**Pharo by Example 5**

Stéphane Ducasse

Dimitris Chloupis

Nicolai Hess

Dmitri Zagidulin

January 14, 2017

commit 7a4b6fc\*

**Фаро 5 на прикладах**

Стефан Дюкас

Дімітріс Хлоупіс

Ніколай Хесс

Дмитрій Загідулін

14 січня 2017 року

переклад українською з доповненнями

Сергій Ярошко

# Розділ 1. Передмова

## 1.1 Що таке Фаро

Фарó (Pharo) – це сучасна динамічно типізована мова програмування з відкритим кодом та середовище програмування на льоту, натхненником яких став Smalltalk. Фаро та його екосистеми складаються з шести основних елементів:

* динамічна мова програмування з таким простим синтаксисом, що його можна вмістити на поштівці, і, водночас, читабельним навіть при першому знайомстві;
* середовище програмування, що дозволяє програмістові змінювати код на льоту під час його виконання, не сповільнюючи свій виробничий процес;
* інтегроване середовище розробки з вичерпним комплектом інструментів, що допомагають керувати складним кодом і спонукають до проектування правильної архітектури;
* багата бібліотека класів, яка робить середовище таким потужним, що його можна сприймати як віртуальну операційну систему, встановлену на віртуальну машину з дуже швидким JIT-компілятором та повним доступом до всіх засобів і бібліотек «рідної» операційної системи;
* культура розробки, у якій зміни та вдосконалення заохочуються та високо цінуються;
* спільнота, яка вітає програмістів з усіх кінців світу з будь-якими вміннями та знанням усяких мов програмування.

Фаро прагне надати вільно поширювану платформу для професійного створення програмного забезпечення та потужну надійну основу для досліджень і розробки динамічних мов і середовищ. Pharo також слугує середовищем реалізації для платформи Seaside розробки веб-додатків, яка доступна на сайті <http://www.seaside.st>.

Ядро Фаро містить лише той код, що розповсюджується за ліцензією МІТ (група ліцензій, розроблених Масачусетським технологічним інститутом для розповсюдження вільного програмного забезпечення). Проект Pharo розпочався у березні 2008 року як відгалуження Squeak, крос-платформного втілення класичної системи програмування Smalltalk-80, а перша бета-версія 1.0 побачила світ 31 липня 2009 року. Поточною версією Pharo є 6.1. Її було випущено 24 липня 2017 року. Триває розробка альфа-версії 7.0.

Фаро є дуже мобільним, оскільки може працювати в багатьох операційних системах: OS X, Windows, Linux, Android, iOS та Raspberry Pi. Його легко переносити на нову платформу, бо навіть його віртуальна машина написана на Smalltalk. Це робить його також легким для налагодження, розуміння та внесення змін і доповнень. Фаро є рушієм широкого спектру інноваційних проектів від мультимедійних застосунків і освітніх платформ до комерційних середовищ розробки веб-застосувань.

Однією з головних рис Pharo є те, що воно є справді новим втіленням Smalltalk, а не звичайною копією колишніх його реалізацій. Ми усвідомлюємо, що проекти створення всього й одразу рідко коли досягають успіху, тому Pharo прихильно ставиться до еволюційних і поступових змін. Ми хочемо мати можливість експериментувати з важливими новими засобами чи бібліотеками. Еволюція означає, що Pharo допускає помилки і не прагне досягти досконалих рішень за один великий крок, навіть, якщо б нам цього дуже хотілося. Натомість Pharo підтримує невеликі послідовні зміни, але зроблені багато-багато разів. Успіх Pharo залежить від доробку його спільноти. Врешті, Pharo є не тільки для читання, він щодня інтегрує зміни, зроблені спільнотою. Pharo отримав доповнення від сотні авторів з цілого світу, і ви теж можете вплинути на Pharo!

## 1.2 Для кого ця книга

Попередня версія цієї книги базувалася на Pharo 1.4. Англомовна версія була оновлена для узгодження з Pharo 5.0, для перекладу на українську використано Pharo 6.1. Наша книга описує різні аспекти Фаро, починаючи від основ і продовжуючи темами середньої складності. Просунуті теми представлено в «Deep into Pharo» – книзі про внутрішній світ Фаро. Нова книга «Enterprise Pharo: a Web Perspective» зацікавить читачів, які розробляють веб-додатки. Обидві книги вільно доступні на <http://books.pharo.org>.

Ця книга не вчитиме вас програмувати. Читач мав би знати якісь мови програмування. Бажаним є знання основ об’єктно-орієнтованого програмування.

Книга познайомить вас із середовищем програмування Фаро, мовою та відповідними інструментами. Вам буде продемонстровано загальні ідіоми та практичні підходи, але головний наголос буде зроблено на технології, а не на об’єктно-орієнтованому проектуванні. Ми покажемо, де тільки можливо, численні приклади. На це нас надихнула чудова книжка про Smalltalk Алека Шарпа[[1]](#footnote-1).

Є багато інших книжок, присвячених Smalltalk, вільно доступних у всесвітній мережі, проте жодна з них не присвячена власне Фаро. Дивіться, наприклад, <http://stephane.ducasse.free.fr/FreeBooks.html>.

## 1.3 Порада

Не переймайтеся, якщо ви не зрозумієте відразу якусь частину коду на Smalltalk. Ви не повинні знати геть усе! Алан Найт виразив цей принцип так[[2]](#footnote-2):

**Намагайтесь не хвилюватися**. Програмісти-початківці на Smalltalk часто стикаються з труднощами тому, що думають, що вони мусять докладно розуміти, як все працює, перш ніж його використовувати. Це означає, що вони затрачають чимало часу, перш ніж запрограмують *Transcript show: 'Hello World'*. Та однією з величезних переваг ООП є можливість на запитання «Як це працює?» відповісти «Мені байдуже».

Не вагайтеся ні миті запитати нас, якщо ви чогось не зрозуміли, складного чи простого. Ви можете скористатися нашими списками розсилання (pharo-users@lists.pharo.org, pharo-dev@lists.pharo.org) або IRC чи Slack. Ми любимо запитання і радо вітаємо зацікавлених з будь-якими навиками.

## 1.4 Відкрита книга

Ця книга є відкритою у такому розумінні:

* Книгу опубліковано під ліцензією Creative Commons із зазначенням авторства і розповсюдження на таких самих умовах (скорочено: CC BY-SA). Це означає, що Ви можете вільно поширювати і адаптувати цю книгу, якщо тільки дотримуєтесь вимог ліцензії, доступних за таким URL: <http://creativecommons.org/licenses/by-sa/3.0/>
* Ця книга описує лише ядро Фаро. Ми б хотіли заохотити й інших авторів додавати до неї параграфи, присвячені тим частинам Фаро, які ми не описали. Якщо Ви хотіли б долучитися до такої праці, будь ласка, сконтактуйте з нами. Ми б дуже хотіли побачити більше книг, присвячених Фаро!
* Ви можете також зробити свій внесок безпосередньо до цієї книги через Github. Репозиторій Github ви можете знайти тут [https://github.com/SquareBracket Associates/UpdatedPharoByExample](https://github.com/SquareBracket%20Associates/UpdatedPharoByExample). Просто додайте свою пропозицію до списку розсилки, дотримуючись розташованих там інструкцій.

## 1.5 Спільнота Фаро

Спільнота Фаро є дружелюбною і активною. Нижче подано короткий перелік ресурсів, які можуть бути корисними для вас:

* <http://www.pharo.org> – головний сайт Фаро.
* Ви можете знайти нас у IRC на сервері freenode.net на каналі «pharo».
* SmalltalkHub (<http://www.smalltalkhub.com/>) – аналог SourceForge/Githhub для проектів Фаро. Тут зберігається багато додаткових пакетів класів і проектів.
* Фаро також активний на Slack – платформі для чатів, що базуються на IRC (<http://pharoproject.slack.com>) – достатньо попросити про запрошення на <http://slackinvites.pharo.org>. Запрошуємо усіх!

## 1.6 Приклади і вправи

Ми намагаємося надати стільки прикладів, скільки можливо. Зокрема, є багато прикладів, що демонструють фрагменти коду, придатні для безпосереднього виконання. Ми використовуємо довгу стрілку, щоб вказати на результат, який ви отримаєте, якщо виокремите певний вираз і виберете команду *print it*:

3 + 4

>>>7 "якщо ви позначите 3+4 і виберете 'print it', то отримаєте 7"

Якщо ви хочете виконувати у Фаро такі фрагменти коду і не маєте часу набирати їх власноруч, то можете завантажити текстовий файл з програмами всіх прикладів із сайту <http://books.pharo.org/pharo-by-example/> з бічної панелі *Resources*.

## 1.7 Подяки

Перш за все ми хочемо подякувати Алану Кею (Alan Kay), Дену Інґальсу (Dan Ingalls) та їхній команді за те, що зробили Squeak, це дивовижне Smalltalk середовище розробки, доступним в якості проекту з відкритим кодом, адже Pharo бере свій початок саме від нього. Існування Pharo також було б неможливе без неймовірної праці Squeak-розробників.

Ми також хочемо подякувати Гіларі Фернандес (Hilaire Fernandes) і Сержу Стінквічу (Serge Stinckwich) за дозвіл перекласти їхні статті про Smalltalk та Дам’єну Касо (Damien Cassou) за написання розділу, присвяченого потокам. Ми особливо дякуємо Олександру Бергелю (Alexandre Bergel), Орлі Гріві (Orla Greevy), Фабріціо Перін (Fabrizio Perin), Лукасу Ренглі (Lukas Renggli), Джорджу Рессіа (Jorge Ressia) та Ервану Вернлі (Erwann Wernli) за їхні детальні рецензії.

Дякуємо Університету Берну (Швейцарія) за люб’язну підтримку цього проекту з відкритим кодом та за хостинг сайту цієї книги.

Ми також дякуємо спільноті Squeak за повідомлення про помилки, виявлені у першому виданні цієї книги.

Ми також дякуємо спільноті Фаро за їхній ентузіазм і підтримку проекту створення книги. Дякуємо всім перекладачам.

## 1.8 Спеціальні подяки

Ми хочемо подякувати першим авторам цієї книги! Без їхньої початкової версії було б дуже важко зробити цю. Книга «Фаро на прикладах» (Pharo by Example) є основною для залучення новачків, тому має величезне значення.

Дякуємо Манфреду Крюнерту (Manfred Kröhnert), Маркусу Шлагеру (Markus Schlager), Вернеру Касенсу (Werner Kassens), Майклу Окіфу (Michael OKeefe), Арьї Хофману (Aryeh Hoffman), Полю Макінтошу (Paul MacIntosh), Ґаураву Сінху (Gaurav Singh), Джіґясу Ґруверу (Jigyasa Grover), Крейґу Алену (Craig Allen), Сержу Стінквічу (Serge Stinckwich), avh-on1, Юрію Тимчуку, zio-pietro за виправлення помилок і відгуки. Особлива подяка Демієну Кассу (Damien Cassou) та Сирілю Ферліко (Cyril Ferlicot) за їхню велику допомогу в оновленні книги. Нарешті хочемо подякувати інституту Inria за постійну значну фінансову підтримку і членам команди RMoD за невтомне просування Фаро.

Окрема велика подяка Демієну Полле (Damien Pollet) за його чудовий шаблон книги і групі студентів факультету прикладної математики та інформатики ЛНУ імені Івана Франка (м. Львів) за допомогу в перекладі книги українською.

