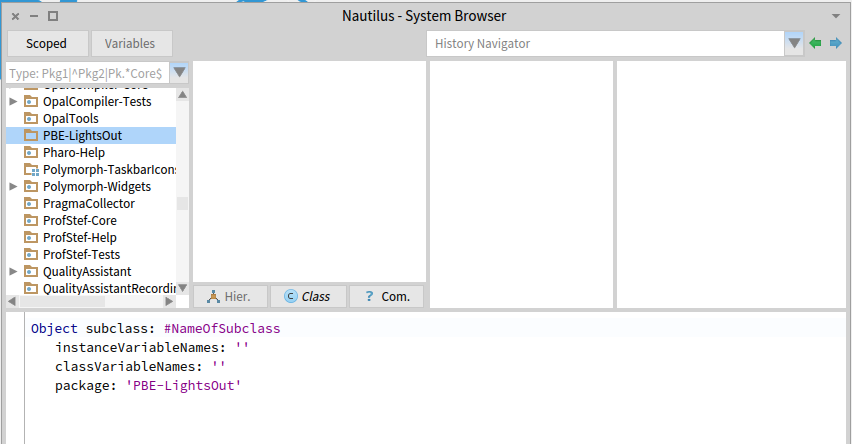
**Перша програма**

У цьому розділі ми створимо просту гру: Lights Out (<https://en.wikipedia.org/wiki/Lights_Out_(game)>). В процесі цього ми покажемо більшість інструментів, якими користуються Pharo програмісти для написання та відлагодження програм, і покажемо як поділитись своєю програмою з іншими розробникам. Ми розглянемо браузер, інспектор об’єктів, відлагоджувач та браузер пакетів Monticello.

У Pharo можна програмувати у звичній манері: визначити клас, потім його поля та його методи. Однак у Pharo процес розробки може бути більш продуктивним! Ви можете визначати змінні та методи на льоту. Також ви можете писати код у відлагоджувачі використовуючи точний контекст щойно виконаних об’єктів. У цьому розділі ми розглянемо альтернативний спосіб та покажемо як збільшити вашу продуктивність.



**Рисунок 3.1** Поле гри Lights Out



**Рисунок 3.2** Створення Пакету та шаблон класу

**3.1 Гра Lights Out**

Щоб показати вам як користуватись програмними інструментами Pharo, ми створимо просту гру під назвою Lights Out. Ігрове поле складається з прямокутника заповненого жовтими клітинками. При натисканні на одну з клітинок, чотири сусідні стають синіми. При повторному натисканні вони стають жовтими знову. Мета гри – отримати якомога більше синіх клітинок.

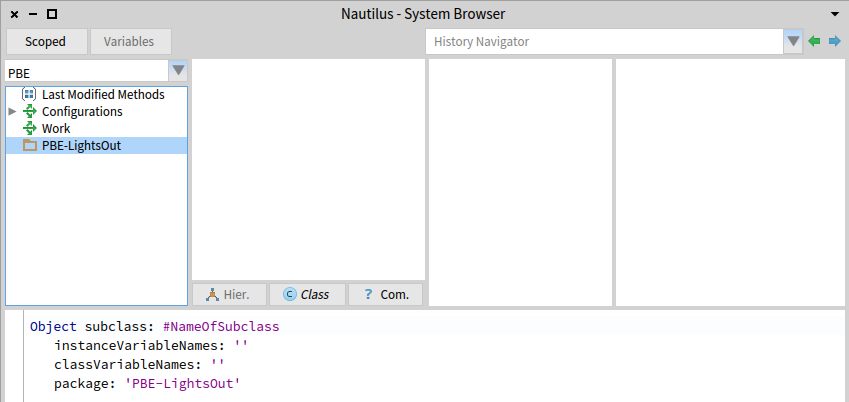
Lights Out має два типи об’єктів: ігрове поле та 100 окремих клітинок. Код, що реалізовує гру міститиме два класи: один для гри та один для клітинок. Зараз ми покажемо, як визначити ці класи з використанням програмних засобів Pharo.

**3.2 Створення нового Пакету**

Ми вже зустрічали браузер у розділі Швидкий Огляд Pharo, де ми навчились переміщатись між пакетами, класами та методами, і побачили, як визначати нові методи. Зараз ми побачимо, як створювати пакети та класи.

З меню World відкриємо System Browser. Правий клік по існуючому пакету на панелі пакетів та оберемо Додати пакет (Add package...) з меню. Введемо назву нового пакету (PBE-LightsOut у прикладах) у діалоговому вікні і натиснемо OK (або клавішу вводу). Новий пакет буде створено й розміщено у алфавітному порядку у списку пакетів (див. Рисунок 3.2).

Підказки: Ви можете написати PBE у фільтрі, щоб отримати ваш пакет відфільтрованим від інших (див. Рисунок 3.3).



**Рисунок 3.3** Фільтрація нашого пакету для більш ефективної роботи

**3.3 Визначення класу LOCell**

Зараз у новому пакеті, звісно, немає класів. Проте, на головній панелі редагування відображається шаблон для створення нового класу (див. Рисунок 3.3)

Цей шаблон показує нам вираз Pharo, що відправляє повідомлення до класу під назвою Object, вказуючи йому створити підклас з назвою NameOfSubClass. Новий клас не матиме змінних та належатиме до категорії (пакету) LightsOut.

**Створення нового класу**

Для створення потрібного нам класу ми просто редагуємо запропонований шаблон. Змініть шаблон створення класу наступним чином:

* Замініть Object на SimpleSwitchMorph.
* Замініть NameOfSubClass на LOCell.
* Добавте mouseAction до списку змінних.

