор

6.10. Використання переваг Налагоджувача

На цьому етапі, якщо ви відкриєте Робоче вікно, введете фрагмент «*LOGame new*» і виконаєте його, Pharo поскаржиться, що йому невідомий зміст деяких виразів (див. рис. 6.10). Він відкриє Налагоджувач і повідомить, що екземпляр *LOGame* не розуміє повідомлення *cellsPerSize*. Проте *cellsPerSize* не є помилкою: просто це метод, який ще не визначили. Зараз ми це зробимо.

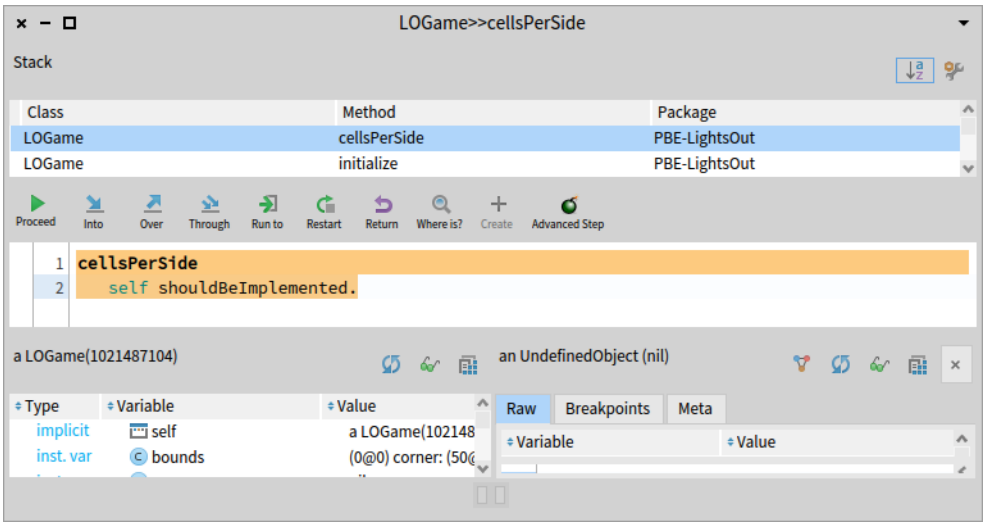

Рис. 6.11. Система створила новий метод, тіло якого потрібно визначити

Не закривайте Налагоджувач, а натисніть на його кнопку ***Create***. У відповідь на запит про клас, який міститиме метод, оберіть *LOGame*. У запиті про протокол методу введіть «*accessing*» і натисніть ***Ok***. Налагоджувач створить метод *cellsPerSize* на льоту й одразу запустить його на виконання. Створена так стандартна реалізація методу помістила в

його тіло повідомлення «*self shouldBeImplemented*». Його виконання запускає виняток і ... знову відкриває Налагоджувач на визначенні методу (рис. 6.11).

Тепер ви можете написати визначення нового методу. Цей метод важко зробити простішим: він завжди повертає константу 10.

```
LOGame >> cellsPerSide
  "The number of cells along each side of the game"
  ^ 10
```

Однією з переваг подання констант за допомогою методів є те, що у випадку розвитку програми, коли значення такої константи стає залежним від інших властивостей, метод можна легко змінити для обчислення цього значення.

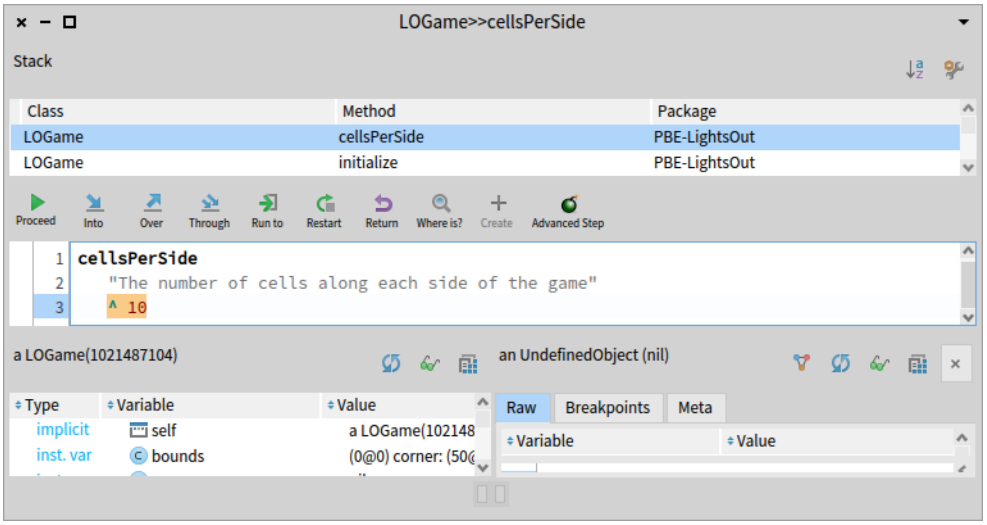

Рис. 6.12. Визначення методу *cellsPerSide* в Налагоджувачі

Не забудьте скомпілювати метод, як тільки напишете його, за допомогою все тієї ж команди «*Accept*». Ви мали б отримати ситуацію, зображену на рис. 6.12. Якщо ви натиснете кнопку ***Proceed***, то програма продовжить своє виконання і зупиниться, бо ми не визначили метод *newCellAt:at:*.

Можемо визначити його, як і попередній, але поки що зупинимось, щоб пояснити детальніше, що вже зробили. Закрийте Налагоджувач і погляньте на визначення класу ще раз (натиснувши на *LOGame* на панелі класів Системного оглядача). Ви побачите, що Оглядач доповнив визначення класу так, що воно тепер містить змінну екземпляра *cells*. Клацніть на методі *initialize* – тут також відбулися певні зміни. Колір повідомлення *cellsPerSide* змінився з червоного на звичайний чорний, і поменшало повідомлень про помилки від Помічника з якості. Червоним залишилося лише повідомлення *newCellAt:at:*.

6.11. Вивчаємо метод *initialize*

Давайте розберемо метод *initialize*. Для зручності пояснення ми перенумерували його рядки.

Рядок 2

У рядку 2 вираз «| *sampleCell width height n* |» оголошує чотири тимчасові змінні. Їх називають тимчасовими, тому що їхня область видимості та тривалість життя

обмежені цим методом. Тимчасовим змінним дають пояснювальні імена, щоб зробити код легшим для сприйняття. Рядки 4–7 задають значення цих змінних.

```
1 initialize
2    | sampleCell width height n |
3    super initialize.
4    n := self cellsPerSide.
5    sampleCell := LOCell new.
6    width := sampleCell width.
7    height := sampleCell height.
8    self bounds: (50 @ 50 extent:
                      (width * n)@(height * n) + (2 * self borderWidth)).
9    cells := Array2D
        new: n
        tabulate: [ :i :j | self newCellAt: i at: j ]
```

Рядок 4

Якого розміру мало б бути поле для гри? Достатнього, щоб вмістити деяку невід'ємну кількість клітинок та намалювати межу навколо них. Скільки клітинок має бути? 5? 10? 100? Наразі ми не знаємо, яке число вибрати, але, якби й знали, то ймовірно могли б змінити свій вибір пізніше. Тому ми делегуємо відповідальність за знання цієї кількості на інший метод, який називаємо *cellsPerSide* (його ще не існувало на момент написання методу ініціалізації). Не лякайтесь цього, насправді це є хорошою практикою писати код, звертаючись до ще не визначених методів. Чому? Бо поки ми не почали писати метод *initialize*, ми й не здогадувались, що нам знадобиться *cellsPerSide*. І в цей момент ми можемо дати йому змістовне ім'я та продовжувати, без переривання процесу. Як ми вже бачили, такий метод легко визначити під час випробування об'єкта, а можливість відкласти реалізацію на потім є суперсилою Pharo.