# Розділ 2. Короткий огляд Фаро

У цьому розділі ми проведемо огляд Фаро без заглиблення в деталі, щоб допомогти вам освоїти середовище програмування. При цьому буде досить багато нагод випробувати написане, тому було б дуже добре, читаючи цей розділ, мати під рукою комп’ютер.

Зокрема, ви запустите Pharo, вивчите різноманітні способи взаємодії з системою і познайомитеся з деякими основними інструментами. Ви навчитеся також визначати новий метод, створювати об’єкт і надсилати йому повідомлення.

*Зауваження*. Більшість прикладів вступного матеріалу працюватимуть у Pharo довільної версії, тому ви можете продовжувати використовувати встановлене середовище Pharo, якщо воно у вас уже є. Проте, оскільки книга написана для Pharo 6.1, Ви можете помітити відмінності між тим, що написано, і тим, як виглядає чи як веде себе Ваша система.

## 2.1 Встановлення Pharo

### Завантаження Pharo

Pharo можна вільно завантажити з <http://pharo.org/download>. Клацніть на кнопці з назвою Вашої операційної системи, щоб завантажити відповідний zip-файл. Наприклад, повний дистрибутив Pharo 6.1 для OS X доступний за адресою [https://files.pharo.org/ platform/Pharo6.1-mac.zip](https://files.pharo.org/%20platform/Pharo6.1-mac.zip).

Розгорніть архів, і ви отримаєте все, що потрібно для запуску Pharo, включно з віртуальною машиною, файлами образу і вихідних текстів, як описано далі.

**Використання зручних скриптів.** <http://files.pharo.org/get/> надає колекцію скриптів для завантаження конкретних версій Pharo. Це справді зручно для автоматизації процесу. Для завантаження найновішої версії 6.1 системи використовуйте скрипт

wget -O- https://get.pharo.org | bash

### Встановлення Pharo

Pharo є автономним застосунком, тому не потребує жодних додаткових інсталяцій у вашій системі. Залежно від вашої платформи, завантажте відповідний архів, розгорніть його на своєму локальному комп’ютері в директорії за власним вибором, і ви зможете запускати Pharo. У випадку Ubuntu Linux є додаткові параметри встановлення за допомогою Pharo PPA. Pharo можна також встановити з командного рядка.

## 2.2 Файли компонентів Pharo

Pharo складається з чотирьох головних файлів. Під час читання цієї книги ви не будете взаємодіяти з ними безпосередньо, проте корисно розуміти їхнє призначення.

1. *Віртуальна машина* (VM) – єдиний компонент Pharo, що відрізняється для кожної операційної системи. VM є виконавчою системою, подібною до віртуальної машини Java. Кожного разу, коли користувач компілює фрагмент тексту програми, компілятор генерує байт-код Smalltalk. VM отримує такий байт-код, перетворює його на команди процесора і виконує їх. Pharo постачається з Cog VM, з дуже швидкою JIT-машиною (JIT – just in time, машина, що компілює на льоту). Виконуваний файл віртуальної машини називають:

* *Pharo.exe* для Windows;
* *pharo* для Linux;
* *Pharo* для OS X (всередині пакету його також називають *Pharo.app*).

Інші компоненти можна переносити між операційними системами. Їх можна копіювати і запускати на відповідній VM.

1. У *.source* файлі зберігається вихідний код усіх частин системи, що змінюється нечасто. Цей файл важливий, оскільки файл образу зберігає лише об’єкти з компільованими методами та їхніми байт-кодами і не містить вихідних текстів. Зазвичай source-файл генерують один раз під час випуску нової версії Pharo. Для Pharo 6.1 він називається *PharoV60.sources*.
2. *Файл змін* зберігає протокол усіх змін вихідного тексту від моменту створення .*source* файлу, зокрема, всіх зроблених вами змін, поки ви програмували у Pharo. Записується історія змін кожного методу, що полегшує їх затвердження чи скасування. Це означає, що ви зможете відновити всі запрограмовані вами методи з .*changes* файла навіть тоді, коли образ системи не вдалося зберегти через зависання чи через власну забудькуватість. Кожен реліз середовища постачається з майже порожнім файлом змін, що називається відповідно до релізу, наприклад, *Pharo6.1.changes*. Файл змін пов’язаний з файлом образу. Вони завжди працюють разом.
3. Поточний *образ системи* є миттєвим знімком стану працюючого Pharo. Він міститься у *.image* файлі, що має крос-платформний формат. Файл образу зберігає стан усіх об’єктів системи, актуальний на певний момент часу, включно з класами і методами, оскільки вони також є об’єктами. Образ є контейнером віртуальних об’єктів*.* Файл образу також називають відповідно до релізу, наприклад, *Pharo6.1.image*. Він синхронізований з файлом змін *Pharo6.1.changes*.

### Пара Image/Change

Разом з Pharo ви отримуєте живе середовище програмування, яке можна пристосувати для власних потреб. Його початковою точкою є *.image* та *.changes* файли з інсталяційного архіву. Коли ви працюєте з Pharo, вони зазнають змін, тому переконайтеся, що вони розташовані в каталозі, в якому ваша операційна система дозволяє вам записувати. Не забудьте вилучити *.image* та *.changes* файли зі списку перевірки своєї антивірусної програми. Завжди зберігайте ці два файли разом і ніколи не редагуйте їх за допомогою редактора текстів, оскільки Pharo зберігає в них за власними правилами об’єкти, з якими ви працювали, і зміни коду, які ви зробили. Було б добре зберігати запасну копію завантажених *.image* та *.changes* файлів. Тоді ви завжди зможете розпочати роботу з самого початку і повторно завантажити створений вами код. Проте найефективнішим способом зберігання коду є використання системи контролю версій, яка забезпечує легке копіювання коду та відслідковування змін.

### Спільне встановлення

Ви можете помістити описані компоненти Pharo до одного каталогу, проте частіше віртуальну машину та *.source* файл розташовують в окремому каталозі і надають до нього доступ тільки для читання усім користувачам. Оберіть будь-який варіант, який буде зручний для вас та ефективний для вашої операційної системи.

## 2.3 Запуск Фаро

Запустіть Pharo так, як це звичайно роблять у вашій операційній системі: перетягніть *.image* файл на значок віртуальної машини, або двічі клацніть на значку *.image* файла, або надрукуйте в командному рядку ім’я віртуальної машини і слідом – повне ім’я *.image* файла.

* У системі OS X двічі клацніть пакунок (bandle) *Pharo5.0.app* у розархівованому завантаженні.
* У системі Linux двічі клацніть (або запустіть з командного рядка) виконуваний bash-скрипт *pharo*.
* У системі Windows зайдіть у розархівовану папку Pharo і двічі клацніть *Pharo.exe*.

Загалом Pharo намагається «робити правильні речі». Якщо ви двічі клацнете на VM, то віртуальна машина шукатиме у своїй папці файл образу, щоб завантажити його. Pharo дає змогу зберігати образ у файлі з довільним іменем. Це зручно, коли ви провадите різні розробки: кожному проекту відповідатиме свій образ. Отож, запущена VM може знайти декілька .*image* файлів. У цьому випадку машина розпочинає діалог вибору файла, щоб користувач міг вказати потрібний йому образ.

Якщо ви двічі клацнете на .*image* файлі, то операційна система спробує знайти відповідну програму, щоб відкрити його. Можливо, вам доведеться допомогти їй у цьому.

Якщо на вашому комп’ютері інстальовано декілька віртуальних машин, то операційна система може вибрати не ту. У цьому випадку надійніше перетягти файл образу на значок потрібної віртуальної машини чи скористатися командним рядком.

### Запуск Pharo з командного рядка

Схема команди запуску Pharo з консолі складається з двох частин:

<Pharo executable> <path to Pharo image>

**Командний рядок Linux**. Припустимо, що ми перебуваємо в розархівованій папці *pharo6.1*. Тоді команда запуску має вигляд:

./pharo shared/Pharo6.1.image

**Командний рядок OS X**. Припустимо, що ми перебуваємо в директорії з розархіво­ваним пакетом *Pharo6.1.app*. Тоді команда:

Pharo6.1.app/Contents/MacOS/Pharo

Pharo6.1.app/Contents/Resources/Pharo6.1.image

При використанні пакету Pharo ви мусите вибирати з контекстного меню програми *Pharo6.1.app* команду ”Show Package Contents”, щоб отримати доступ до образу системи. Якщо доводиться робити це часто, то просто завантажте окремо пару образ-файл змін і перетягайте файл образу на значок *Pharo6.1.app*.

**Командний рядок Windows**. Якщо ми перебуваємо в розархівованій папці *Pharo6.1*, то команда має вигляд:

Pharo.exe Pharo6.1.image

У загальному випадку потрібно вказати повний шлях до VM. Наприклад, так:

C:\Program Files\Pharo6.1\Pharo.exe Pharo6.1.image

## 2.4 Програма PharoLauncher

Якщо ви використовуєте різні версії Pharo, створюєте багато образів системи, то в пригоді вам стане спеціальна програма – PharoLauncher. Це інструмент, що допомагає завантажувати та впорядковувати образи Pharo. Він дуже зручний як для отримання нових версій Pharo, так і для оновлення наявних (для виправлення виявлених помилок). Також він надає вам доступ до попередньо сконфігурованих образів зі спеціальними бібліоте­ками, що звільняє від ручного встановлення та налаштування таких бібліотек.

Опис PharoLauncher можна прочитати на <https://github.com/pharo-project/pharo-launcher>, а завантажити відповідну до операційної системи програму – на SmalltalkHub за адресою <http://smalltalkhub.com/#! /~Pharo/PharoLauncher> чи на файл-сервері pharo.org за адресою <http://files.pharo.org/pharo-launcher/>.

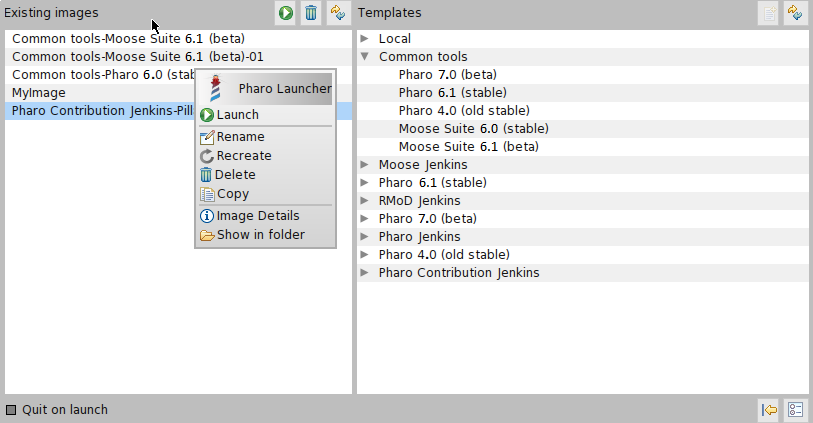


Рис. 2.1 Інтерфейс користувача PharoLauncher

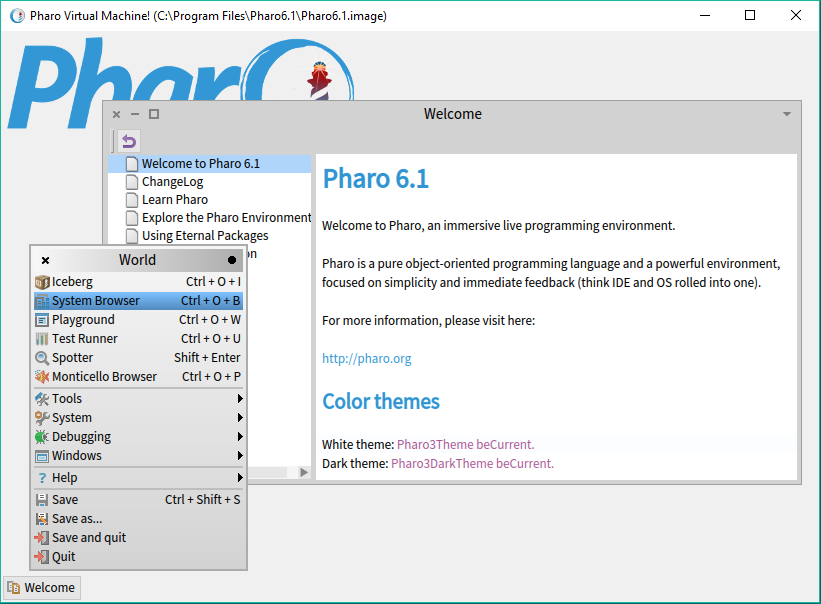


Рис. 2.2 Головне вікно Pharo. Клацання мишею на довільній ділянці вікна  
викликає появу World-меню

Після встановлення та запуску PharoLauncher матиме вигляд, подібний до зображеного на рис. 2.1. Зазвичай він складається з двох стовпців.