Визначення класу виглядатиме наступним чином:

SimpleSwitchMorph *subclass: #LOCell*

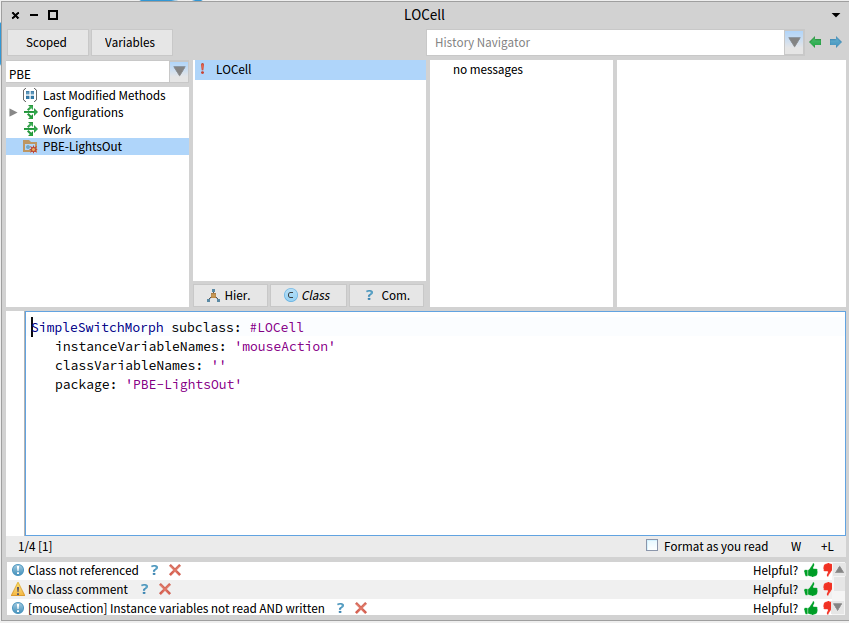
*instanceVariableNames: 'mouseAction'*

*classVariableNames: ''*

*package: 'PBE-LightsOut'*

Нове визначення складається з виразу Pharo, що відправляє повідомлення до існуючого класу SimpleSwitchMorph, вказуючи йому створити підклас з назвою LOCell. (Насправді, оскільки LOCell ще не існує, ми передаємо символ #LOCell, як аргумент, що відповідає за назву класу, який потрібно створити.) Ми також кажемо йому, що екземпляри нового класу повинні мати поле mouseAction, яку ми потім використаємо для визначення дії, яку клітинка повинна виконувати при натисканні мишки не неї.

На даний момент ми досі нічого не створили. Зауважимо, що правий верхній кут панелі змінився на оранжевий. Це означає, що є незбережені зміни. Щоб виконати надсилання повідомлення про створення підкласу, необхідно спочатку зберегти (прийняти) програмний код. Це можна зробити правим кліком і вибором пункту Прийняти (Accept) або використовуючи комбінацію клавіш CMD-s. Повідомлення буде надіслано до SimpleSwitchMorph, що спричинить компіляцію нового класу. Це зображено на Рисунку 3.4.



**Рисунок 3.4** Створення класу

Як тільки визначення класу було прийнято, клас створився і з’явився на панелі класів браузера (див. Рисунок 3.4). Панель редагування тепер показує визначення класу. Знизу доступний відгук Помічника Якості: він автоматично запускає правила якості на вашому коді та звітує їх.

**Про коментарі**

Користувачі Pharo високо цінують не тільки читабельність їхнього коду, а й також хороші якісні коментарі.

**Коментарі до методів.** Людям властиво вірити, що добре написані методи не обов’язково коментувати: цей підхід неправильний та заохочує неохайність. Звісно, поганий код потрібно перейменувати та відрефакторити. Очевидно, що немає змісту коментувати тривіальні методи. Коментар – це не просто код, написаний людською мовою, а пояснення, що цей метод робить, його контекст або логіка схована за реалізацією.

**Коментарі до класів.** Зразок коментування класу Pharo дає хорошу ідею, як має виглядати якісний коментар до класу. Прочитайте його! Ідея полягає у тому, що коментар повинен констатувати відповідальність класу у кількох реченнях та як він взаємодіє з іншими класами, щоб реалізувати ці відповідальності. Додатково можна зазначити API (основні повідомлення, які об’єкт розуміє), навести приклад (зазвичай у Pharo визначають приклади, як методи класу), та деякі деталі про внутрішню реалізацію.

Виберіть кнопку коментування та визначте коментар класу за цим зразком.

**Категорії чи пакети**

Історично склалось, що пакети у Pharo були реалізовані як «категорії» (група класів). З новими версіями Pharo, термін категорії став застарілим та був замінений пакетом.

Якщо ви використовуєте стару версію Pharo чи старий підручник, шаблон класу виглядатиме так:

SimpleSwitchMorph *subclass: #LOCell*

*instanceVariableNames: 'mouseAction'*

*classVariableNames: ''*

*category: 'PBE-LightsOut'*

Це еквівалентно шаблону, який ми згадували раніше. У цій книзі ми використовуватимемо лише термін пакет.

3.4 Додавання методів до класу

Давайте додамо кілька методів до нашого класу. Оберіть протокол ‘без повідомлень’ (‘no messages’) у панелі протоколів. Ви побачите шаблон для створення методу на панелі редагування. Замініть шаблонний текст наступним (не забудьте скомпілювати його):

*initialize*

*super initialize.*

*self label: ''.*

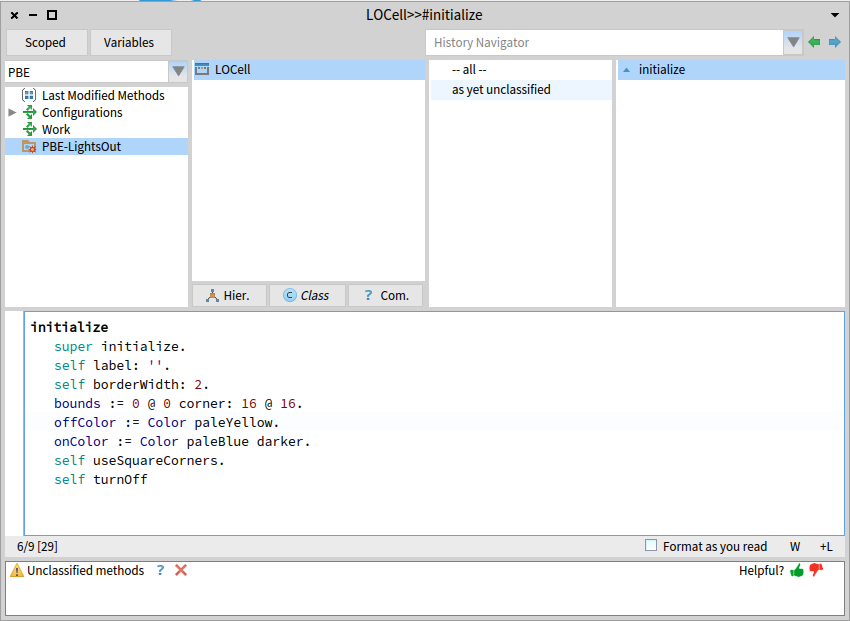
*self borderWidth: 2.*

*bounds := 0 @ 0 corner: 16 @ 16.*

*offColor := Color paleYellow.*

*self useSquareCorners.*

*self turnoff*



**Рисунок 3.5** Щойно створений метод initialize

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

**Методи ініціалізації.** Зауважимо, що метод називається initialize. Це ім’я дуже важливе! За конвенцією, якщо у класі визначено метод з назвою initialize, то його буде викликано зразу після створення об'єкту. Отже, якщо ми виконаємо LOCell new, повідомлення initialize буде автоматично відправлене до новоствореного об’єкта. Методи ініціалізації використовуються для встановлення стану об’єктів, зазвичай, щоб встановити їхні поля; це саме те, що ми зробили тут.