Отже, рядок 4 відправляє повідомлення *cellsPerSide* до *self*, тобто самому собі. Відповідь, кількість клітинок на одній стороні ігрового поля присвоюється змінній *n*.

Наступні три рядки створюють новий екземпляр *LOCell*, присвоюють його ширину і висоту відповідним тимчасовим змінним. Навіщо все це? Розміри клітинки потрібні, щоб обчислити розміри поля, а в кого ж їх довідатися, як не в самої клітинки? Найкраще з можливих місць зберігання розмірів клітинки – екземпляр класу *LOCell*.

Рядок 8

Рядок 8 задає межі нового об'єкта гри. Не вникаючи у деталі, повірте, що вираз у дужках створює квадрат з лівим верхнім кутом у точці (50; 50) і правим нижнім кутом достатньо далеко, щоб вмістити потрібну кількість клітинок.

Останній рядок

Останній рядок присвоює полю *cells* новостворену матрицю, екземпляр класу *Array2D*, з правильною кількістю рядків і стовпців. Ми створили матрицю, надіславши повідомлення «*new:tabulate:*» до класу *Array2D*. Класи також є об'єктами, тому можемо надсилати їм повідомлення. Ми знаємо, що «*new:tabulate:*» приймає два аргументи, бо він має дві двокрапки (:) в імені. Аргументи вказують зразу після двокрапок. Якщо ви звикли до мов, що передають аргументи всі разом всередині дужок, то це спочатку може видатись дивним. Але не панікуйте, це лише синтаксис. Виявляється, що це дуже навіть хороший синтаксис, оскільки назву методу можна використовувати для пояснення

призначення аргументів. Наприклад, цілком зрозуміло, що «*Array2D rows: 5 columns: 2*» має 5 рядків і 2 стовпці, а не 2 рядки та 5 стовпців.

«*Array2D new: n tabulate: [:i :j | self newCellAt: i at: j]*» створює новий двовимірний масив (матрицю) *n*×*n* та ініціалізує її елементи. Початкове значення кожного елемента залежить від його координат. Елемент на позиції (*i*, *j*) буде ініціалізовано результатом обчислення «*self newCellAt: i at: j*».

6.12. Поділ методів на протоколи

Перш ніж визначати інші методи, глянемо на третю панель угорі Оглядача. Так само, як перша панель Оглядача дає змогу нам класифікувати класи на пакети, панель протоколів допомагає класифікувати методи, щоб не доводилось працювати з дуже довгим списком імен на панелі методів. Такі групи методів називають «протоколами».

За замовчуванням ви матимете віртуальний протокол «*instance side*», який містить усі методи класу.

Якщо ви виконували приклад створення гри, то панель протоколів вашого Оглядача мала б містити протоколи «*initialization*» та «*overrides*». Вони додаються автоматично, коли ви перевизначаєте метод «*initialize*». Системний оглядач автоматично організовує методи та додає їх до відповідного протоколу, коли тільки можливо.

Звідки оглядач знає, як вибрати правильний протокол? Загалом він не знає напевне, але може робити певні припущення. Наприклад, якщо метод *initialize* визначено в надкласі, то він припустить, що наш метод *initialize* потрібно зачислити до того ж протоколу, що й метод, який він перевизначає.

Панель протоколів може також містити протокол «*as yet unclassified*», до якого належать усі некласифіковані методи. Тоді, щоб виправити ситуацію, оберіть команду «*categorize all uncategorized*» з контекстного меню панелі протоколів, або класифікуйте кожен метод вручну.

6.13. Завершення розробки гри

А зараз давайте визначимо інші методи, використані в *LOGame>>initialize*. Ви можете використовувати для цього і Оглядач класів, і Налагоджувач. У будь-якому випадку почнемо з *LOGame>>newCellAt:at:* у протоколі *initialization*.

```
LOGAme >> newCellAt: i at: j

  "Create a cell for position (i,j) and add it to my on-screen
  representation at the appropriate screen position.
  Answer the new cell"

    | c origin |
    c := LOCell new.
    origin := self innerBounds origin.
    self addMorph: c.
    c position: ((i - 1) * c width) @ ((j - 1) * c height) + origin.
    c mouseAction: [ self toggleNeighboursOfCellAt: i at: j ]
```

Зауваження. Наведений код спричинить виняток, оскільки містить навмисну помилку.

Форматування

Як ви можете бачити, у коді є порожні рядки та відступи. Для того, щоб дотримуватись однакових правил форматування коду, можете покластися на допомогу Pharo: виберіть команду «*Format code*» з контекстного меню панелі редагування методу, або використайте [*Cmd* + *Shift* + *F*]. Це автоматично відформатує текст вашого методу.

Від перекладача. Команду «*Format code*» відшукати не так просто: вона розташована останньою в підменю розділу «*Source code*» контекстного меню панелі редагування методу. Цей розділ – перший розділ меню, але з'являється тільки після того, як збережено текст методу. Отже, для швидкого форматування використовуйте гарячі клавіші.

Перемикання сусідів

Визначений вище метод створює нову *LOCell* та задає її екранне розташування залежно від індексів (*i*, *j*) у матриці клітинок. Останній рядок робить значенням поля *mouseAction* нової клітинки блок *[self toggleNeightboursOfCellAt: i at: j]*. У ньому визначено поведінку обробника події клацання мишкою на клітинці. Відповідний метод також потрібно визначити.

Лістинг 6.5. Метод опосередкованого виклику

```
LOGAme >> toggleNeighboursOfCellAt: i at: j
  i > 1
        ifTrue: [ (cells at: i - 1 at: j) toggleState ].
  i < self cellsPerSide
        ifTrue: [ (cells at: i + 1 at: j) toggleState ].
  j > 1
        ifTrue: [ (cells at: i at: j - 1) toggleState ].
  j < self cellsPerSide
        ifTrue: [ (cells at: i at: j + 1) toggleState ]
```

Метод *toggleNeighboursOfCellAt:at:* перемикає стан чотирьох сусідніх клітинок угорі, внизу, ліворуч і праворуч від клітинки (*i*, *j*). Єдине ускладнення полягає в тому, що ігрове поле обмежене, тому перш ніж перемикати стан клітинки, потрібно переконатись, що вона існує.