* Стовпець «Existing images» відображає перелік образів, що зберігаються локально на вашій машині (зазвичай у спільній системній папці). Ви одразу можете запустити будь-який з них: двічі клацніть на ньому, або позначте і натисніть кнопку «Launch». Контекстне меню, яке викликають правою кнопкою миші, містить декілька корисних команд: копіювання та перейменування образів, відшукання їхнього розташування у файловій системі тощо.
* Стовпець «Templates» містить список образів, які можна завантажити з мережі. Щоб завантажити такий віддалений образ, позначте його в переліку і натисніть кнопку «Create image», або двічі клацніть на ньому.

За допомогою PharoLauncher ви можете запускати власні образи системи, збережені чи перейменовані під час використання Pharo. Для цього помістіть відповідні .*image* та .*changes* файли у призначену папку образів. Її розташування задано в параметрах налаштування PharoLauncher. Переконайтеся, що файл образу та відповідний файл змін мають однакові імена.

## 2.5 Меню «World»

Коли Pharo запуститься, ви мали б побачити одне велике вікно, що, можливо, містить декілька менших: інформаційне, робочу область тощо (див. рис. 2.2). Вигляд вікна трохи незвичний, і не зрозуміло одразу, як з ним взаємодіяти. Відсутнє головне меню аплікації, бо Pharo головно використовує контекстно залежні спадні меню.

Клацання в довільному місці тла головного вікна Pharo відкриває *World-меню*, або *головне меню* системи. Воно містить низку інструментів, утиліт і налаштувань системи. Як і більшість меню Pharo, головне меню не модальне. Ви можете залишати його на екрані, скільки забажаєте, пришпиливши клацанням на піктограмі канцелярської кнопки, що зображена у правому верхньому кутку вікна меню.

На початку World-меню ви побачите декілька головних інструментів Фаро, серед яких є Оглядач класів (System Browser), Робоче вікно (Playground), Менеджер пакетів (Monticello Browser) та інші. Ми розповімо про них докладніше в наступних розділах.

### Взаємодія з Pharo

Smalltalk одразу було спроектовано для комп’ютера, оснащеного мишкою з трьома кнопками. Якщо Ваша мишка має менше ніж три кнопки, вам доведеться натискати додаткові клавіші на клавіатурі, щоб імітувати наявність додаткових кнопок. Мишка з двома кнопками працює у Pharo цілком добре, але якщо ви використовуєте мишку з однією кнопкою, то варто серйозно задуматися про купівлю мишки з колечком прокручування (яким можна клацати), щоб робота з Pharo стала набагато приємнішою.

Середовище Pharo пропонує користувачеві три способи взаємодії за допомогою мишки чи іншого вказівного пристрою.

**Клацнути** (click): натиснути найчастіше вживану кнопку мишки (зазвичай це означає клацнути лівою кнопкою), що для мишки з однією кнопкою означає просте клацання без натискання жодних модифікуючих клавіш. Клацніть, наприклад, у вікні системи, щоб розгорнути головне меню (рис. 2.2).

**Контекстно клацнути** (action-click): натиснути іншу часто вживану кнопку мишки, якою зазвичай розгортають контекстне меню. Таке меню може містити різний перелік команд залежно від місця, на яке вказувала мишка в момент клацання (див. рис. 2.3). Якщо у вашої мишки тільки одна кнопка, вам потрібно буде задати конфігурацію модифікуючих клавіш Ctrl так, щоб клацання мишкою при їх натисканні розгортало контекстне меню.

**Мета-клацнути** (meta-click) – спосіб виконання цієї дії у Pharo залежить від налаштувань вашої операційної системи. Потрібно клацнути колечком прокручування, або мишкою у поєднанні з натисканням модифікуючих клавіш Shift + Alt, Shift + Ctrl (у Windows чи Linux), або з Shift + Option (у OS X). Графічний інтерфейс користувача середовища Pharo збудований з об’єктів бібліотеки Morphic. Всі вікна, написи, меню, які ми бачимо на екрані Pharo, є *морфами*. Мета-клацанням на будь-якому об’єкті, зображеному у вікні образу, викликають його «морфовий ореол» або меню-ореол – набір маніпуляторів, розташованих навколо об’єкта, які використовують для виконання дій із самим екранним об’єктом. Наприклад, для обертання, зміни розміру, кольору тощо (див. рис. 2.4). Якщо ви затримаєте вказівник мишки над маніпулятором, з’явиться спливаюча підказка з поясненням про призначення цього маніпулятора.

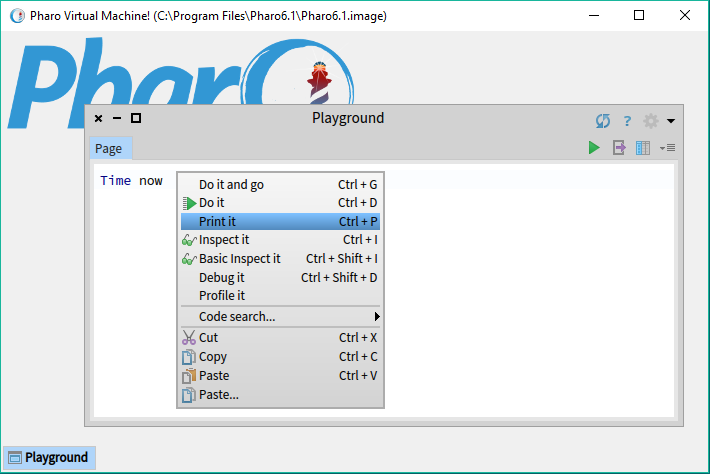


Рис. 2.3 Контекстне клацання (клацання правою кнопкою) розгортає контекстне меню

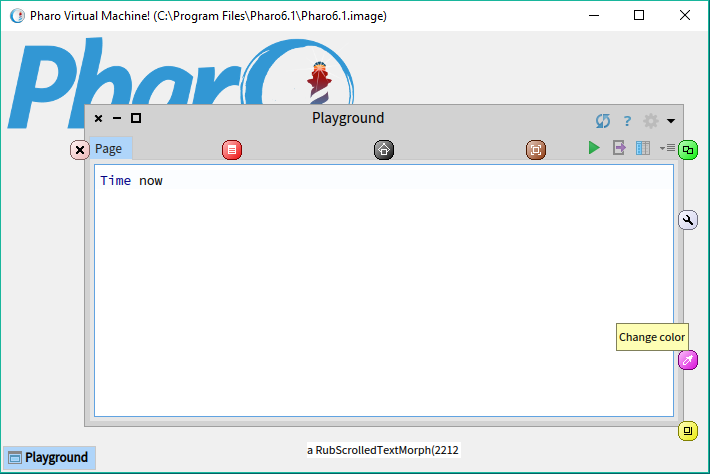


Рис. 2.4 Мета-клацання викликає меню-ореол графічного об’єкта

Ми будемо вважати, що зазвичай мишку сконфігуровано так, що *клацнути* можна лівою кнопкою мишки, *контекстно клацнути* – правою, а *мета-клацнути* – колечком прокручування.

Якщо Pharo працює у вас в операційній системі Windows, ви помітите, що звичайне і контекстне клацання можуть діяти однаково. Наприклад, обома кнопками мишки можна вибрати команду головного меню Pharo, закрити вікно. Мета-клацання можна виконати двома способами. Наприклад, для цілого вікна меню спрацює клацання колечком, а для довільного рядка меню (для довільної видимої частини будь-якого вікна) – клацання кнопкою у поєднанні з *Alt* + *Shift*.

## 2.6 Надсилання повідомлень

Для швидкого виконання фрагментів коду у Pharo використовують спеціальний інструмент – робоче вікно *Playground* – «випробувальний майданчик», який є удосконале­ним аналогом *Workspace* з попередніх версій Pharo. Щоб відкрити його, клацніть у будь-якому вільному місці вікна системи і виберіть з головного меню, що з’явилося, відповідну команду. Як тільки відкриється Робоче вікно, ми можемо вводити і виконувати в ньому довільні вирази мовою Smalltalk. Давайте почнемо з такого прикладу:

CalendarMorph openOn: Date today

Уведіть текст прикладу і контекстно клацніть біля нього, у спадному меню, що з’явилося, виберіть команду «*Do it (Ctrl + D)*» (див. рис. 2.5). У лівому верхньому куті вікна Pharo мало б з’явитися невелике вікно з календарем поточного місяця.

|  |  |
| --- | --- |
| Рис. 2.5 Інтерпретувати вираз можна командою «Do it» | Рис. 2.6 Меню-ореол морфи календаря |

Вітаємо, ви щойно виконали один з ваших перших Smalltalk-виразів! Smalltalk спирається на концепцію надсилання повідомлень об’єктам. Об’єкти Pharo схожі на солдатів, готових одразу виконати ваш наказ, якщо тільки вони його розуміють. У прикладі ви надіслали два повідомлення різним класам. Клас *Date*, отримавши повідомлення *today*, повернув свій екземпляр, що репрезентує поточну дату. Далі клас *CalendarMorph* отримав повідомлення *openOn:* з аргументом – створеним екземпляром класу *Date*.

Клас *Date* сам вирішив, що робити з повідомленням *today*, тобто знайшов і виконав відповідний метод, яким можна опрацювати це повідомлення. Так само інший клас, *CalendarMorph*, знайшов метод для опрацювання повідомлення *openOn:*, використав дані, отримані від аргументу цього повідомлення і побудував вікно з зображенням календаря відповідного місяця. Пізніше ми докладніше обговоримо, як об’єкти розуміють повідомлення.

Спілкуючись з програмістами на Smalltalk, ви швидко помітите, що вони ніколи не використовують таких виразів як «виконати операцію» чи «викликати метод», натомість вони говорять «надіслати повідомлення». Це відображає ідею про те, що об’єкти самі відповідають за власну поведінку, а пов’язані з повідомленням методи вибираються динамічно. Коли надсилають повідомлення, то не адресант, а сам об’єкт вибирає метод, виконанням якого потрібно реагувати на отримане повідомлення. Зазвичай цей метод називається так само, як повідомлення.