**Виклик ініціалізації суперкласу.** Перше, що цей метод робить (рядок 2) – виконує метод initialize свого суперкласу, SimpleSwitchMorph. Суть у тому, що будь-який успадкований стан буде правильно ініціалізовано методом initialize суперкласу. Завжди слід ініціалізовувати успадкований стан відправляючи super initialize перед тим, як робити інші дії. Ми не знаємо точно, що робить метод initialize класу SimpleSwitchMorph, але з великою ймовірністю він встановить деякі раціональні початкові значення полям. Тому нам краще викликати його, щоб не зіткнутись з нечистим станом. Далі метод встановлює стан цього об’єкту. Відправляючи self label: ‘’, наприклад, встановлює написом цього об’єкта порожній рядок.

**Про створення точки та прямокутника.** Вираз 0@0 corner: 16@16 напевне варто розглянути детальніше. 0@0 представляє об’єкт Point з x та y координатами, обидві рівні 0. Насправді, 0@0 відправляє повідомлення @ числу 0 з аргументом 0. В результаті, число 0 вкаже класу Point створити новий екземпляр з координатами (0, 0). Тепер ми відправляємо цій новоствореній точці повідомлення corner: 16@16, що змушує її створити Rectangle з кутами 0@0 та 16@16. Цей новостворений прямокутник буде присвоєно змінній bounds, успадкованій від суперкласу.

Зауважимо, що початком екрану в Pharo є верхній лівий кут і координата y збільшується вниз.

**Решта.** Решта методу має говорити сама за себе. Частиною мистецтва написання хорошого коду Pharo є вибір хороших імен методів, щоб код міг читатись як людська мова. Ви можете уявити об’єкт, який розмовляє з собою і каже «Я сам, використовуй квадратні кути!», «Я сам, виключись!».

Зауважимо маленьку зелену стрілку біля методу (див. Рисунок 3.5). Це означає, що метод існує у суперкласі і є перевизначеним у вашому класі.

**Огляд об’єкта**

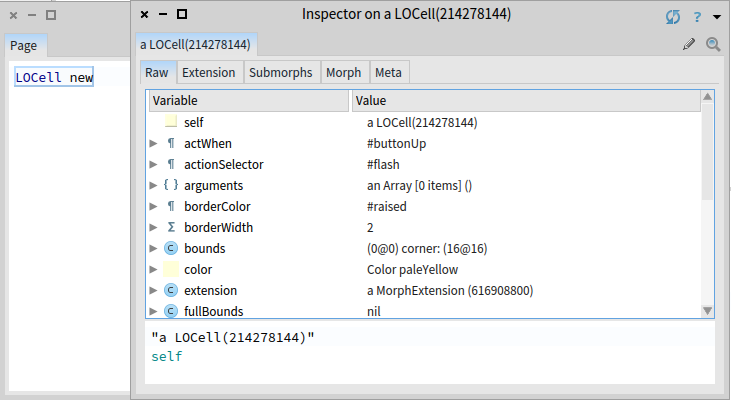
Ви можете миттєво протестувати ефект написаного коду, створивши новий об’єкт LOCell і оглянувши його: Відкрийте Playground, введіть вираз LOCell new і оберіть Inspect it (використовуючи одноіменний пункт меню).

Колонка зліва інспектора показує список полів, а колонка справа – їх значення (див. Рисунок 3.6).

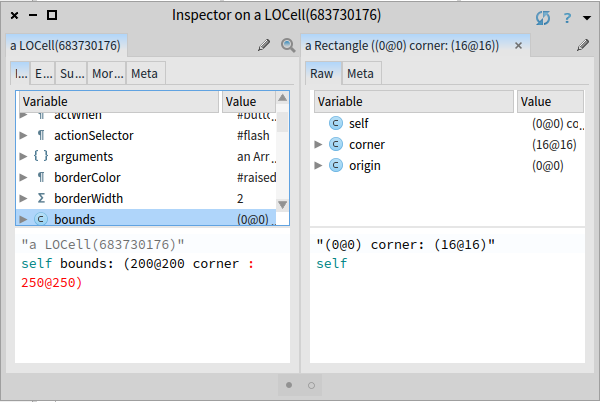
Якщо ви натиснете на одне з полів, інспектор відкриє нову панель з деталями обраного поля (див. Рисунок 3.7).

**Виконання виразів.** Нижня панель інспектора служить міні-пісочницею. Вона корисна тим, що у ній псевдозмінна self прив’язана до вибраного об’єкта.

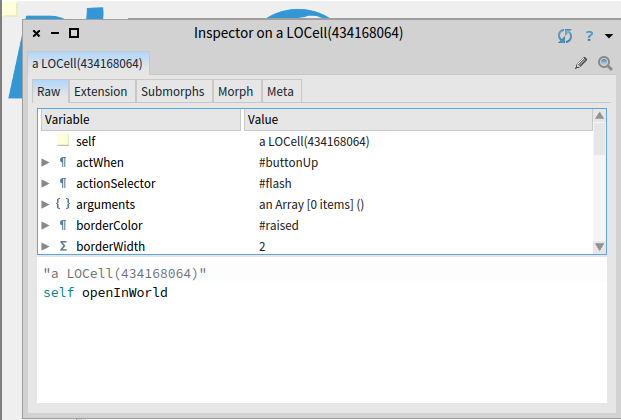
Перейдіть до цієї пісочниці внизу панелі та наберіть наступний текст: self bounds: (200@200 corner : 250@250). Щоб оновити значення, натисніть на кнопку update (маленьке синє коло) у правому верхньому куті панелі. Змінна bounds повинна змінитись у інспекторі. Тепер наберіть текст self openInWorld у міні-пісочниці та виконайте його. У лівому верхньому куті екрану повинна з’явитись клітинка. Вона з’явилась саме у тій позиції, що вказана у змінній bounds.