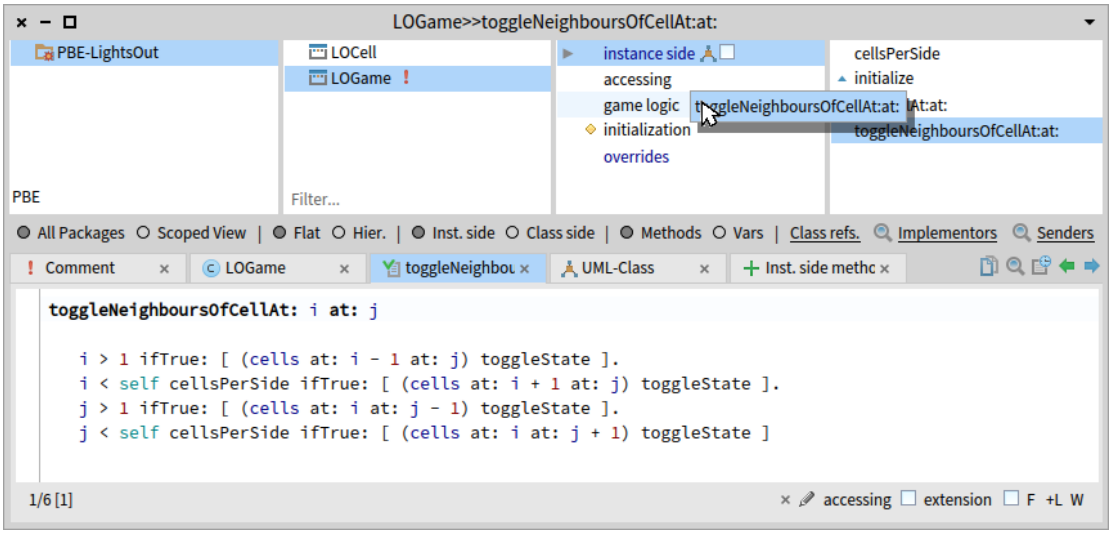

Рис. 6.13 Перетягування методу в протокол

Помістіть цей метод у новий протокол, що називається «*game logic*». Щоб додати новий протокол, використайте відповідну команду контекстного меню панелі протоколів. Тепер, щоб перемістити метод до нового протоколу, просто перетягніть його мишкою (див. рис. 6.13).

Лістинг 6.6. Типовий метод-модифікатор

```
LOCell >> mouseAction: aBlock
  mouseAction := aBlock
```

Лістинг 6.7. Обробник події

```
LOCell >> mouseUp: anEvent
  mouseAction value
```

6.14. Останні методи класу *LOCell*

Для завершення розробки гри Lights Out нам потрібно визначити ще два методи у класі *LOCell*. Цього разу для обробки подій мишки.

Перший з них, зображений у лістингу 6.6, є звичайним методом доступу. Він не робить нічого особливого, лише присвоює змінній *mouseAction* клітинки свій аргумент. Ми мали б оголосити його в протоколі «*accessing*».

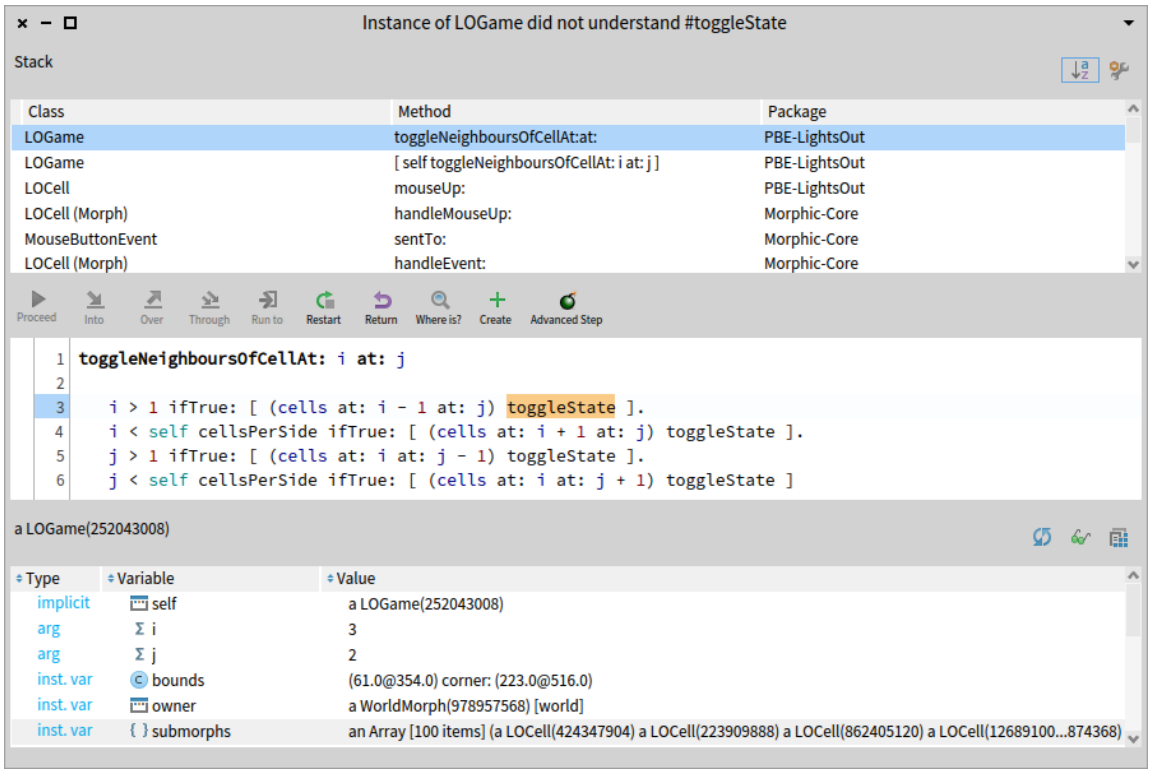

Рис. 6.14. Налагоджувач підсвічує метод, який спричинив помилку

Нам залишилось визначити метод *mouseUp:*. Його автоматично викликатиме графічний інтерфейс користувача в момент відпускання кнопки мишки, коли курсор є над певною клітинкою. Перегляньте лістинг 6.7: метод надсилає повідомлення *value* об'єктові, що зберігається у змінній екземпляра *mouseAction*. В методі *LOGame >> newCellAt:at:* ми створили блок *[self toggleNeighboursOfCellAt: i at: j]*, виконання якого перемикає всіх сусідів клітинки, та присвоїли його змінній *mouseAction* клітинки. Отже, надсилаючи

повідомлення *value* цьому блокові, ми примушуємо його виконатися та, як наслідок, перемкнути стан сусідніх клітинок.

6.15. Випробуємо код

Це все: створення гри Lights Out завершене! Якщо ви виконали усі кроки, то зможете зіграти у гру, що складається лише з двох класів і семи методів. У Робочому вікні або Пісочниці наберіть «*LOGame new openInHand*» та виконайте (командою «*Do it*»).