З вікном календаря ви можете взаємодіяти як з окремою програмою, наприклад, клацнути на заголовку та вибрати місяць чи рік для відображення. Складніше буде закрити це вікно, бо морфа календаря не містить звичайних кнопок керування. Щоб перемістити чи закрити календар, мета-клацніть на ньому – з’явиться меню-ореол (рис. 2.6), і можна буде скористатися його кнопками.

Попередній приклад досить цікавий, але для вивчення Pharo є кращий. Виконайте в Робочому вікні такий фрагмент:

ProfStef go.

Він запустить простий інтерактивний підручник, з яким ви здійсните захопливу мандрівку системою і вивчите основи Pharo.

Як користувач системи ви не мусите знати, як працює кожне повідомлення. Єдине, що справді потрібно для взаємодії з певним об’єктом, то це перелік повідомлень, на які він може відповісти. Такий підхід приховує складність об’єктів, а програмування залишається на стільки простим, на скільки це взагалі можливо, причому без втрати гнучкості.

Згодом ми розповімо, як отримати протокол взаємодії для будь-якого об’єкта.

## 2.7 Зберігання, завершення та повторний запуск сесії Pharo

Вийти з Pharo можна будь-коли. Для цього просто закрийте його вікно так само, як закриваєте інші застосунки, або використайте команду головного меню: «*Save and quit*» або «*Quit*». У будь-якому випадку Pharo перепитає вас, чи хочете ви зберегти образ системи. Якщо ви збережете образ Pharo, то після наступного запуску побачите його точно в тому стані, в якому завершили роботу. Як таке можливо? Файл образу зберігає об’єкти, які Pharo завантажує у пам’ять віртуальної машини: редагований текст, розташування вікон, додані методи чи класи – адже всі вони є об’єктами. Таким чином при виході з Pharo нічого не втрачається.

Коли ви запускаєте Pharo вперше, VM завантажує вказаний файл образу. Він містить миттєвий знімок багатьох об’єктів, включно з величезним обсягом відкомпільованого коду й інструментами програмування (кожен з яких є об’єктом). Працюючи з Pharo, ви будете надсилати повідомлення цим об’єктам, створюватимете нові об’єкти, деякі з них переставатимуть існувати, а їхня пам’ять буде використана повторно після автоматичного збирання сміття.

Закриваючи Pharo, ви зберігатимете образ усіх своїх об’єктів. У результаті успішного збереження .*image* файл буде перезаписано новим образом системи. Звичайно, ви можете зберегти образ у файлі з новим іменем.

Як ми вже говорили, крім .*image* файла існує .*changes* файл. Він містить журнал усіх змін вихідного коду, зроблених вами за допомогою стандартних інструментів. У більшості випадків ви можете взагалі не турбуватися про цей файл. Проте, як ми скоро побачимо, .*changes* файл може бути дуже корисним для виправлення помилок чи для відновлення втрачених змін. Але про це – згодом!

Може скластися враження, що образ є ключовим механізмом для зберігання програмних проектів і керування ними, але це не так. Незабаром ми познайомимося з набагато кращим засобом для керування кодом та організації командної розробки програмного забезпечення. Образи дуже корисні, але ви звикнете не турбуватися особливо про створення і вилучення образів, оскільки засоби контролю версій, наприклад такі, як *Monticello*, пропонують набагато кращі способи керування версіями та поширення коду програмного забезпечення серед команди розробників. Крім того, якщо вам потрібно зберігати об’єкти, ви можете використати декілька спеціальних систем: *Fuel* (швидкий бінарний серіалізатор об’єктів), *STON* (текстовий серіалізатор об’єктів) або базу даних.

## 2.8 Вікна Playground і Transcript

Давайте виконаємо декілька вправ.

1. Закрийте всі відкриті у Pharo вікна.
2. Відкрийте вікно *Transcript* і вікно *Playground* або *Workspace*. (Transcript можна відкрити за допомогою підменю *World* > *Tools* > ... .)
3. Розташуйте вікна так, щоб Робоче вікно було поверх Transcript, але видно було обидва (див. рис. 2.7).

Ви можете змінювати розмір вікна за допомогою перетягування одного з його кутів. З усіх відкритих вікон лише одне активне, воно розташоване поверх усіх інших.

*Transcript* – це об’єкт, який часто використовують для зберігання журналу системних повідомлень. Це свого роду консоль системи.

Робоче вікно (чи вікна) зручне для введення і випробування фрагментів коду. Ви також можете використовувати його для звичайного введення тексту, який би ви хотіли запам’ятати. Наприклад, для створення списків «до виконання» або інструкцій для будь-кого, хто використовуватиме ваш образ системи.

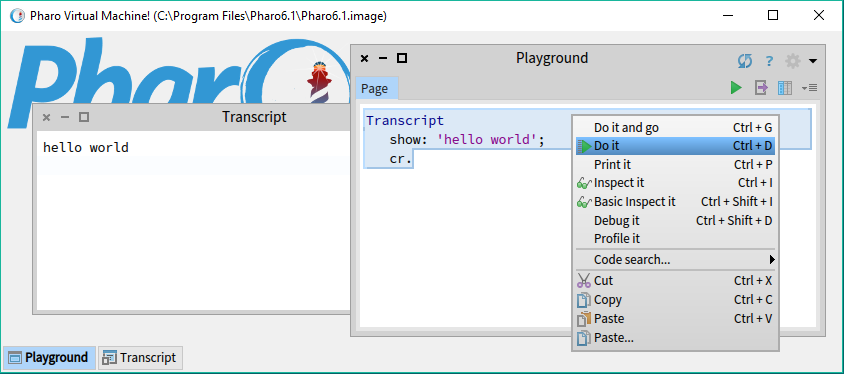


Рис. 2.7 Виконання виразу відображає рядок тексту в Transcript

Уведіть у Робочому вікні такий текст:

Transcript

show: 'hello world';

cr.

Випробуйте подвійне клацання в різних місцях щойно надрукованого тексту. Позначено буде слово, цілий рядок або весь текст залежно від позиції клацання: в межах слова, в кінці рядка, чи в кінці виразу. Зокрема, якщо ви встановите курсор перед першою літерою виразу або після останньої і двічі клацнете, то позначено буде весь абзац.

Позначте весь текст у Робочому вікні, контекстно клацніть і виберіть «*Do it*». Зверніть увагу, як привітання “hello world” з’явиться у вікні Transcript (див. рис. 2.7). Виконайте вираз ще раз.

## 2.9 Гарячі клавіші

Щоб виконати вираз, ви не мусите щоразу контекстно клацати. Натомість ви можете використати комбінації гарячих клавіш, вказані в пунктах меню. Попри те, що Pharo схоже на середовище, кероване мишею, воно пам’ятає більше двохсот комбінацій клавіш для взаємодії з різноманітними інструментами та надає користувачеві можливість призначити нову комбінацію для будь-якого з 80000 методів, записаних у образі Pharo. Для того, щоб переглянути актуальні комбінації клавіш, використайте меню *World* > *System* > *Keymap* *Browser*.

Залежно від вашої операційної системи, до складу комбінації клавіш входить одна з модифікуючих клавіш: *Control*, *Alt*, або *Command*. Надалі в книзі ми будемо позначати її *CMD*, тому, коли ви прочитаєте щось на зразок «*CMD-d*», просто замініть наше позначення відповідною до вашої операційної системи клавішею. Наприклад, у Windows це буде означати «*Ctrl-d*».

Ви мали б помітити в контекстному меню крім команди «*Do it*» також «*Do it and go*», «*Print it*», «*Inspect it*» та декілька інших. Поглянемо на кожну з них.

### Виконання VS виведення

Надрукуйте вираз *3*+ *4* у Робочому вікні і застосуйте до нього «*Do it*» за допомогою комбінації клавіш.

Не дивуйтеся, що нічого не відбулося! Ви надіслали повідомлення «+» з аргументом *4* об’єкту *3*. У результаті виконання відповідного методу було отримано результат, число *7*, яке повернулося до Робочого вікна, але воно не знало, що з ним робити, й тому нічого не показало. Якщо ви хочете бачити результат, то використовуйте замість «*Do it*» команду «*Print it*». Під час її виконання вираз компілюється, компільований код виконується, отриманому результату надсилається повідомлення *printString*, а вікно відображає результуючий рядок.

Позначте *3*+ *4* і виберіть «*Print it*» (*CMD-p*). Цього разу ми побачимо результат, на який сподівалися:

3 + 4

>>> 7

Ми домовилися використовувати в цій книзі позначення «>>>», щоби вказувати на результат, який продукує конкретний Smalltalk-вираз, якщо виконати «*Print it*».

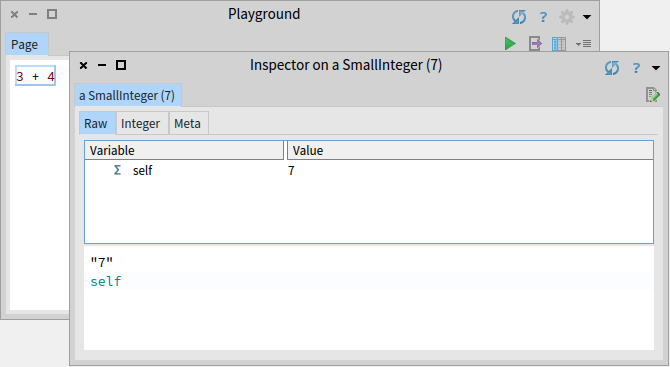


Рис. 2.8 Інспектування звичайного числа за допомогою «*Inspect it*»

### Інспектування

Позначте вираз *3*+ *4* або помістіть курсор у рядок з ним і виберіть команду контекстного меню «*Inspect it*» (*CMD*-*i*).

Мало би відкритися нове вікно із заголовком «*Inspector on a SmallInteger (7)*», як показано на рис. 2.8. Інспектор є надзвичайно корисним інструментом, що дає вам змогу переглядати будь-який об’єкт в системі та взаємодіяти з ним. Заголовок вікна повідомляє нам, що *7* є екземпляром класу *SmallInteger*. Верхня панель відображає змінні екземпляра та їхні значення (у числа це єдина змінна *self*). Нижню панель можна використати для надсилання повідомлень екземплярові. Надрукуйте в ній «*self squared*» і виберіть «*Print it*». Результат виконання виразу з’явиться одразу в нижній панелі інспектора.

Інспектор містить специфічні сторінки для відображення різноманітної інформації про екземпляр та його виглядів, залежно від типу об’єкта, який ви інспектуєте. Спробуйте викликати інспектора для фрагменту «*Morph new openInWorld*». Ви мали б отримати щось схоже до зображеного на рис. 2.9.

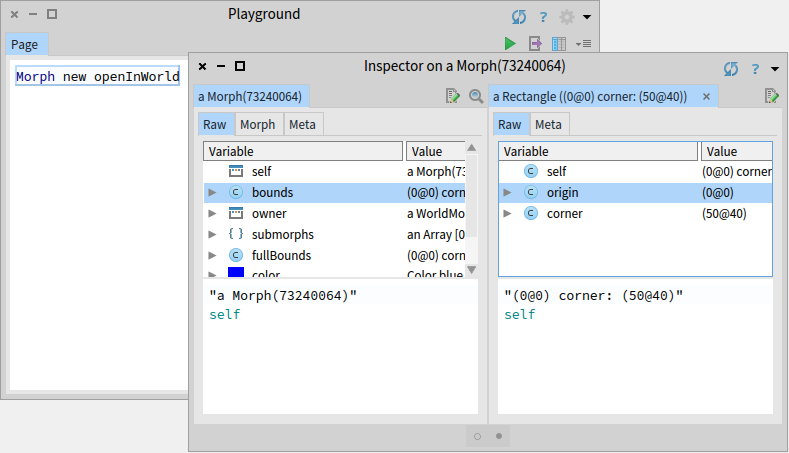


Рис. 2.9 Інспектування морфи за допомогою «*Inspect it*»

Інспектор можна використати для мандрівки структурою об’єкта. Для того, щоб дослідити будь-яку змінну екземпляра, достатньо просто клацнути на ній, і Інспектор автоматично відкриє нову вкладку та відобразить її вміст.