**Рисунок 3.6** Огляд об'єкту LOCell у інспекторі



**Рисунок 3.7** Коли ми натискаємо на поле, ми бачимо його значення



**Рисунок 3.8** LOCell відкрився у world

**3.6 Визначення класу LOGame**

А зараз створимо інший клас, необхідний для гри, який назвемо LOGame.

**Створення класу**

Зробіть видимим шаблон створення класу у головному вікні браузера. Для цього натисніть на назву пакета (або правий клік на панелі Class і виберіть додати клас (Add Class)). Відредагуйте код, щоб він виглядав наступним чином і прийміть його.

*BorderedMorph subclass: #LOGame*

*instanceVariableNames: ''*

*classVariableNames: ''*

*package: 'PBE-LightsOut'*

Тут ми наслідуємо класс BorderedMorph. Ми також могли б вставити імена змінних класу між лапками, але, поки що, залишимо цей список порожнім.

**Ініціалізація гри**

Тепер визначимо метод initialize для LOGame. Введіть наступний код у браузер, як метод LOGame і спробуйте прийняти його.

*initialize*

*| sampleCell width height n |*

*super initialize.*

*n := self cellsPerSide.*

*sampleCell := LOCell new.*

*width := sampleCell width.*

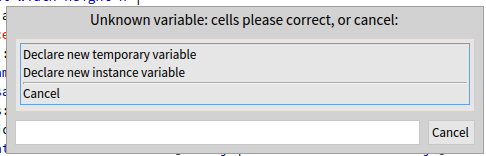
*height := sampleCell height.*

*self bounds: (5 @ 5 extent: (width \* n) @ (height \* n) + (2 \* self*

*borderWidth)).*

*cells := Matrix new: n tabulate: [ :i :j | self newCellAt: i at: j ]*

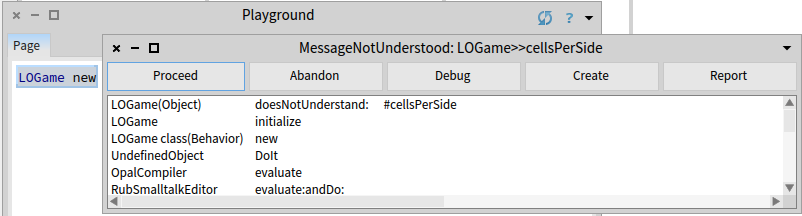
Pharo поскаржиться, що йому невідомо значення cells (див. Рисунок 3.9), та запропонує кілька способів виправити це. Оберіть Declare new instance variable, щоб створити нове поле cells.



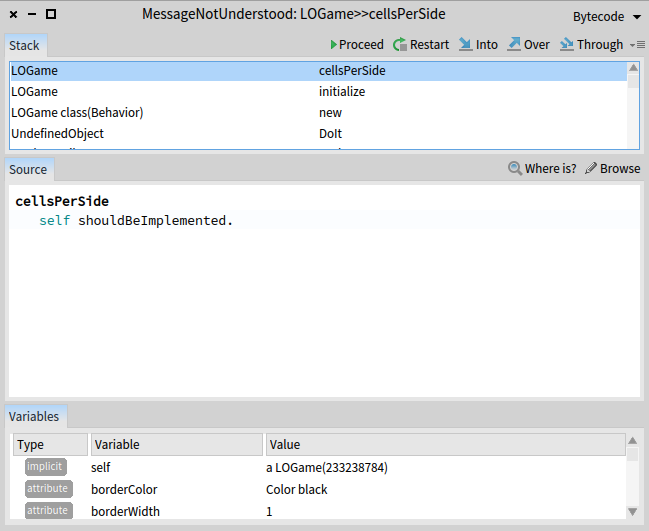
**Рисунок 3.9** Оголошення нового поля cells

**Використання переваг відлагоджувача**

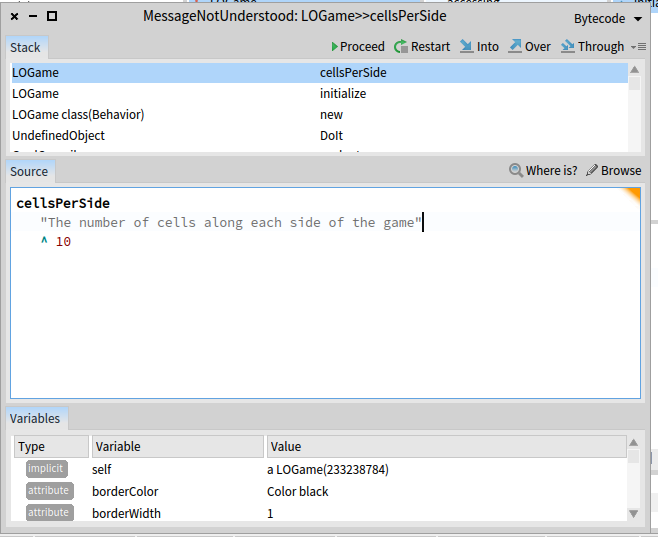
На цьому етапі, якщо ви відкриєте пісочницю (Playgroung), введете LOGame new і виконаєте це, Pharo поскаржиться, що йому невідомі значення деяких виразів (див. Рисунок 3.10). Він повідомить, що не знає про повідомлення cellsPerSize і відкриє відлагоджувач. Але cellsPerSize не є помилкою; це всього лиш метож, який ми ще не визначили. Ми зробимо це згодом.



**Рисунок 3.10** Pharo зустрічає невідомий вираз

Тепер зробимо наступне: введіть LOGame new і виконайте. Не закривайте відлагоджувач. Натисніть на кнопку Create (створити) відлагоджувача. При запиті класу, який міститиме метод, оберіть LOGame та натисніть ok. При запиті протоколу метода введіть accessing. Відлагоджувач створить метод cellsPerSize на льоту і виконає його негайно. Але не дивно, що створений метод всього лиш спричинить виняток (exception) і відлагоджувач знову зупиниться (див. Рисунок 3.11) даючи можливість визначити поведінку методу. 