Гра відкриється причеплена до курсору: ви зможете перемістити її в довільне зручне місце на екрані та зафіксувати там клацанням. Далі ви мали б клацати по клітинках і бачити, як усе працює. Принаймні так в теорії... Але коли ви клацнете по клітинці, з'явиться налагоджувач. У верхній частині його вікна ви побачите стек виконання, що показує усі активні методи. Виберіть будь-який з них, і в середній панелі з'явиться код, що виконується у цьому методі. Частина, що спричинила помилку, буде підсвічена.

Натисніть на рядку з написом «*LOGame toggleNeighboursOfCellAt:at: PBE-LightsOut*» (вгорі списку). Налагоджувач покаже контекст виконання методу, де трапилася помилка (див. рис. 6.14).

Унизу Налагоджувача є ділянка змінних, що належать до контексту виконання. Ви можете інспектувати об'єкт – отримувач повідомлення, яке спричинило виконання методу, позначеного в Налагоджувачі. То ж ви можете переглянути тут значення змінних екземпляра-отримувача. Також можете побачити значення аргументів методу так само, як і проміжні значення, отримані під час виконання.

За допомогою Налагоджувача можете покроково виконувати код, інспектувати параметри методів і локальні змінні, виконувати фрагменти коду так само, як у Робочому вікні, і, що найбільш несподівано для користувачів інших налагоджувачів, змінювати сам код під час його налагодження! Деякі Pharo-програмісти майже постійно працюють у Налагоджувачі – більше, ніж в Оглядачі класів. Перевагою такого підходу є те, що ви бачите, як буде виконано метод, який пишете, зі справжніми параметрами та в актуальному контексті виконання.

У нашому випадку заголовок вікна Налагоджувача (див. рис. 6.14) інформує, що повідомлення *toggleState* надійшло до екземпляра *LOGame*, хоча очевидно, що то мав би бути екземпляр *LOCell*. Найбільш ймовірно, що проблема є з ініціалізацією матриці клітинок. Переглянувши код *LOGame >> initialize*, бачимо, що *cells* заповнено значеннями, які повертає метод *newCellAt:at:*, але, якщо уважніше розглянути його текст, то побачимо, що там немає виразу повернення! За замовчуванням метод у Pharo повертає *self* – отримувача, який у випадку *newCellAt:at:* справді є екземпляром *LOGame*. Як ми уже говорили, для повернення значення з методу у Pharo використовують вираз «^ *значення*».

Лістинг 6.8. Виправлення помилки

```
LOGAme >> newCellAt: i at: j

  "Create a cell for position (i,j) and add it to my on-screen
  representation at the appropriate screen position.
  Answer the new cell"

    | c origin |
    c := LOCell new.
```

```
        origin := self innerBounds origin.
        self addMorph: c.
        c position: ((i - 1) * c width) @ ((j - 1) * c height) + origin.
        c mouseAction: [ self toggleNeighboursOfCellAt: i at: j ].
        ^ c
```

Лістинг 6.9. Перевизачення події переміщення мишки

```
LOCell >> mouseMove: event
    "Do nothing"
```

Отже, виправте знайдену помилку. Закрийте вікно Налагоджувача. Додайте вираз «^ *c*» у кінці методу *LOGame>>newCellAt:at:*, щоб він повертав *c*. Не забудьте закінчити попередній рядок крапкою (див. лістинг 6.8).

У багатьох випадках ви можете виправити код безпосередньо в налагоджувачі і, натиснувши ***Proceed***, продовжити виконання програми. У багатьох, але не в нашому. Через те, що помилка була в ініціалізації об'єкта, а не в методі, найпростіше буде закрити вікно Налагоджувача, зупинити запущений екземпляр гри і створити новий. Щоправда, відкрити нашу гру простіше, ніж закрити. Вона не має кнопок керування, тому скористаємося меню-ореолом. Чи вдалося вам метаклацнути на межі вікна гри, що має товщину один піксель? Якщо ні, то закрийте спочатку морфу будь-якої клітинки (через меню-ореол), а тоді – морфу поля гри.

Виконайте «*LOGame new openInHand*» знову, бо, якщо використати старий екземпляр гри, то працюватиме блок зі старою логікою.

Зараз гра має працювати правильно... або майже правильно. Якщо ми порухаємо мишку у проміжок часу між натисканням кнопки та відпусканням, тоді клітинка, над якою перебувала мишка, також змінить свій стан. Виявляється, це поведінка, успадкована від *SimpleSwitchMorph*. Ми можемо виправити помилку, просто перевизначивши метод *mouseMove*: так, щоб він нічого не робив (див. лістинг 6.9). Зауважимо також, що методи опрацювання подій мишки варто зачислити до протоколу «*event handling*».

Нарешті ми закінчили!

6.16. Домовленості щодо найменування методів доступу

Якщо ви використовували методи-селектори та методи-модифікатори в інших мовах програмування, то можете очікувати, що методи доступу до змінної екземпляра *mouseAction* називатимуться *getMouseAction* та *setMouseAction*. Але у Pharo діє інша домовленість про іменування. Селектор завжди має таке саме ім'я, як і змінна, яку він повертає, а модифікатор має таке ж ім'я, але з двокрапкою наприкінці: *mouseAction* та *mouseAction:*, відповідно. Селектори і модифікатори разом називають *методами доступу* (accessor methods) і за домовленістю їх зачисляють до протоколу *accessing*. У Pharo всі змінні об'єкта є приватними для нього, тому інші об'єкти можуть читати чи записувати ці змінні виключно через методи доступу, як ті, що описані вище. Звичайно, об'єкт має вільний доступ до всіх успадкованих полів, але ви ніколи не зможете доступитися до змінних іншого об'єкта, навіть, якщо він є екземпляром вашого класу або самим класом.

6.17. Про налагоджувач

Коли трапляється помилка у Pharo, система за замовчуванням відображає налагоджувач. Проте ми можемо повністю контролювати цю поведінку. Наприклад, можемо записати помилку у файл. Можемо навіть серіалізувати стек виконання до файлу, архівувати і відкрити його в іншому образі системи. Під час розробки програм Налагоджувач дає нам змогу рухатись так швидко, як це можливо. Проте у готових системах розробники часто налаштовують налагоджувач так, щоб їхні помилки не заважали занадто сильно роботі їхніх користувачів.

Якщо все пішло не так

Передусім не нервуйте! Цілком нормально творити безлад під час написання програм. Якщо що, то це правило, а не виняток. Напевно, найнеприємніше, що може трапитися, коли ви починаєте експериментувати з графічними елементами в Pharo, – це те, що екран захаращують незнищенні, здавалося б, морфи. Не панікуйте. Спробуйте отримати інспектора на одній з них: метаклацніть на ній, щоб відкрити меню-ореол, виберіть бузковий маніпулятор з гайковим ключем на ньому – це «*debug*», оберіть з його меню команду «*inspect morph*». Як тільки ви відкриєте Інспектора, ви перемогли! Щоб закрити гру, наберіть у панелі редактора в Інспекторі та виконайте:

- якщо ви інспектуєте екземпляр гри, то «*self delete*»;
- якщо ви інспектуєте клітинку гри, то «*self owner delete*».