Побічний ефект від виконаної вправи – синій прямокутник у лівому верхньому куті вікна Pharo. Це створена нами морфа, яку ми інспектуємо. Для того, щоб закрити її, потрібно використати меню-ореол.

### Інші дії

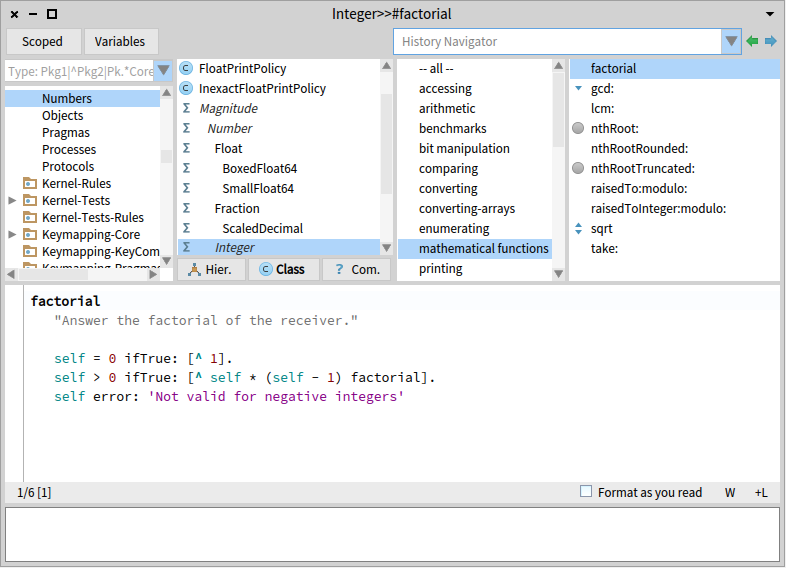
Інші команди контекстного меню:

* «*Do it and go*» додатково відкриває у Робочому вікні сторінку інспектора. Ви можете використовувати її так само, як звичайний Інспектор для дослідження структури об’єкта. Випробуйте з цією командою попередній вираз «*Morph new openInWorld*».
* «*Basic Inspect it*» відкриває класичний інспектор, який надає набагато скромніший графічний інтерфейс користувача але дозволяє змінювати значення змінних інспектованого об’єкта. Виконайте команду для «*Morph new openInWorld*» і в нижній панелі класичного інспектора надрукуйте, наприклад, такий код: «*self color: Color red*» та виконайте його.
* «*Debug it*» відкриває Налагоджувач коду.
* «*Profile it*» будує часовий профіль коду. Він показує скільки часу займає надсилання кожного повідомлення.
* «*Code search*» пропонує декілька способів пошуку програмного коду, реалізованих в оглядачі класів. Серед них пошук певного виразу серед усіх методів Pharo, пошук відправників повідомлення та об’єктів, здатних відповісти на нього, тощо.

## 2.10 Оглядач класів

Системний оглядач (*System Browser*), відомий також як «оглядач класів», є одним з головних інструментів для програмування. Ми побачимо згодом, що з Pharo функціонує декілька цікавих оглядачів, проте *System Browser* є основним, його ви знайдете в кожному образі системи. Поточна реалізація оглядача класів називається Nautilus (так Жуль Верн назвав підводного човна, описаного в романі «1000 льє під водою»).

Оглядача легко викликати за допомогою меню *World* > *System Browser*, або комбіна­цією клавіш *CMD-o,b*, або програмно, виконавши вираз «*Nautilus open*». У вікні оглядача ви побачите спочатку лише перелік пакетів класів. Відкрити можна будь-який з них простим клацанням, але пошук потрібного класу чи методу серед тисяч інших справа невдячна, тому поговоримо про швидші способи.



Відгук від Quality Assistant

Тіло методу

Коментар методу

Селектор методу

Фільтр пакетів

Пакети

Методи

Протоколи

Класи

Рис. 2.10 Оглядач класів показує метод *factorial* класу *Integer*

### Відкривання Оглядача на заданому класі чи методі

Ви можете переглянути код будь-якого класу, якщо відкриєте його в System Browser за допомогою виразу «*Nautilus openOnClass: ClassName*», де *ClassName* – ім’я класу, що вас цікавить. Ще більш конкретне завдання – це відкривання в Оглядачі класів певного методу певного класу. Його можна виконати виразом «*Nautilus openOnMethod: methodName*».

Зазвичай для відкривання методу ми використовуємо ще кращі засоби ніж Оглядач класів, але зараз, щоб презентувати його вигляд, відкриємо його на конкретному методі. Для цього виконаємо такий фрагмент:

Nautilus openOnMethod: Integer>>#factorial

Він відкриє Оглядач класів на методі *factorial*. Ми мали б отримати вікно System Browser як на рис. 2.10. Заголовок вікна вказує, що ми переглядаємо клас *Integer* і його метод *factorial*. На рисунку показано різні сутності, які відображає Оглядач: пакети, класи, протоколи, методи та визначення методу.

Зауважимо, що у Pharo запис вигляду *Клас*>>*метод* – це звичайний спосіб ідентифі­кувати метод певного класу.

За замовчуванням у Pharo системним оглядачем є Nautilus. Проте в середовищі можна інсталювати й інші оглядачі, наприклад, *AltBrowser*. Кожен системний оглядач може мати свій графічний інтерфейс, що відрізнятиметься від інтерфейсу Nautilus’а. Надалі ми вживатимемо терміни *Оглядач класів*, *Системний оглядач* (*System Browser*) чи, іноді, *Nautilus* як синоніми.

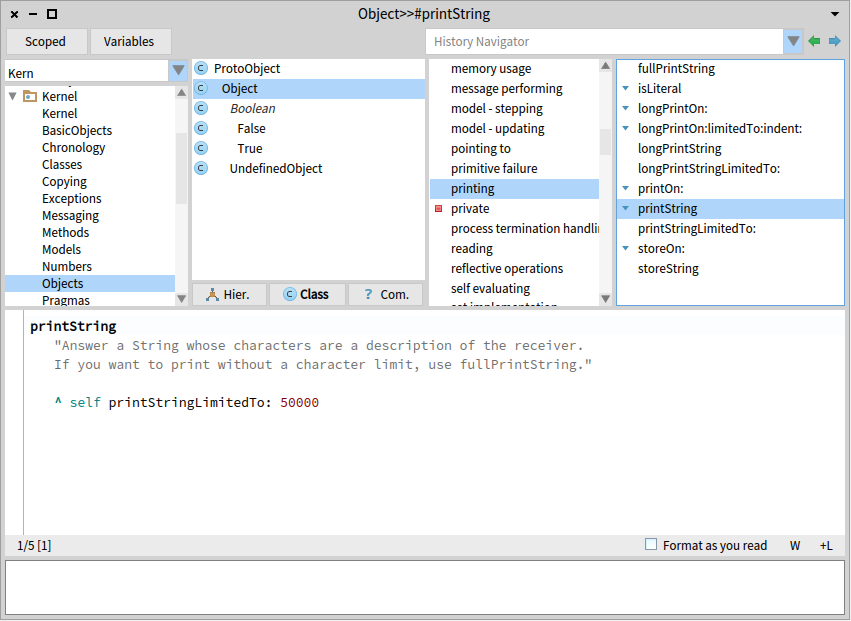


Рис. 2.11 Оглядач класів показує метод *printString* класу *Object*

### Мандрівка системою за допомогою Оглядача класів

Для відшукання потрібного коду в системі Pharo є *Spotter* (описаний далі). З його допомогою ми можемо перейти безпосередньо до класу чи методу. Але зараз ми хочемо показати вам послідовність кроків для пошуку за допомогою Оглядача класів. Давайте розглянемо, як знайти метод *printString*, визначений в класі *Object*. Наприкінці пошуку Оглядач виглядатиме, як зображено на рис. 2.11.

**Відкрийте Оглядач класів за допомогою World > System Browser.** Коли нове вікно Оглядача відкриється, всі панелі крім крайньої лівої будуть порожніми. Вона містить перелік усіх відомих пакетів, які містять групи пов’язаних класів.

**Відфільтруйте пакети.** Надрукуйте частину імені пакета в рядку фільтра над лівою панеллю. Він відбирає пакети для відображення у ній. Надрукуйте, наприклад, «*Kern*».

**Розгорніть пакет *Kernel* і виберіть елемент *Objects*.** Якщо вибрати пакет, то друга панель відобразить список усіх класів, що входять до цього пакета. Ви мали б побачити ієрархію класу *ProtoObject*.

**Виберіть клас *Object*.** Якщо вибрати клас, дві панелі, що залишилися, заповняться даними. Третя панель відображає протоколи вибраного класу. Вони виконують зручний поділ усіх методів на групи чи категорії пов’язаних між собою (функціонально чи за призначенням). Якщо жодного протоколу не вибрано, четверта панель відображає список усіх методів.

**Виберіть протокол *printing*.** Вам доведеться прокрутити список протоколів, щоб знайти його. Ви також можете клацнути на панелі протоколів і почати друкувати «*pr*» для автоматичного пошуку пунктів списку, що починаються цими літерами. Урешті виберіть *printing*, і ви побачите в четвертій панелі лише ті методи, які стосуються цього протоколу.

**Виберіть метод *printString*.** Тепер ми бачимо в нижній панелі вихідний код методу *printString*. Його поділяють усі об’єкти системи (крім тих, що його перевизначають).

А тепер ми розглянемо набагато кращий спосіб відшукання методів.

## 2.11 Відшукання класів

Є декілька способів відшукання класів у Pharo. Перший з них, як ми щойно побачили, використовує Системний оглядач для переходу до визначення класу. Для цього потрібно знати (або вгадати), до якого пакету належить клас.

Другий спосіб полягає в тому, щоб надіслати класові повідомлення *browse* – попро­сити його відкрити Оглядача на собі. Припустимо, що ми хочемо переглянути клас *Point*.

|  |  |
| --- | --- |
| Рис. 2.12 Пошук у *Spotter* | Рис. 2.13 Пошук реалізацій *printString* |

### Використання повідомлення *browse*

Надрукуйте «*Point browse*» у Робочому вікні і виберіть «Do it». Оглядач відкриється на класі *Point*.

### Використання *CMD-b* для перегляду

У будь-якому текстовому вікні можна використовувати комбінацію *CMD-b* (browse), щоб викликати Оглядач. Для цього просто позначають слово і натискають *CMD-b*. Випробуйте дію комбінації на слові *Point*.

Зверніть увагу на те, що коли в Оглядачі класів позначено клас *Point*, але не вибрано ні протоколу, ні методу, то в нижній панелі замість визначення методу ми бачимо визначення класу. Це не що інше, як звичайне повідомлення до батьківського класу з проханням створити підклас. Тут ми бачимо, що клас *Object* попросили створити підклас, що називається *Point*, має дві змінні екземпляра, не має змінних класу і належить до пакету *Kernel-BasicObjects*. Якщо клацнути на кнопці *Comments* (? Com.) внизу панелі класів, то відкриється спеціалізована панель з описом класу.