**Рисунок 3.11**Створення нового методу

У цьому місці можна написати метод. Цей метод важко зробити простішим: він повертає константу 10. Однією з переваг використання констант, як методів, є те, що у випадку розширення програми так, що костанта залежатиме від інших властивостей, метод можна буде легко змінити для обрахування цього значення. 

*cellsPerSide*

*"The number of cells along each side of the game"*

*^ 10*

Рисунок 3.12 Визначення методу cellsPerSide у відлагоджувачі

Визначте метод cellsPerSide у відлагоджувачі. Не забудьте скомпілювати його натиснувши Прийняти (Accept). Ваша ситуація має відповідати зображеній на Рисунку 3.12. Якщо ви натиснете кнопку Продовжити (Proceed), програма продовжить своє виконання – і зупиниться, бо ми не визначили метод newCellAt:. Ми можемо визначити його, як і попередній, але поки що зупинимось, щоб пояснити детальніше, що ми вже зробили. Закрийте відлагоджувач і погляньте на визначення класу ще раз (натиснувшина LOGame на другій панелі системного браузеру). Ви побачите, що браузер змінив визначення класу, щоб те включало змінну cells.

**Вивчаємо метод initialize**

Давайте розберемо метод initialize

*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: (5 @ 5 extent: (width \* n) @ (height \* n) + (2 \* selfborderWidth)).*

*9 cells := Matrix new: n tabulate: [ :i :j | self newCellAt: i at: j ]*

* У 2 рядку вираз | sampleCell width height n | оголошує 4 тимчасові змінні. Їх називають тимчасовими, тому що їхня область видимості та тривалість життя обмежені цим методом. Тимчасові змінні з пояснювальними іменами допомагають зробити код більш легким для сприйняття. Рядки 4-7 встановлюють значення цим змінним.
* Яким великим повинне бути поле для гри? Достатньо великим, щоб вмістити деяку цілу кількість клітинок та достатньо великим, щоб намалювати межу навколо них. Скільки клітинок повинно бути? 5? 10? 100? Наразі ми не знаємо, але, якби й знали, ймовірно ми б змінили це число пізніше. Тому ми делегуємо відповідальність за знання цієї кількості на інший метод, який назвемо cellsPerSide (його ми напишемо згодом). Не лякайтесь цього, насправді це є хорошою практикою присати код, звертаючись до ще не визначених методів. Чому? Поки ми не почали писати метод initialize, ми й не здогадувались, що він нам знадобиться. І на цьому моменті ми можемо дати йому змістовне ім'я та продовжувати, без переривання процесу.
* Четвертий рядок використовуючи цей метод n := self cellsPerSide., відправляє повідомлення cellsPerSide до self, тобто до цього ж об’єкту. Відповідь (кількість клітинок на стороні) присвоюється змінній n.
* Наступні 3 рядки створюють новий LOCell об’єкт і присвоюють його ширину і висоту відповідним тимчасовим змінним.
* Рядок 8 задає межі нового об’єкту. Не вникаючи у деталі, скажемо, що вираз у дужках створює квадрат з лівим верхнім кутом у точці (5,5) і правим нижнім кутом достатньо далеко, щоб вмістити потрібну кількість клітинок.
* Останній рядок присвоює полю cells новостворену матрицю (Matrix) з правильною кількістю рядків та стовпців. Це робиться відправленням повідомлення new: tabulate: до класу Matrix (класи є об’єктами також, тому ми можемо відправлями їм повідомлення). Ми знаємо, що new: tabulate: приймає два аргументи, бо він має дві двокрапки (:) у імені. Аргументи йдуть зразу після двокрапок. Якщо ви звикли до мов, що передають аргументи всередині дужок, це може здатись дивним на початку. Але не панікуйте, це всього лиш синтаксис. І виявляється, що це дуже навіть хороший синтаксис, оскільки назва методу може використовуватись для пояснення ролей аргументів. Наприклад, цілком зрозуміло, що Matrix rows: 5 columns: 2 має 5 рядків та 2 стовпці, а не 2 рядки та 5 стовпців.
* Matrix new: n tabulate: [ :i :j | self newCellAt: i at: j ] створює нову матрицю n на n і ініціалізує її елементи. Початкове значення кожного елемента залежить від його координат. Елемент на позиції (i, j) буде проініціалізовано результатом обчислення self newCellAt: i at: j.

**Організація методів у протоколи**

Перед тим, як визначати інші методи, глянемо на третю панель зверху браузера. Таким самим чином, як перша панель браузера дозволяє нам класифікувати класи у пакети, панель протоколів дозволяє нам класифікувати методи, щоб нам не доводилось працювати з дуже довгим списком імен методів на панелі методів. Такі групи методів називають «протоколами».

Браузер також пропонує нам віртуальний протокол --all--, який містить усі методи класу.

У цьому прикладі, панель протоколів містить протокол, який є некласифікованим. Правий клік по панелі протоколів та оберіть класифікувати всі некласифіковані (categorize all uncategorized), щоб виправити це, та перемістіть метод initialize до нового протоколу, названого initialization.

Але як системний браузер знає, що це правильний протокол? В загальному, Pharo не знає точно, але у цьому випадку метод initialize є також у суперкласі, і Pharo припускає, що наш initialize метод повинен бути у тому ж протоколі, що й метод, який він перевизначає.

**Типографічна конвенція.** Програмісти Pharo часто використовують нотацію Class >> method, щоб ідентифікувати клас, якому належить метод. Наприклад, cellsPerSide метод у класі LOGame буде записаний так: LOGame >> cellsPerSide. Це не є точний синтаксис Pharo, а швидше зручна нотація, щоб вказати «метод екземпляру cellsPerSide, що належить класу LOGame”. (Відповідна нотація для методу класу буде LOGame class >> someClassSideMethod.)

Відтепер, коли ми показуватимемо метод у цій книзі, ми будемо записувати його ім’я у такій формі. Звісно, при наборі коду в браузер, вам не потрібно писати назву класу чи >>; натомість, достатньо переконатись, що відповідний клас обрано на панелі класів.

**Завершення гри**

А зараз давайте визначимо інші методи, що використовуються у LOGame >> initialize. Давайте визначимо LOGame >> newCellAt: at: у протоколі initialization.

**Форматування**. Як ви можете бачити, у коді є порожні рядки та табуляція. Для того, щоб дотримуватись однакової ковенції, ви можете натиснути правим кліком по зоні редагування методу та натиснути Format (або використати CMD-Shift-f). Це відформатує ваш метод.