Від перекладача. Метаклацання – це ще те завдання! Використання ореолу різноколірних маніпуляторів ні на що не схоже. Хотілося б мати якийсь звичний і надійний спосіб контролю запуску та завершення гри, наприклад, за допомогою меню. Наступний параграф описує, як це зробити. Удосконалення гри стане хорошою нагодою продемонструвати нові корисні можливості Pharo.

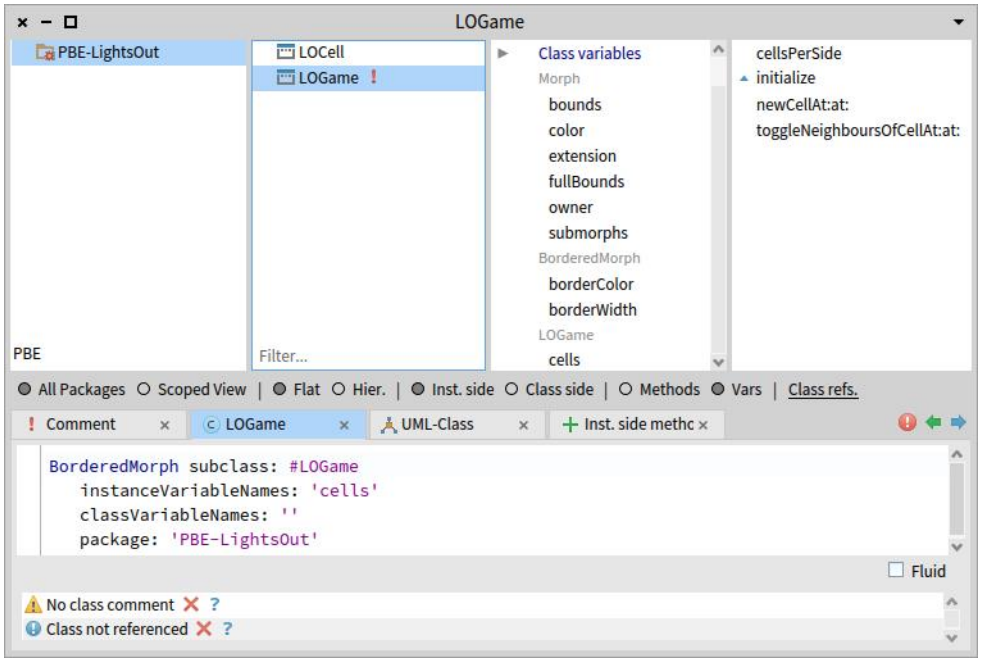

Рис. 6.15. Перелік змінних екземпляра класу *LOGame*

6.18. Удосконалення гри[3]

Ми щойно закінчили, а вже виникли думки щодо удосконалення нашого продукту. Чи дуже зручно вам закривати гру маніпулятором меню-ореола? Нам теж ні. Ніяк не вдається поцілити вказівником мишки в межу поля гри… Здається, варто зробити її ширшою. Але широка чорна рамка виглядатиме сумно, то ж змінимо також і її колір. Щоб досягти бажаного, доведеться провести невелике дослідження ієрархії класів, з якої наслідує клас *LOGame*.

Зміна методу ініціалізації

Значення товщини межі та її кольору мали б десь зберігатися. Давайте з'ясуємо, які поля даних успадкував *LOGame*. Відкрийте Оглядач класів на *LOGame* і оберіть ***Vars*** на перемикачі ***Methods/Vars*** Оглядача (нижче від панелі протоколів). Панель протоколів відобразить перелік усіх змінних екземпляра, структурований за класами, в яких ці поля оголошені, як на рис. 6.15. Бачимо, що надклас *LOGame* містить саме ті поля, які нам потрібні: *borderColor* і *borderWidth*.

Відразу виникає спокуса задати цим змінним бажані значення в методі *LOGame >> initialize*. Наприклад, так:

```
initialize
  | sampleCell width height n |
  super initialize.
  borderColor := Color blue.
  borderWidth := 3.
  n := self cellsPerSide.
  . . .
```

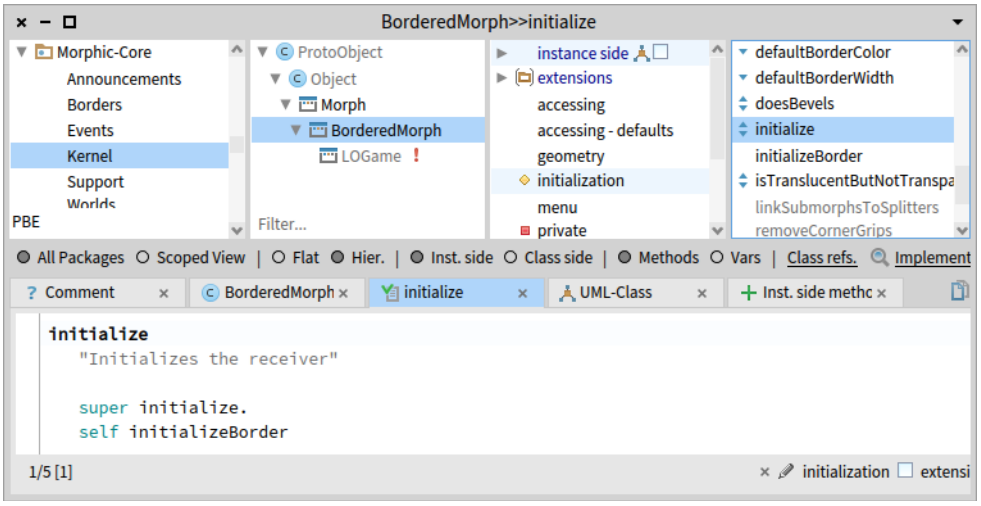

Рис. 6.16. Ієрархія наслідування класу *LOGame*

Якщо зберегти зроблені зміни і запустити гру, то побачимо, що все працює, гра набула саме такого вигляду, як на рис. 6.1. Але не все так добре, як може видатися на перший погляд. Адже поля *borderColor* і *borderWidth* оголошені в класі *BorderedMorph*, і саме він має задавати їхні початкові значення. Схоже, ми запрограмували повторну ініціалізацію змінних *borderColor* і *borderWidth*. Це двічі неправильно: виконується зайва робота, і

[3] Цей параграф написав перекладач книги.

нові значення, задані в підкласі, можуть спричинити несподівані помилки. Давайте продовжимо дослідження коду і придумаємо кращий спосіб, щоб досягти бажаного.

Значення полів *borderColor* і *borderWidth* мав би задавати метод *BorderedMorph >> initialize*. Знайдемо його в Оглядачі класів. Нижче від панелі класів Оглядача є перемикач ***Flat/Hier.***, що приховує або вмикає відображення в панелі ієрархії наслідування класів незалежно від їхньої належності до пакетів. Оберіть ***Hier.***, послідовно клацніть на класі *BorderedMorph*, протоколі *initialization*, методі *initialize*. Ви мали б перейти до методу ініціалізації надкласу, як зображено на рис. 6.16.