Додатково система підтримує такі комбінації клавіш з мишкою:

* *CMD-Click* на слові. Якщо слово – ім’я класу, то відкриється визначення класу; якщо слово – селектор повідомлення, записаний в тілі методу, то відкриється список класів, що реалізують відповідні методи.
* *CMD-Shift-Click* на слові. Якщо слово – ім’я класу, то відкриється список посилань на нього; якщо слово – селектор повідомлення, записаний в тілі методу, то відкриється список відправників цього повідомлення

***Зауваження від перекладача*.** Для версії Pharo 6.1 в ОС Windows 10 «*CMD-Click*» означає «*Ctrl-Alt-Click*», комбінація «*Ctrl-Click*» викликає контекстне меню, у якому можна вибрати «*Code search…*» і таким чином знайти визначення, відправників, реалізації, посилання тощо. «*Ctrl-Shift-Click*» викликає контекстне меню морфи.

### Використання Сищика (*Spotter*)

Найшвидший і, можливо, найкрутіший спосіб відшукати клас – використати *Spotter*. Натискання *Shift-Enter* відкриває Сищика (*Spotter*), дуже потужний інструмент для відшукання класів, методів, і виконання багатьох пов’язаних дій. У першому рядку вікна можна ввести шукане слово чи його частину. На рисунку 2.12 показано результати пошуку слова «*point*».

Сищик пропонує декілька можливих варіантів пошуку. Ви можете задати категорію, яка вас цікавить. Наприклад, для пошуку виключно класів уведіть «*#class*» перед шуканим словом. Якщо не вказувати категорію, Сищик знайде всі (рис. 2.12).

Рисунок 2.13 показує, як ми можемо попросити Spotter показати нам усі класи, що реалізують задане повідомлення. Як видно, назву категорії можна вводити не повністю. Інші категорії такі: меню (*#menu*), пакети класів (*#package*), методи (*#implementor*), прик­лади (*#example*), прагми (*#pragma*), адресанти (*#sender*), посилання на клас (*#reference*) і навіть фрагменти коду, введені у Робочому вікні (*#playground*). Щоб задати категорію, ви можете просто надрукувати її початок, наприклад, «*#ref Point*» дасть усі посилання на клас *Point*.

Сищика можна використати навіть для перегляду файлової системи. Він також містить категорію «історія» (*#history*), яка зберігає знайдені раніше результати, щоб можна було швидко переглянути їх ще раз.

### Перехід за результатами пошуку

Ми можемо використовувати Сищика для переходів за результатами пошуку подібно до того, як ми використовували Оглядач класів. Результати пошуку в Сищику поміщено в список і поділено за категоріями, наприклад, класи віднесено до категорії *Classes*, методи віднесено до *Implementors*, розділи довідки – до *Help Topics* тощо.

Кожен рядок списку (пункт) містить стрілку біля свого правого краю. Клацання на ній створить нову вкладку у вікні Сищика з ярликом у заголовку. Вона відобразить нам зміст вибраного пункту. Залежно від того, що ми вибрали, нова вкладка може бути поділена на нові категорії. Клацання на ярликах вкладок дає змогу легко повернутися до того місця, звідки ми починали пошук.

Наприклад, якщо вибрати клас *Point*, то ми потрапимо до групи категорій методів: методи екземпляра, методи класу, методи надкласу тощо.

Сищиком можна керувати за допомогою клавіатури. Користувач може перебирати пункти списку за допомогою клавіш зі стрілками *Вгору*/*Вниз*, переходити з вкладки на вкладку за допомогою комбінацій *CMD-Ліворуч*/*CMD-Праворуч*. Водночас рядок пошуку утримує фокус уведення, тому користувач без зусиль може переходити від вибору пунктів списку до введення нового тексту для пошуку. Натискання клавіші *Enter* на вибраному пункті відкриє оглядач класів (а Spotter закриє) на конкретному вибраному результаті пошуку. Пошук завжди можна завершити клавішею *Esc*.

### Використання команди «*Find class*» в Системному оглядачі

Ви також можете шукати клас у Системному оглядачі за його іменем. Припустимо, що ви, наприклад, шукаєте невідомий клас, який моделює дату і час.

Відкрийте Оглядача, а тоді – вікно пошуку класів за допомогою комбінації *CMD-f,c* або за допомогою команди «*Find Class…*» контекстного меню панелі пакетів чи панелі класів. Далі в рядку введення надрукуйте «*time*». Вікно відобразить список класів, до імен яких входить підрядок «*time*» (без огляду на регістр). Тепер ви можете вибрати один з них. Для цього мишею прокрутіть список до потрібного рядка, наприклад, *Time* і клацніть на ньому. Те саме можна зробити за допомогою клавіатури: після введення пошукового слова натисніть *Enter* і клавішами зі стрілками знайдіть потрібний рядок.

Щоб відкрити вибраний клас, достатньо натиснути клавішу *Enter*, або клацнути на кнопці *OK*, чи кнопці *Browse*.

Зауважте, що якщо правильно ввести повне ім’я класу з урахуванням регістра, то після натискання *Enter* Оглядач класів пропустить крок вибору і відразу покаже цей клас.

### Використання засобу пошуку *Finder*

Ви також можете використати Пошуковця (*Finder*), відкривши його командою меню *World* > *Tools*... Надрукуйте в рядку введення частину імені класу та змініть у спадному списку праворуч «*Selectors*» на «*Classes*». Пошуковець не такий ефективний, як Сищик чи Оглядач класів. Більша користь від нього для інших видів пошуку коду, таких, як пошук методів за зразками, про що ми розповімо згодом.

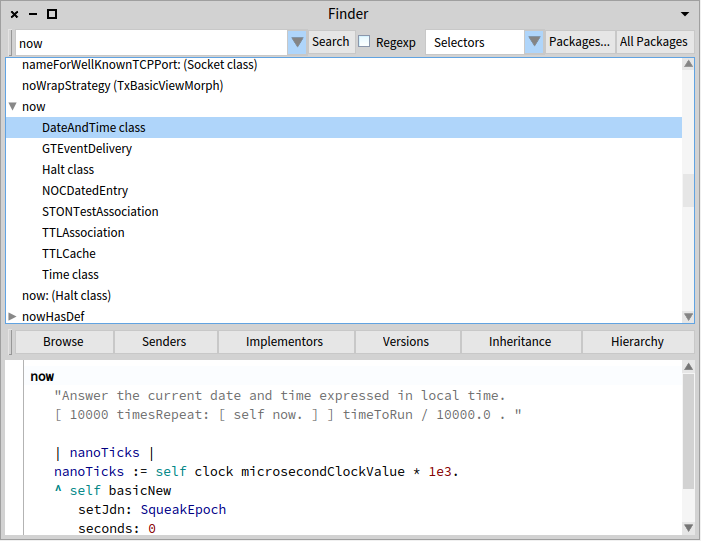


Рис. 2.14 Пошуковець показує всі класи, що реалізують метод *now*

## 2.12 Відшукання методів

Іноді ім’я методу чи його частини можна вгадати легше, ніж ім’я класу. Наприклад, якщо вас цікавить поточний час, то ви могли б сподіватися, що метод називатиметься «*now*» або міститиме «*now*» як підрядок. Але де він міг би бути? Сищик і Пошуковець можуть вам допомогти.

### Spotter

За допомогою Сищика ви також можете знаходити методи. Або знайти клас і перейти до його методів, або використати такі категорії:

* *#implementor methodName* відобразить усі наявні методи з іменем *methodName*, наприклад, ви можете отримати всі методи «*do:*»;
* *#selector methodName* відобразить усі селектори з таким іменем.

### Finder

Виберіть *World Menu* > *Tools* > *Finder*. Уведіть «*now*» в рядок пошуку і клацніть на кнопці *Search* (або натисніть клавішу *Enter*). Ви мали б побачити список результатів, як на рисунку 2.14.

Пошуковець відобразить список усіх імен методів, що містять підрядок «*now*». Щоб швидко прокрутити його власне до методу «*now*», перемістіть фокус уведення до списку і натисніть «*n*». Такий автоматичний пошук працює у всіх вікнах з прокручуванням. Розгорніть пункт «*now*», і ви побачите перелік усіх класів, що реалізують цей метод. Якщо вибрати котрийсь з них, то панель коду внизу вікна відобразить текст методу.

### Пошук методів за зразками

Часом трапляється так, що ви впевнені в існуванні методу, але гадки не маєте, як би він міг називатися. Пошуковець і тоді може допомогти! Припустимо, що ви хотіли б знайти метод, який переводить рядок літер до верхнього регістру (наприклад, перетворює ’*eureka*’ на ’*EUREKA*’). Ми можемо надати Пошуковцю вхідні дані та очікуваний результат виконан­ня методу, і він спробує знайти для нас такий метод.

Пошуковець володіє справді потужною функціональністю: ви можете задати отримувача, аргументи й очікуваний результат, а він намагатиметься підібрати відповідне повідомлення.

Виберіть у Пошуковці режим «*Examples*» у спадному списку, що за замовчуванням показує режим «*Selectors*». Тепер у рядку пошуку можна вводити зразки у вигляді «*отримувач . [аргумент(и) . ]результат*».

Введіть «*'eureka' . 'EUREKA'*» в рядок пошуку і натисніть клавішу *Enter*. Тоді Пошуко­вець запропонує метод, який робить те, що вам потрібно, та покаже список класів, які реалізують метод з таким іменем. У нашому випадку він визначить, що метод «*asUppercase*» є одним з тих, чия дія збігається зі зразком.

Клацніть на трикутнику ліворуч від виразу *'eureka' asUppercase --> 'EUREKA'*, щоб розгорнути список класів, які реалізують цей метод.

Зірочка на початку рядка списку класів позначає той метод, який застосовують для отримання потрібного результату. Тому зірочка перед класом *String* повідомляє нам, що метод *asUppercase*, визначений в класі *String*, після виконання поверне той результат, який нам потрібно. Класи без зірочки також реалізують метод з іменем *asUppercase*, але резуль­тат вони повертають інший. Так метод *Character*>>*asUppercase* не спрацював у нашому прикладі, оскільки *'eureka'* не є екземпляром класу *Character* (це екземпляр класу *String*).

За допомогою Пошуковця ви можете також знаходити методи з одним чи кількома аргументами. Наприклад, якщо вам потрібно знайти метод, який обчислює найбільший спільний дільник двох натуральних чисел, ви можете спробувати пошукати за зразком «*25 . 35 . 5*». Ви можете також надати декілька зразків, щоб звузити область пошуку. Довідковий текст у нижній панелі вікна пояснює, як це зробити.

## 2.13 Визначення нового методу

Поява технології розробки, керованої тестами, Test Driven Development (TDD) зміни­ла спосіб написання коду. Головна ідея TDD полягає в тому, що тест, який визначає бажану поведінку коду, пишуть перш ніж написати сам код. Поведінку класу (чи методу) описують у термінах тестів, і лише тоді пишуть клас, що задовольняє ці тести.

Припустимо, що ми маємо завдання написати метод, який «говорить щось голосно і виразно». Що це насправді означає? Як правильно назвати такий метод? Як ми можемо впевнитися в тому, що програмісти, які супроводжуватимуть наш метод у майбутньому, матимуть однозначне визначення того, що він робить? Ми можемо відповісти на всі ці запитання на прикладі.

Ми маємо на меті визначити в класі *String* новий метод, що називається *shout* (кри­чати). Це повідомлення мало б перетворювати рядок до великих літер і дописувати знак оклику, як показано нижче.