*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 ]*

Метод визначений вище створює нову LOCell та ініціалізує позицією (i, j) у матриці клітинок. Останній рядок визначає для нової клітинки mouseAction у блоці [ self toggleNeightboursOfCellAt: i at: j ]. Тут визначається поведінка колбеку при натисканні мишкою. Відповідний метод також потрібно визначити.

*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 клітинок на північ, південь, захід і схід від клітинки (i,j). Єдиною складністю є те, що ігрове поле є скінченним, отже нам потрібно переконатись, що сусідня клітинка існує, перед тим, як перемикати її стан.

Перемістіть цей метод у новий протокол, названий game logic. (Права кнопка по панелі протоколів, щоб додати новий протокол). Щоб перемістити (перекласифікувати) метод, можна просто натиснути на його назві та перетягнути на новостворений протокол (див. Рисунок 3.13).

Щоб завершити гру Lights Out, нам потрібно визначити ще два методи у класі LOCell. Цього разу для обробки подій мишки.

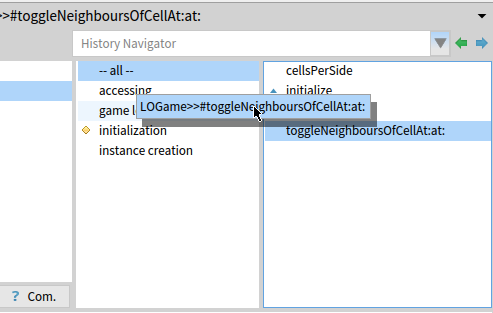


Рисунок 3.13 Перетягніть метод у протокол

*LOCell >> mouseAction: aBlock*

*^ mouseAction := aBlock*

Цей метод не робить нічого, окрім того, що присвоює змінну mouseAction клітинки аргументу і повертає нове значення. Будь-який метод, що змінює значення поля у такий спосіб називається сеттером (setter method). Метод, що повертає поточне значення поля називається геттером (getter method).

**Конвенція геттерів/сеттерів.** Якщо ви звикли до геттерів та сеттерів у інших мовах програмування, ви можете очікувати, що ці методи називатимуться setMouseAction та getMouseAction. Але конвенція Pharo є іншою. Геттер завжди має таке саме ім’я, як і змінна, яку він повертає, а сеттер має таке ж ім’я, але з “:” вкінці: mouseAction та mouseAction:. Разом геттери та сеттери називають методами доступу і за конвенцією повинні бути розміщені у протоколі accessing. У Pharo всі змінні є приватними для об’єкту, який містить їх, тому єдиний спосіб для інших об’єктів читати чи записувати ці змінні є через методи доступу, як ці. Змінні також можуть бути доступними у підкласах.

Перейдіть до класу LOCell, визначте LOCell >> mouseAction: і перемістіть у протокол accessing.

Нам залишилось визначити метод mouseUp:. Він буде викликатись автоматично GUI фреймворком при відпусканні кнопки миші, коли курсор є над певною клітинкою. Додайте метод LOCell >> mouseUp: і потім виконайте автоматичне класифікування методів (Categorize).

Цей метод відправляє повідомлення value до об’єкта, що зберігається у полі mouseAction. У LOGame >> newCellAt: i at: j ми створили блок [ self toggleNeighboursOfCellAt: i at: j ], який перемикає усіх сусідів клітинки, та присвоїли цей блок до mouseAction клітинки. Отже, відправлення повідомлення value спричинить виконання цього блоку і зміну станів усіх сусідніх клітинок.

*LOCell >> mouseUp: anEvent*

*mouseAction value*

**Запускаємо код**

Це все: гра Lights Out завершена! Якщо ви виконали усі кроки, ви зможете зіграти у гру, що складається з всього лише 2 класів та 7 методів. У пісочниці (Playground) наберіть LOGame new openInWorld та виконайте (Do it).

Гра відкриється і ви зможете натискати на клітинки і бачити, як усе працює. Принаймні в теорії... Коли ви клікнете на клітинці, з’явиться відлагоджувач. У верхній частині вікна відлагоджувача ви побачите стек виконання, що показує усі активні методи. Вибравши будь-який з них, у середній панелі з’явиться код Smalltalk, що виконується у цьому методі. Частина, що спричинила помилку, буде підсвічуватись.

Натисніть на рядку з написом LOGame >> toggleNeighboursOfCellAt: at: (біля верху). Відлагоджувач покаже контекст виконання у методі, де з'явилась помилка (див. Рисунок 3.14).

Тут ми можемо бачити у першому рядку верхньої панелі, що повідомлення toggleState було відправлене до екземпляру LOGame, хоча воно повинне було бути відправленим екземпляру LOCell. Найбільш ймовірно, що проблема з ініціалізацією клітинок матриці. Переглянувши код LOGame >> initialize, бачимо, що cells заповнено з значеннями, що повертає newCellAt: at:, але якщо ми глянемо на цей метод, то побачимо, що там немає виразу повернення! За замовчуванням, метод повертає self, який у випадку newCellAt: at: є справді екземпляром LOGame. Синтаксом для повернення значення з методу у Pharo є ^.

Закрийте вікно відлагоджувача. Добавте вираз ^ c вкінці методу LOGame >> newCellAt:at:, щоб він повертав с.

*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*

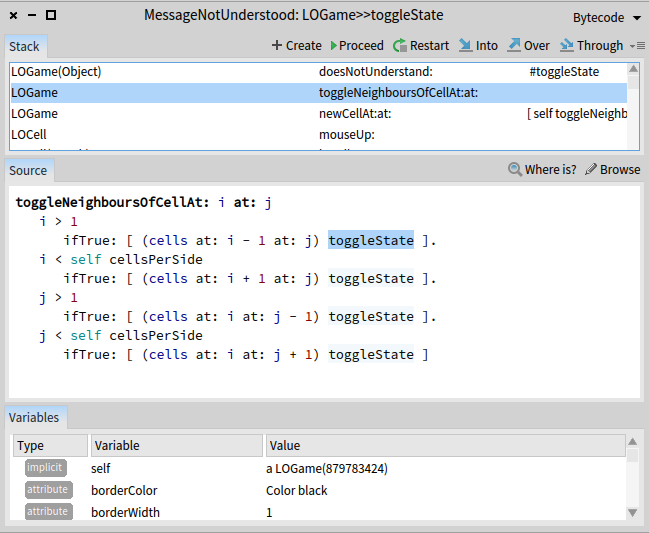


Рисунок 3.14 Відлагоджувач з обраним методом toggleNeighboursOfCell:at:

Зазвичай, ви можете виправляти код зразу у відлагоджувачі і, натиснувши Proceed, продовжити виконання програми. У нашому випадку, через те, що помилка була у ініціалізації об'єкту, а не у методі, у якому сталась помилка, найпростіше буде закрити вікно відлагоджувача, зупинити запущений екземпляр гри (CMD-Alt-Shift та клік) і створити новий.