З тексту методу легко здогадатися, що змінними *borderColor* і *borderWidth* займається метод *BorderedMorph >> initializeBorder*. А вже в його тілі знайдемо:

```
BorderedMorph >> initializeBorder
      "Initialize the receiver state related to border."

      borderColor:= self defaultBorderColor.
      borderWidth := self defaultBorderWidth
```

Тепер все зрозуміло: значення кольору і товщини межі задають відповідні методи класу *BorderedMorph*. Якщо ми хочемо задавати інші значення в нашому підкласі, то маємо перевизначити методи *defaultBorderColor* і *defaultBorderWidth*. Нагадаємо, що в мові Pharo псевдозмінна *self* вказує на отримувача повідомлення (виконавця методу) і слугує інструментом, за допомогою якого клас може передавати повідомлення своїм підкласам, навіть, ще не написаним.

Отож додамо до протоколу *initialization* класу *LOGame* два нові методи (а присвоєння змінним в методі *LOGame >> initialize* вилучимо).

```
LOGame >> defaultBorderColor
      "answer the default border color for the game board"
      ^ Color blue

LOGame >> defaultBorderWidth
      "answer the default border width for the game board"
      ^ 3
```

Тепер гра працює так, як треба, а ми зайвий раз переконалися, що задавати константи за допомогою методів – це правильний підхід. Такі значення легко модифікувати в підкласах, не порушуючи код інших методів.

Зміна способу запуску

Усі складові частини Pharo можна запустити програмно, або командою меню, або, навіть, комбінацією клавіш. А нашу гру ми запускаємо тільки програмно (з Пісочниці). Давайте виправимо цю несправедливість і додамо відповідну команду до якогось меню! Наприклад, до *World Menu*. Ми ще не вміємо цього робити, але Pharo – відкрита система, тому спробуємо.

Відкрийте Навідника і в рядку пошуку наберіть «*world menu*». Перше ж посилання серед результатів пошуку – *Breakpoint class>>#debugWorldMenuOn:* – показує, як класи системи додають команди до меню. Наприкінці списку результатів є посилання на розділ «World Menu Items» довідкової системи – це саме те, що потрібно. Сам Навідник довідки не відкриє, але нам не буде важко знайти його через команду «*Help > Help browser*» головного меню. У розділі написано, що для того, щоб додати новий пункт до головного

меню, потрібно визначити *метод класу*, який називається *menuCommandOn*:. Довідка надає також приклади оголошення такого методу в деякому вигаданому класі. Не зайвим буде також пошукати в Pharo справжні класи, які реалізують метод *menuCommandOn*:, і подивитися, як воно зроблено.

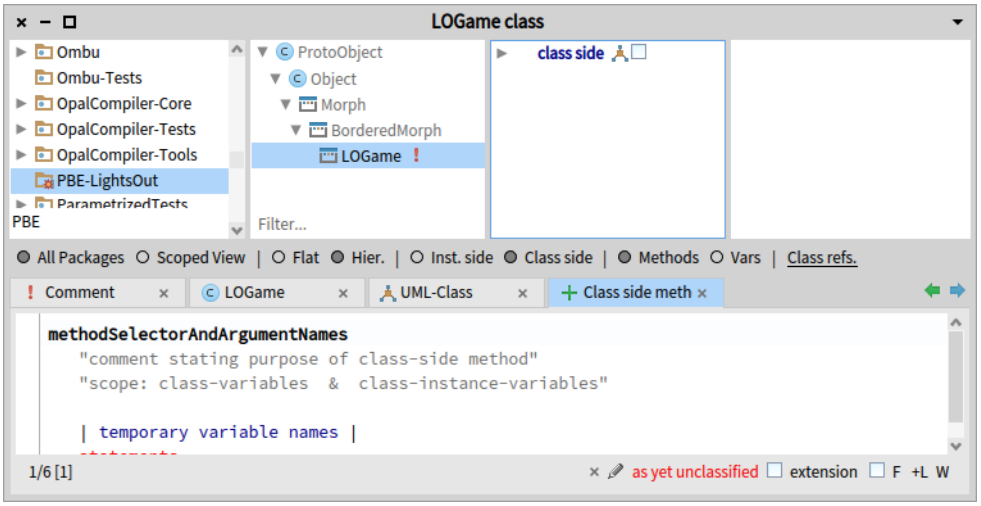

Рис. 6.17. Визначення методу класу

Давайте поміркуємо, чого ми хочемо від команди меню. Вона мала б виконувати той код, який ми набирали в Пісочниці, і запускати гру Lights Out. Але добре було б мати також команду для закривання гри. Займати два рядки головного меню – завелика розкіш, тому об'єднаємо команди відкривання та закривання в одне підменю.

Клацніть в Оглядачі класів на імені *LOGame* і оберіть ***Class side*** в перемикачі ***Inst. side/Class side*** (див. рис. 6.17). Оглядач відобразить перелік методів класу *LOGame* (він поки що порожній). Класи Pharo – також об'єкти, які можуть мати власні методи. Нам потрібно оголосити метод, зазначений в лістингу 6.10.

Лістинг 6.10. Додавання команди до головного меню

```
LOGame class >> menuCommandOn: aBuilder
      <worldMenu>
      (aBuilder item: #LightsOut)
         order: 5.0;
         withSeparatorAfter;
         with: [ (aBuilder item: #Run)
               order: 0;
               action: [ self open ].
            (aBuilder item: #Quit)
               order: 1;
               action: [ self close ] ]
```

Тут прагма *<worldMenu>* вказує на специфіку методу (про прагми ми поговоримо згодом), повідомлення *item:* створює пункт меню з назвою «*LightsOut*», а всі інші повідомлення надходять до цього пункту меню. Зверніть увагу на те, що рядки коду завершуються крапкою з комою. Так у Pharo задають каскад повідомлень, які надходять до того самого отримувача (до пункту меню в нашому випадку). Повідомлення *order:* задає розташування в меню, *withSeparatorAfter* відокремлює пункт від інших частин меню, а *with:* задає структуру вкладеного меню. Пункти підменю описують подібно, тільки замість *with:* використано *action: aBlock* для того, щоб задати реакцію меню на вибір команди.