'No panic' shout

>>> 'NO PANIC!'

Проте, перед написанням самого методу ми повинні написати тестовий метод! У на­ступному параграфі ми використаємо приклад «*No Panic*» для створення тестового методу.

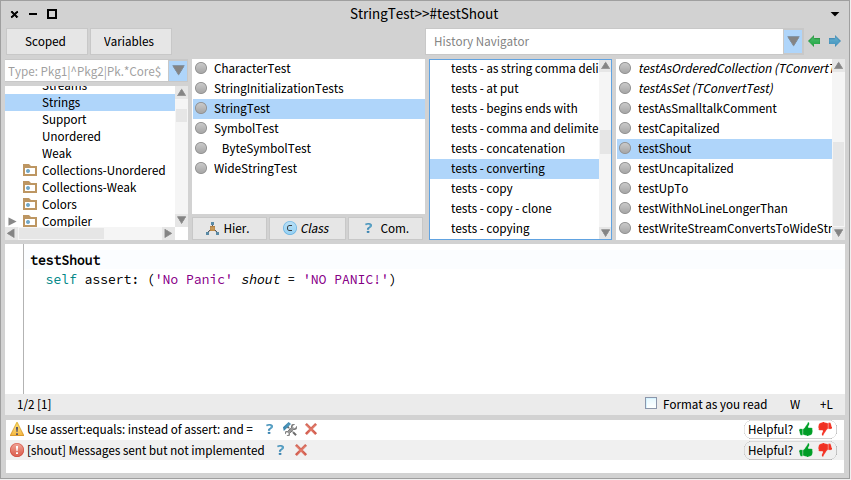


Рис. 2.15 Визначення методу в класі *StringTest*

### Визначення нового тестового методу

Як можна створити новий метод у Pharo? Спочатку потрібно вирішити, до якого класу він належатиме. У нашому випадку метод *shout*, який ми зібралися перевіряти, працюватиме в класі *String*, тому відповідний тест за домовленістю працюватиме в класі *StringTest*.

Спочатку відкрийте Оглядача на класі *StringTest* і відкрийте протокол, що відповідає методові. У нашому випадку це «*tests - converting*». Нижня панель Оглядача містить шаблон визначення методу Pharo. Він нагадує нам, з яких частин воно мало б складатися. Вилучіть текст шаблону (нагадуємо, ви можете двічі клацнути на початку чи в кінці тексту або натиснути *CMD-a*, щоб позначити весь текст) і наберіть тестовий метод: ми можемо зробити його з прикладу перетворення рядка ’*No Panic*’:

testShout

self assert: ('No panic' shout = 'NO PANIC!')

Зауважте, як тільки ви почнете набирати текст, правий верхній кут панелі стане оранжевим. Це нагадування про те, що панель містить не збережені зміни. Коли наберете весь метод, виберіть «*Accept*» з контекстного меню панелі або натисніть *CMD-s*, щоб відкомпілювати і зберегти свій метод. Ви мали б побачити щось схоже, до зображеного на рисунку 2.15.

Коли ви вперше зберігаєте якийсь код у вашому образі системи, середовище швидше всього попросить вас повідомити своє ім’я. Оскільки багато людей додають код до образу, важливо зберігати інформацію про кожного, хто додає чи змінює методи. Просто введіть ваше ім’я та прізвище без жодних пропусків.

Метод *shout* поки що не існує, тому автоматична перевірка коду (Помічник з якості) повідомить вас у нижній панелі Оглядача класів, що ви надіслали повідомлення, яке ще не реалізоване. Це надзвичайно корисно, якщо ви просто зробили помилку введення: можна відразу її виправити. Але в нашому випадку ми справді маємо на увазі «shout», оскільки це метод, який ми збираємося створити.

Помічник також радить використовувати для перевірки рівності отриманого результату й еталону більш доцільний метод «*assert:equals:*».

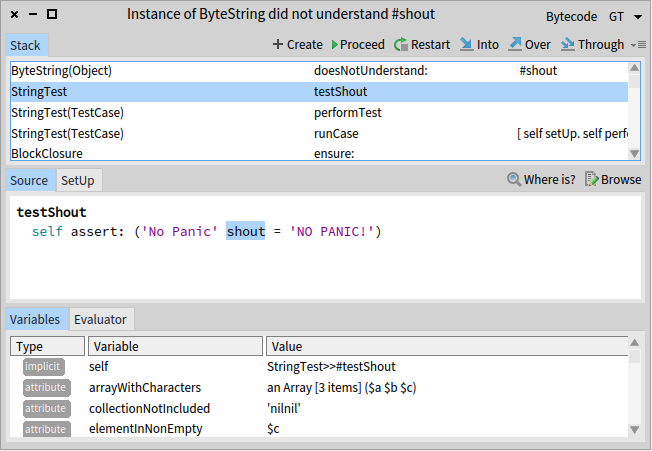


Рис. 2.16 Перегляд помилки в Налагоджувачі

### Виконання тестового методу

Найшвидший спосіб запустити на виконання тестовий метод – клацнути на круглій піктограмі ліворуч від його імені. Але ми підемо довшим шляхом: запустимо Тестувальника командою *World Menu* > *Test Runner*. Дві ліві панелі Тестувальника дуже схожі до таких панелей Оглядача класів. Перша з них містить список пакетів, але тільки тих, що містять класи тестів, друга – список класів вибраного пакета.

Виберіть пакет «*Collections-Tests-Strings*», тоді друга панель відобразить усі його класи тестів, включно з класом *StringTest*. Імена класів уже позначені, тому, щоб запустити тести на виконання, достатньо натиснути кнопку «*Run Selected*».

Ви мали б побачити, що права верхня панель стане червоною. Це означає, що деякі тести завершилися з помилкою. Їхній перелік відобразиться у правій нижній панелі. Як можна бачити, винуватцем є метод *StringTest*>>*test-Shout*. Клацніть на ньому, і помилковий тест запуститься знову, але так, що ви побачите, яка саме помилка сталася: «*Instance of ByteString did not understand #shout*» (див. рис. 2.16). Це відкрилося вікно Налагоджувача Pharo з повідомленням про помилку.

Ми опишемо Налагоджувач та способи його використання в розділі «Середовище Pharo».

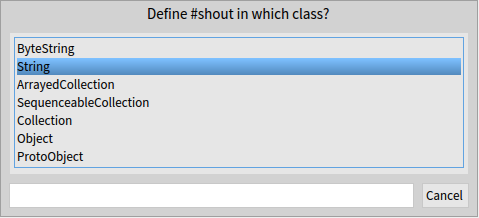


Рис. 2.17 Налагоджувач просить вибрати клас для створення методу

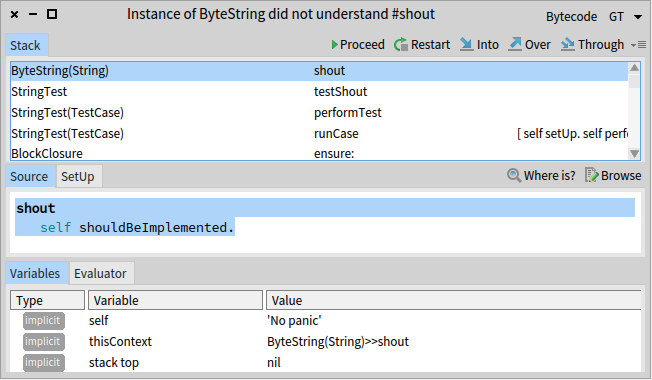


Рис. 2.18 Автоматично згенероване визначення методу *shout* треба замінити справжнім

### Створення тестованого методу в Налагоджувачі

Помилка – це саме те, на що ми сподівалися: виконання тесту спричиняє помилку, бо ми ще не визначили методу, який би навчив рядок кричати. Тим не менше, ми вчинили правильно. Потрібно було переконатися, що тест завершиться невдачею, оскільки це підтверджує, що механізм тестування налаштовано правильно, і новий тест справді працює. Коли ви побачили помилку, ви можете натиснути кнопку «*Abandon*» і перервати виконання тесту, що закриє вікно Налагоджувача.

Замість того, щоб тиснути «*Abandon*», ви можете визначити відсутній метод за допомогою кнопки «*Create*» безпосередньо в самому Налагоджувачі. Він перепитає вас, у якому класі ви хочете визначити новий метод (див. рис. 2.17), тоді запропонує вибрати протокол для методу (у нашому випадку це протокол *converting*), і врешті відкриє вам вікно редагування коду, в якому ви зможете ввести оголошення цього методу. Зауважте, що система вже згенерувала загальне оголошення методу, яке позначає його як такий, що має бути визначений (див. рис. 2.18). Нічого кращого вона зробити не може, бо не знає, чого ви очікуєте від нового методу. Якщо забути про визначення, то виконання згенерованого системою методу відкриє вікно Налагоджувача з відповідним повідомленням про помилку.

Давайте тепер визначимо метод, який успішно пройде тест! Безпосередньо у вікні редагування Налагоджувача введемо визначення методу *shout*:

shout

^ self asUppercase,'!'

Тут кома позначає конкатенацію рядків. Таким чином тіло методу дописує знак оклику до переписаного великими літерами рядка, того екземпляра класу *String*, якому надіслали повідомлення *shout*.

Знак «^» повідомляє компіляторові, що як результат з методу має повернутися значення виразу, записаного слідом. У нашому випадку це – новий рядок, результат конкатенації.

Коли ви введете весь код методу, не забудьте відкомпілювати його за допомогою *CMD-s*. Далі ви можете натиснути кнопку «*Proceed*» і продовжити виконання тестів. Зверніть увагу на те, що «*Proceed*» не перезапускає провалений тест, а продовжує виконання решти тестів набору.

Чи працює створений метод? Давайте запустимо тести і побачимо. Знову натисніть «*Run Selected*» у Тестувальнику. Цього разу ви мали б побачити зелену панель з повідом­ленням про те, що всі тести пройшли без відмов і без помилок. Коли ви дійшли до зеленої панелі, то найкраще, що можна зробити – зберегти результати своєї праці, зберігши образ системи (*World Menu* > *Save*), і зробити перерву. То ж зробимо це просто зараз!

## 2.14 Підсумки розділу

Цей розділ увів вас у середовище Pharo і показав, як користуватися деякими з його головних засобів, таких як Оглядач класів, Сищик, Пошуковець, Налагоджувач і Тестуваль­ник. Ви також трішки познайомилися з синтаксисом Smalltalk, хоча й не розумієте його ще до кінця.

* Працююча система Фаро складається з віртуальної машини, .*sources* файла, .*image* та .*changes* файлів. Лише два останні файли змінюються, коли записують миттєвий знімок стану системи.
* Коли ви відкриваєте образ Фаро, ви отримуєте середовище в тому ж стані (тобто з тими ж працюючими об’єктами), у якому ви востаннє зберігали образ.
* Головне меню системи викликають клацанням на тлі вікна Фаро. Через нього запускають різноманітні засоби системи.
* Робоче вікно Playground – це місце для написання і випробування фрагментів коду. Його можна використовувати також для зберігання довільного тексту.
* Для взаємодії з середовищем зручно використовувати гарячі клавіші, наприклад, для виконання коду в Playground чи іншому засобі. Найважливіші комбінації клавіш *Do it* (*CMD-d*), *Print it* (*CMD-p*), *Inspect it* (*CMD-i*) та *Browse it* (*CMD-b*).
* Оглядач класів є головним засобом перегляду Фаро та написання нового коду.
* Тестувальник є засобом виконання модульних тестів і помічником для керованої тестами розробки коду (для TDD).
* Налагоджувач дає змогу перевіряти помилки та винятки (як помилки чи відмови при виконанні модульних тестів). Ви навіть можете створювати нові методи безпосередньо в Налагоджувачі.