Виконайте LOGame new openInWorld знову, тому що старий екземпляр гри досі містить блок з старою логікою.

Зараз гра повинна працювати правильно... або майже правильно. Якщо ми порухаємо мишку у момент часу між натисканням кнопки та відпусканням, тоді клітинка, над якою є мишка, також змінить свій стан. Це поведінка, яку ми успадкували від SimpleSwitchMorph. Ми можемо це виправити просто перевизначивши mouseMove:, щоб він нічого не робив:

*LOCell >> mouseMove: anEvent*

Нарешті ми закінчили!

**Про відлагоджувач.** За замовчуванням, коли трапляється помилка у Pharo, система відображає відлагоджувач. Проте ми можемо повністю контролювати цю поведінку. Наприклад, ми можемо записати помилку у файл. Ми можемо навіть серіалізувати стек виконання у файл, заархівувати і відкрити його у іншій програмі. Але у режимі розробки відлагоджувач доступний, щоб дозволити нам рухатись так швидко, як це можливо. У готових системах, розробники часто налаштовують відлагоджувач, щоб приховувати свої помилки від клієнтів.

**Зберігання та поширення коду Pharo**

Тепер, коли гра Lights Out працює, ви напевно хочете зберегти її десь, щоб заархівувати та поділитись з друзями. Звісно ви можете зберегти весь образ Pharo і показати вашу першу програму, запустивши його, але ваші друзі напевно мають власний код у їхніх образах і не захочуть втратити його, використавш ваш образ. Що нам потрібно, так це спосіб витягнути код з вашого образу Pharo, щоб інші програмісти змогли внести його у їхні власні образи.

ЗАУВАЖЕННЯ Ми будемо розглядати різні шляхи збереження та поширення коду у наступних розділах (Розділ: Поширення коду та контроль версійності). Зараз ми оглянемо лише декілька доступних способів.

**Збереження чистого коду**

Найпростіший спосіб зробити це – виділити код у файл. Правий клік по панелі пакетів дасть можливість виділити у файл (File Out) весь пакет PBE-LightsOut. Вихідний файл буде більш-менш читабельний для людини, але він призначений в першу чергу для комп’ютерів, а не для людей. Ви можете надіслати цей файл друзям і вони можуть внести його у власний образ Pharo використовуючи файловий браузер.

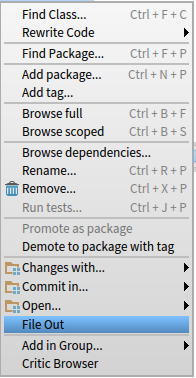


Рисунок 3.15 Виділення коду у файл

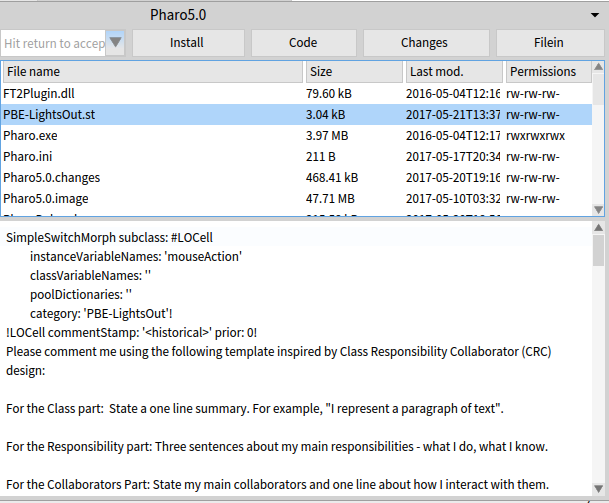


Рисунок 3.16 Імпорт коду з файлового браузеру

Правий клік по пакету PBE-LightsOut та оберіть File Out (див. Рисунок 3.15). У папці з образом повинен з’явитись файл з назвою PBE-LightsOut.st.

Відкрийте новий образ Pharo та, використовуючи інструмент File Browser (Tools 🡪 File Browser) імпортуйте пайл PBE-LightsOut. Переконайтесь, що гра працює на новому образі.

**Пакети Monticello**

Хоча розглянутий вище спосіб робити знімки коду є досить зручним, він безумовно застарілий. Так само, як більшість проектів з відкритим кодом вважають зручним зберігати код у репозиторіях, використовуючи SVN чи Git, так і програмісти Pharo вважають більш зручним керувати їхнім кодом, використовуючи пакети Monticello. Такі пакети представляються файлами з розширенням .mcz. Насправді вони є заархівованими пакунками, що містять весь код вашого пакету.

Використовуючи браузер пакетів Monticello, ви можете зберігати пакети у репозиторіях на серверах різного типу, включно з FTP та HTTP серверами. Ви також можене записувати пакети до репозиторія у локальній файловій системі. Копія вашого пакету завжди кешується на локальному диску у папці package-cache. Monticello дозволяє зберігати кілька версій однієї програми, зливати версії, переходити до старих версій і переглядати різницю між версіями. По факту, Monticello – це розподілена система контролю версій. Це означає, що розробники можуть зберігати свою роботу у різних місцях, а не в одному репозиторії, як це є у випадку CVS чи Subversion.

**Браузер Monticello**

Відкрийте браузер Monticello з меню World (див. Рисунок 3.17).