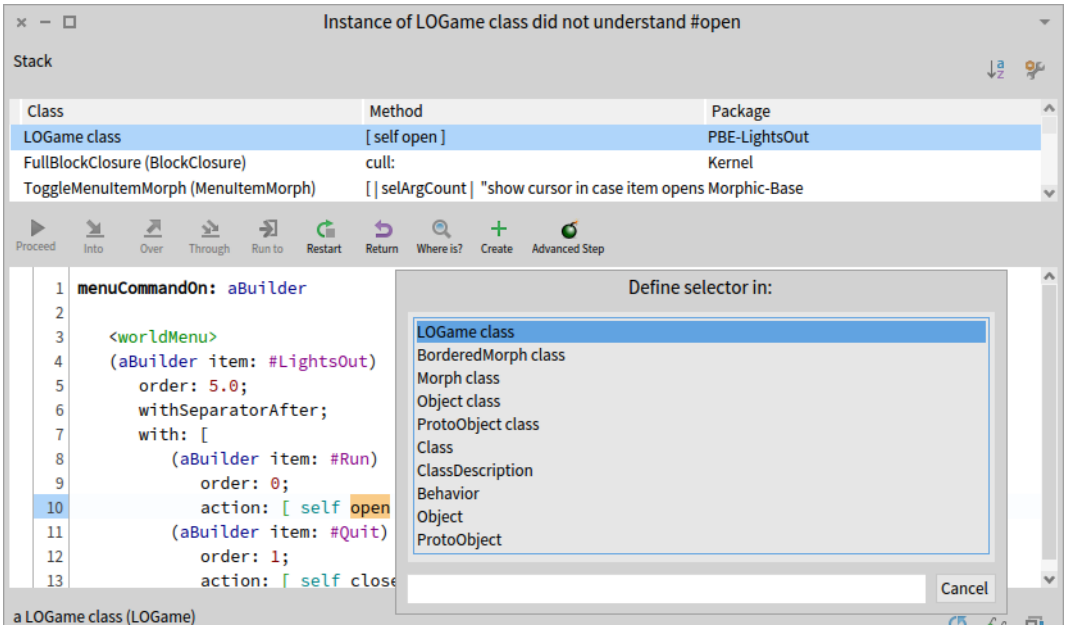

Рис. 6.18. Створення методу класу в Налагоджувачі

Зрозуміло, що команда меню «*Run*» має викликати *LOGame class >> open*, а команда «*Quit*» – *LOGame class >> close*. Ми ще не визначили таких методів класу, але можемо зробити це в Налагоджувачі, як визначали методи екземпляра раніше. То ж викличемо головне меню системи і виберемо «*LightsOut* > *Run*»! Прогнозовано нас «привітає» вікно Налагоджувача повідомленням про помилку. Ми знаємо, де вона трапилася, тому можемо відразу натиснути кнопку «*Create*», вказати клас «*LOGame class*» (див. рис. 6.18), протокол «*instance creation*» і ввести текст методу.

Знайомий нам код «*LOGame new openInHand*» створює безіменний об'єкт, з яким можна взаємодіяти хіба що на екрані за допомогою мишки. Як же його знайде команда закривання? Здається, потрібно зберігати посилання на екземпляр гри, щоб можна було надсилати йому повідомлення програмно. Використаємо для цього змінну класу (у класів Pharo є свої змінні!), яку назвемо «*TheGame*».

У вікні редагування коду Налагоджувача введіть текст методу з лістингу 6.11.

Лістинг 6.11. Створення екземпляра гри

```
LOGame class >> open
  TheGame ifNil: [
     TheGame := self new.
     TheGame openInHand ]
```

Лістинг 6.12. Знищення екземпляра гри

```
LOGame class >> close
  TheGame ifNotNil: [
     TheGame delete.
     TheGame := nil ]
```

Збережіть введений текст, і Налагоджувач перепитає, як оголосити ім'я *TheGame*. Виберіть «*Declare new class variable*». Метод визначено так, що можна буде відкрити лише один екземпляр гри. Посилання на нього зберігатиме змінна класу. До речі, знайдіть у Оглядачі оголошення класу *LOGame* (потрібно повернутися до ***Inst. side***) Воно мало б мати вигляд:

```
BorderedMorph subclass: #LOGame
   instanceVariableNames: 'cells'
   classVariableNames: 'TheGame'
   package: 'PBE-LightsOut'
```

Налагоджувач додав до оголошення у третьому рядку ім'я *TheGame* змінної класу.

Натисніть на кнопку ***Proceed*** Налагоджувача, і він закриється, натомість з'явиться вікно гри! Пограйтеся хвильку, але нам треба визначити ще один метод. Отож виберіть «*World* > *LightsOut* > *Quit*». Старий знайомий Налагоджувач знову повідомить про відсутній метод і допоможе його створити (див. лістинг 6.12). Метод можна зачислити до протоколу *releasing*.

Знову натисніть на ***Proceed*** – закриється і Налагоджувач, і гра. Тепер команди головного меню працюватимуть як треба. Можете випробувати їхню дію (див. рис. 6.19).

Чи завершено роботу над грою? Можливо, що так. Але ви можете захотіти вести статистику гри: рахувати кількість виконаних ходів, кількість увімкнутих клітинок тощо. Було б добре також відображати цю статистику на екрані, але ми залишимо подальші удосконалення гри читачам як вправу.

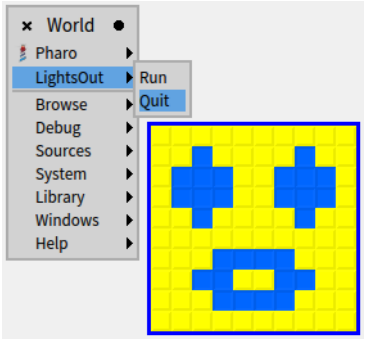

Рис. 6.19 Доповнене головне меню Pharo та гра

Ми успішно доповнили головне меню командою «*LightsOut*», але як тепер повернути його до попереднього вигляду? Через деякий час, награвшись, ви точно захочете зробити це. Виявляється, що достатньо закоментувати прагму <*menuWorld*> у тілі методу *LOGame class* >> *menuCommandOn:*. Не потрібно нічого вилучати, просто приховайте прагму. Захочете знову повернути команду в меню – знімете коментар.

6.19. Зберігання та поширення коду Pharo

Тепер, коли гра Lights Out працює, ви напевно захочете якось її зберегти, щоб мати файл гри та змогу ділитися ним з друзями. Звісно, ви можете зберегти весь образ Pharo і показувати вашу першу програму, запускаючи його. Але ваші друзі, напевно, мають власний код у образах своїх систем і не захочуть втрачати його, використавши ваш образ. Для зберігання коду і відстежування версій уся спільнота Pharo використовує Git і Iceberg – потужний інструмент у складі Pharo, який приховує від користувача низькорівневі деталі.

Використовуйте Iceberg і Git для контролю версій свого коду

Під час розробки лічильника в попередньому розділі ми вже пояснювали вам основи використання Iceberg та Git для зберігання, поширення та контролю версій ваших проєктів. Ви можете використати їх знову для створеної гри. Якщо вам дуже не

терпиться дізнатися більше про Iceberg, то переходьте одразу до розділу 7. Але якщо ви все ще тут, то подивимося, як експортувати програмний код Pharo до файлів.