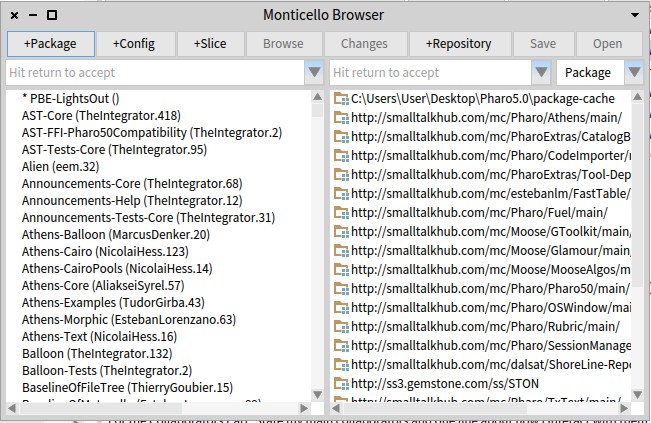


Рисунок 3.17 Браузер Monticello. Пакет PBE-LightsOut поки не збережено

У правій частині відображається список репозиторіїв Monticello, який включає усі репозиторії з яких завантажувався код у образ, який ви використовуєте. Верхнім елементом списку є локальна папка з назвою package-cache. Вона містить кешовані копії усіх пакетів, які завантажувались або публікувались по мережі. Вона також дозволяє працювати з місць, у яких немає доступу до інтернету або доступ є настільки повільним, що ви не хочете часто зберігати зміни у віддаленому репозиторії.

**Збереження та завантаження коду з Monticello**

У лівій частині браузера Monticello є список пакетів, які є завантаженими в образ. Пакети, які були модифікованими, позначені зірочкою. (Їх деколи називають брудними пакетами). Якщо ви виберет пакет, список репозиторіїв показуватиме лише ті репозиторії, що містять копію вибраного пакету.

Добавте пакет PBE-LightsOut до браузера Monticello використовуючи кнопку +Package та введіть PBE-LightsOut.

**SmalltalkHub: Github для Pharo**

На нашу думку, найкращий спосіб зберігати код та ділитись ним – створити обліковий запис для проекту на SmalltalkHub. SmalltalkHub схожий на Github: це веб-інтерфейс для HTTP Monticello сервера, що дозволяє керувати вашими проектами. Публічний сервер доступний за адресою http://www.smalltalkhub.com/.

Щоб мати змогу використовувати SmalltalkHub вам потрібен обліковий запис. Відкрийте сайт у браузері. Натисніть на Join та слідуйте інструкціям, щоб створити новий обліковий запис. Увійдіть в свій обліковий запис. Натисніть на + New project, щоб створити новий проект. Заповніть необхідну інформацію та натисніть кнопку Register project. Ви будете перенаправлені на сторінку вашого профіля, де, з правого боку, буде список ваших проектів, та проектів інших програмістів, за якимим ви слідкуєте. Натисніть на новостворений проект.

Під написом з назвою Monticello реєстрації ви побачите блок з повідомленням smalltalk, схожим на

*MCHttpRepository*

*location: 'http://www.smalltalkhub.com/mc/UserName/ProjectName/main'*

*user: 'YourName'*

*password: 'Optional\_Password'*

Скопіюте вміст та поверніться до Pharo. Після створення проекту на SmalltalkHub, потрібно вказати системі Pharo, як його використовувати. З вибраним пакетом PBE-LightsOut, натисніть +Repository нопку у браузері Monticello. Ви побачите список різних доступних типів репозиторіїв. Щоб додати SmalltalkHub репозиторій, оберіть smalltalkhub.com. Вам покажуть діалогове вікно, де потрібно буде ввести інформацію про сервер. Вставте код, скопійований раніше з SmalltalkHub (див. Рисунок 3.18). Це повідомлення вказує Monticello, де знаходиться проект онлайн. Ви можете також ввести своє ім’я користувача та пароль. Якщо цього не зробити, Pharo буде вимагати їх кожного разу при спробі зберегти зміни в онлайн репозиторії на SmalltalkHub.

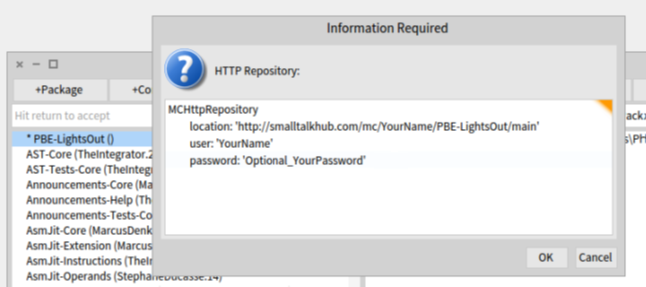


Рисунок 3.18 Створення першого репозиторію на SmalltalkHub

Після того, як ви приймете зміни, новий репозиторій буде розміщено у правій частині браузера Monticello. Натисніть кнопку Save, щоб зберегти першу версію вашої гри Lights Out на SmalltalkHub. Не забудьте додати коментар, щоб ви чи будь-хто інший, розуміли, що міститься у цьому комміті (див. Рисунок 3.19).



Рисунок 3.19 Перший комміт

Щоб завантажити пакет у ваш образ, відкрийте Monticello, оберіть репозиторій, в якому розміщений пакет та натисніть кнопку відкрити. З'явиться нове вікно з двома колонками. У лівій буде пакет, який має бути завантаженим, а у правій – версія цього пакету. Оберіть пакет та версію, яку ви хочете завантажити та натисніть кнопку завантажити.

Відкрийте репозиторій PBE-LightsOut, який ви щойно зберегли. Ви побачите пакет, який ви щойно зберегли.

Monticello маж багато інших функцій, які ми розглянемо детальніше у розділі Поширення коду та контроль версійності.

**Підсумки розділу**

У цьому розділі ми розглянули створення пакетів, класів та методів. Також ми навчились користуватись системним браузером, інспектором, відлагоджувачем та браузером Monticello.

* Пакети – це групи пов’язаних класів.
* Новий клас створюється відправленням повідомлення своєму суперкласу.
* Протоколи – це групи пов'язаних методів у класі.
* Новий метод створюється або змінюється редагуванням його визначення у браузері і подальшим прийняттям змін.
* Інспектор надає простий графічний інтерфейс для спостереження та взаємодії з довільними об'єктами.
* Браузер відслідковує використання неоголошених змінних та пропонує можливі виправлення.
* Метод initialize автоматично виконується при створенні об’єкта у Pharo. Ви можете вставити будь-який код ініціалізації там.
* Відлагоджувач надає високорівневий графічний інтерфейс для спостереження та зміни стану запущеної програми.
* Ви можете ділитись кодом шляхом запису у файл пакету, класу чи методу.
* Кращий спосіб поділитись кодом – використання Monticello для керування зовнішнім репозиторієм, наприклад визначеним, як проект SmalltalkHub