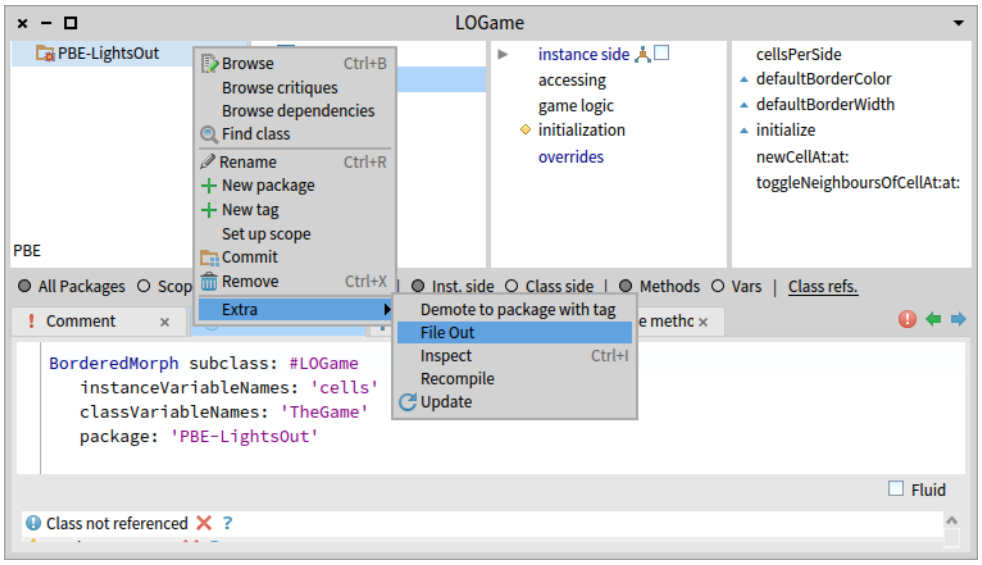

Рис. 6.20. Виведення пакета класів до файлу

Зберігання коду до файлу (додатково)

Найкращий спосіб зберігання коду – використати Git. Найпростіший спосіб отримати текст створених класів – це записати пакет до файлу. Це рудиментарний спосіб, який насправді не відстежує версій, але може виявитися зручним за певних обставин. Спільноти програмістів на Pharo та Squeak називають її «filing out». Позначте в панелі пакетів Системного оглядача *PBE-LightsOut* і виберіть команду «*Extra* > *File Out*» контекстного меню (див. рис. 6.20). Вона виведе весь пакет до файлу «*PBE-LightsOut.st*», створеного в тій папці, де зберігається образ системи. Він більш-менш читабельний, але призначений найперше для комп'ютерів, а не для людей. Ви можете надіслати цей файл друзям і вони зможуть інсталювати його у власний образ Pharo.

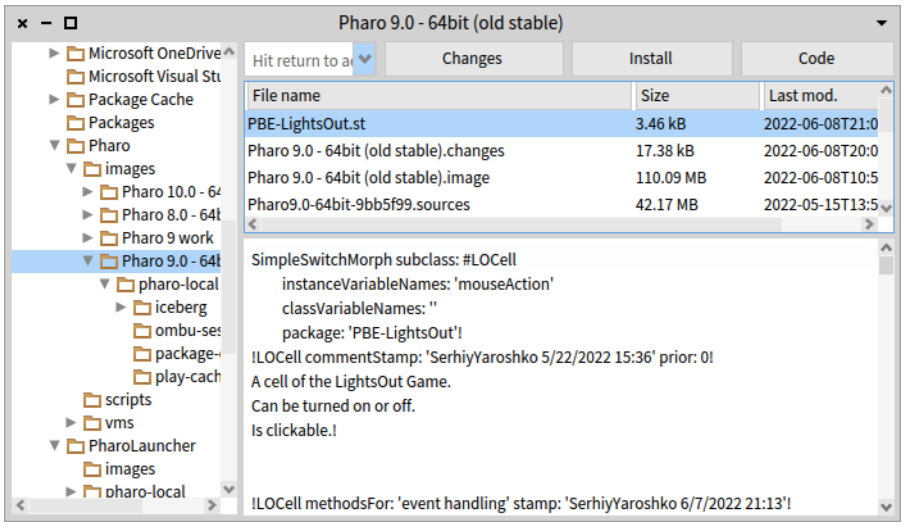

Рис. 6.21. Імпортування коду за допомогою оглядача файлів

Перегляньте отриманий файл за допомогою редактора текстів, щоб зрозуміти, як виглядає текст пакета класів у файлі. Якщо ви хочете переглянути вміст файлу не покидаючи Pharo, то спробуйте надрукувати рядок «*'PBE-LightsOut.st' asFileReference*» в Пісочниці та проінспектувати його.

Відкрийте новий образ Pharo та за допомогою оглядача файлів (*World* > *System* > *File Browser*) знайдіть файл *PBE-LightsOut.st*, відкрийте його контекстне меню та виберіть «*Install into the image*» (див. рис. 6.21). Переконайтеся, що гра працює у новому образі. Надалі використовуйте Git!

Від перекладача. Інспектування «*'PBE-LightsOut.st' asFileReference*» може не спрацювати. Можливо, вам доведеться вказати повне ім'я з маршрутом до нього. За таких умов вміст файлу простіше переглянути в оглядачі файлів Pharo. Подивіться на рис. 6.21: найбільша панель оглядача вже відображає вміст файлу! Перевага інспектора лише в тому, що він виконає синтаксичне підсвічування тексту. До речі, інсталювати файл можна не лише командою контекстного меню, а й кнопкою *Install* оглядача файлів.

6.20. У Pharo ви не можете втратити код[4]

Прикро, але аварія Pharo цілком можлива. Pharo – експериментальна система, яка дає змогу міняти все включно з частинами, які необхідні для функціонування самого Pharo!

Хороші новини в тому, що ви ніколи не загубите вашу роботу, навіть якщо трапиться аварія Pharo, і воно відкотиться до попередньої робочої версії, яка створена, можливо, годину тому назад. Увесь код, виконаний вами в Оглядачі та в Налагоджувачі, зберігається в *.change* файлі, код з Playground зберігається в окремих файлах у папці */pharo-local/playchace* – його можна переглянути звичайним редактором текстів.

Файл змін також можна відкрити звичайним редактором текстів, але зорієнтуватися в ньому буде складно, оскільки розміри файлу іноді вимірюються мегабайтами. Доцільно використати спеціальний оглядач змін, доступний за командою головного меню *World > Sources > Code Changes*. У його лівій панелі ви побачите повний список знімків системи, у правій – перелік подій вибраного знімка, а в нижній – зміст вибраної події. Використовуйте команди контекстного меню для повторного виконання бажаних змін.

6.21. Підсумки розділу

Цей розділ допоміг нам поглибити здобуті раніше знання про Pharo під час розробки простої гри, що використовує бібліотеку графічних елементів (морф). Ми навчилися використовувати Налагоджувач для виправлення помилок і написання коду «на льоту»: так ми оголошували методи екземпляра, методи класу та змінні класу.

Ми відкривали меню-ореол навколо графічного елемента, щоб маніпулювати ним, та, що важливо, щоб позбутися його.

Ми переконалися, що оголошувати константу як результат виконання методу є хорошою практикою, та допомагає легко налаштувати поведінку підкласу.

Невелике дослідження та сміливий експеримент допомогли нам видозмінити недоторкану, здавалося б, частину системи – її головне меню.

Ми також навчилися зберігати пакет класів до текстового файлу та інсталювати його до образу Pharo для швидкого поширення готових програм.

---

[4] Цей параграф перекладач додав з попередньої версії оригіналу.