# Розділ 3. Перший застосунок

У цьому розділі ми створимо просту гру Lights Out ([https://en.wikipedia.org/wiki/Lights\_ Out\_(game)](https://en.wikipedia.org/wiki/Lights_%20Out_(game))). По ходу роботи ми покажемо більшість інструментів, якими користуються Фаро-програмісти для написання та налагодження своїх програм, і покажемо, як поділитися програмою з іншими розробникам. Ми використаємо Оглядач класів, Інспектор об’єктів, Налагоджувач та Менеджер 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 екземпляра та методи на льоту, писати код у Налагоджувачі, використо­вуючи точний контекст об’єктів, що існують на поточний момент. У цьому розділі ми розглянемо такий альтернативний спосіб та покажемо, як збільшити вашу продуктивність.

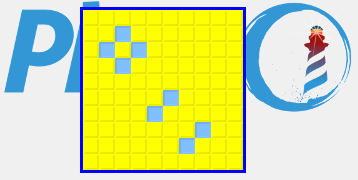


Рис. 3.1 Поле гри Lights Out

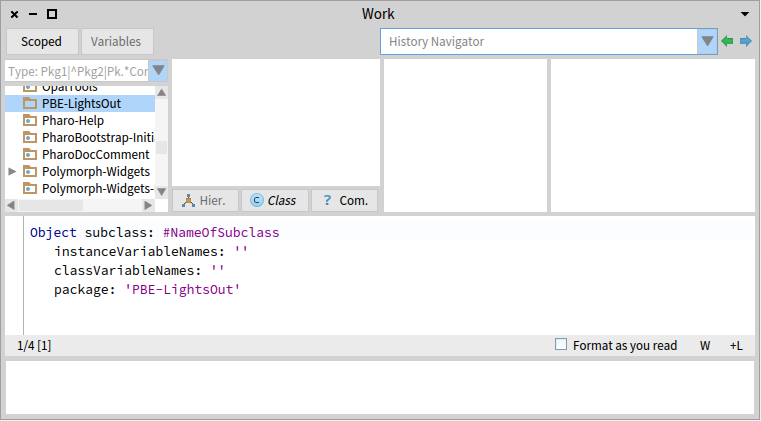


Рис. 3.2 Створення пакету та шаблон оголошення класу

## 3.1 Гра Lights Out

Щоб показати вам, як користуватись програмними інструментами Pharo, ми створимо просту гру під назвою «Lights Out». Ігрове поле – це прямокутник, заповнений жовтими клітинками. Натискання на одну з клітинок перемикає колір чотирьох сусідніх, і вони стають синіми. 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. Зараз ми покажемо, як визначити ці класи за допомогою засобів програмування Pharo.

## 3.2 Створення нового пакету класів

Ми вже познайомилися з Оглядачем класів у параграфі 2.10 і навчилися переходити до пакетів, класів та методів, і побачили, як визначати нові методи. Зараз ми побачимо, як створювати пакети та класи.

Відкрийте Оглядач класів через меню World. Контекстно клацніть по одному з наявних пакетів класів у панелі пакетів та виберіть «*Add package...*» з контекстного меню. Уведіть назву нового пакету (ми будемо використовувати *PBE-LightsOut*) у діалоговому вікні і клацніть *OK* (або натисніть клавішу *Enter*). Новий пакет буде створено й розміщено у алфавітному порядку у списку пакетів (див. рис. 3.2).

**Підказка.** Ви можете написати *’PBE’* у рядку фільтра, щоб відфільтрувати ваш пакет від усіх інших (див. рис. 3.3).

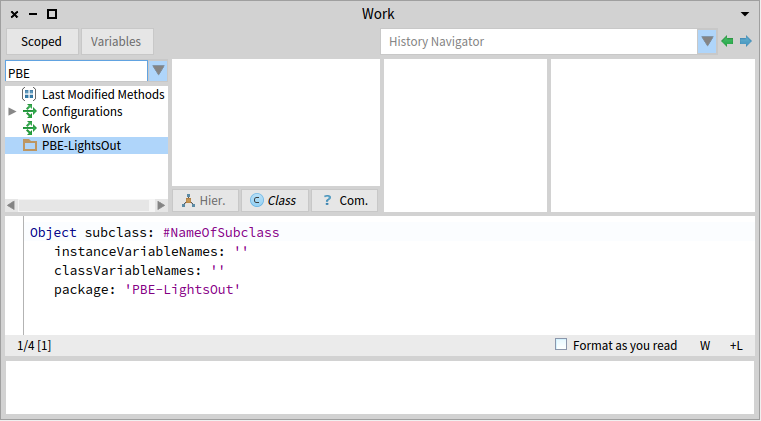


Рис. 3.3 Фільтрування імені пакету для ефективнішої роботи

## 3.3 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. 3.3).

Цей шаблон містить вираз Smalltalk, що надсилає повідомлення класові *Object* з проханням створити підклас з іменем *NameOfSubClass*. Новий клас не матиме змінних та належатиме до категорії (пакету) *PBE-LightsOut*.

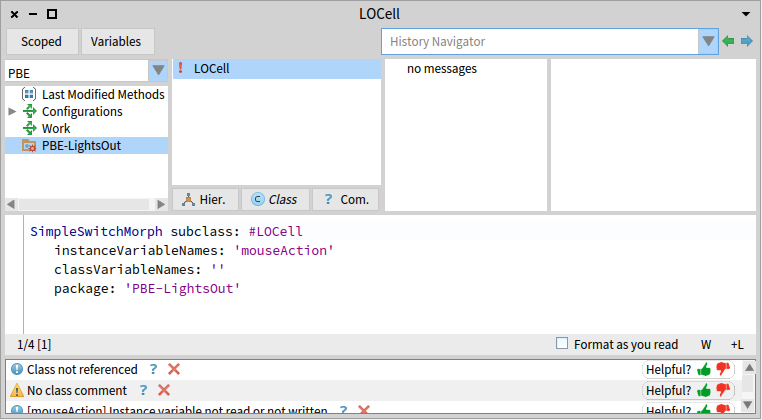


Рис. 3.4 Новостворений клас *LOCell*

### Створення нового класу

Для створення потрібного нам класу потрібно просто відредагувати запропонова­ний шаблон. Змініть шаблон створення класу наступним чином:

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

Ви мали б задати таке визначення класу:

SimpleSwitchMorph subclass: #LOCell

instanceVariableNames: 'mouseAction'

classVariableNames: ''

package: 'PBE-LightsOut'

Це визначення – це вираз Smalltalk, який надсилає повідомлення до існуючого класу *SimpleSwitchMorph* з проханням створити підклас *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*. Його ми використаємо для визначення дії, яку клітинка виконуватиме при клацанні мишки на ній.

Ви надрукували текст визначення класу, але ви все ще нічого не створили. Зауважте, що правий верхній кут панелі редагування змінив колір на оранжевий. Це означає, що у ній є незбережені зміни. Щоб справді надіслати повідомлення про створення підкласу, потрібно спочатку зберегти (прийняти) програмний код. Це можна зробити або командою «*Accept*» контекстного меню, або за допомогою комбінації клавіш *CMD-s* (*save*, зберегти). Повідомлення буде надіслано до *SimpleSwitchMorph*, що спричинить компіляцію нового класу. Ви мали б отримати результати, зображені на рисунку 3.4.

Як тільки визначення було збережено, утворився клас *LOCell*, а його ім’я з’явилося в панелі класів Оглядача (див. рис. 3.4). 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.

### Про коментарі

Фаро-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. Це хибна думка, яка заохочує неохайність. Звісно, погано написаний код крім коментування потрібно виправити та відредагувати. Коментар не мав би бути перекладом коду людською мовою, натомість він мав би пояснювати, що цей метод робить, контекст його виконання або резон створення. Очевидно, що немає змісту коментувати тривіальні методи.

**Коментарі до класів.** Ви вже знайомі з кнопкою коментарів унизу панелі класів. Натисніть її, і ви побачите, що панель коментарів новоствореного класу містить зразок якісного коментаря. Прочитайте його! Зразок побудовано згідно з CRC-дизайном[[3]](#footnote-3). Коротко кажучи, коментар кількома реченнями описує відповідальність класу, та як він взаємодіє з іншими класами, щоб реалізувати цю відповідальність. Додатково можна зазначити інтер­фейс (основні повідомлення, які розуміє екземпляр класу), навести приклад (зазвичай у Pharo визначають приклади, як методи класу) та деякі деталі внутрішнього влаштування класу чи обґрунтування реалізації.

Натисніть кнопку коментування та визначте коментар класу за цим зразком.

### Про категорії та пакети

Історично склалось так, що пакети у Pharo були реалізовані як «категорії» (групи класів). У нових версіях Pharo термін *категорії* застарів та був замінений виключно терміном *пакет*.

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

SimpleSwitchMorph subclass: #LOCell

instanceVariableNames: 'mouseAction'

classVariableNames: ''

category: 'PBE-LightsOut'

Воно еквівалентне оголошенню, яке ми записували раніше. У цій книзі ми будемо використовувати лише термін пакет. Саме пакети Pharo ви будете використовувати для контролю версій свого програмного коду за допомогою засобів Monticello.

## 3.4 Додавання методів до класу

Давайте додамо декілька методів до нашого класу. Оберіть протокол ’*no messages*’ у панелі протоколів. Ви побачите шаблон для створення методу в 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 yellow.

onColor := Color r:0.5 g:0.75 b:1.

self useSquareCorners.

self turnoff

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.

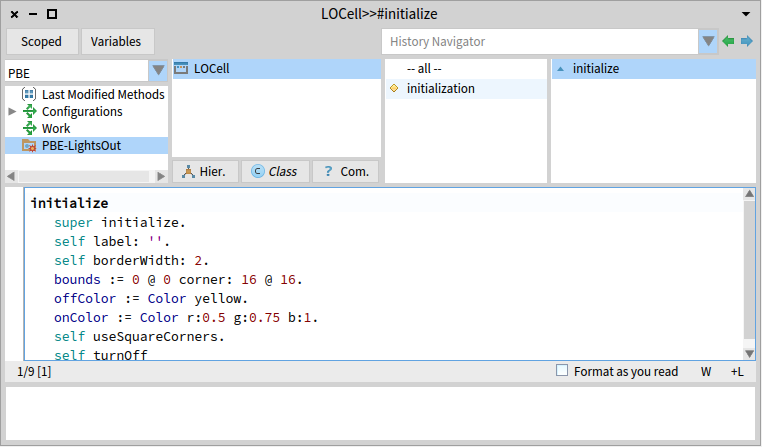


Рис. 3.5 Щойно створений метод *initialize*

**Методи ініціалізації.** Зауважимо, що метод називається *initialize*. Це ім’я дуже важливе! За домовленістю, якщо у класі визначено метод з назвою *initialize*, то 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 встановлення 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*. Суть у тому, що будь-який успадкований стан буде правильно ініціалізовано методом *initialize* надкласу. Завжди буде гарною думкою ініціалізувати успадкований стан, надсилаючи *super initialize*, перед тим, як 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 задає порожній рядок написом цього об’єкта.

**Про створення точки та прямокутника.** Вираз «*0@0 corner: 16@16*», здається, варто пояснити докладніше. *0@0* зображає об’єкт класу *Point* з обома координатами, *x* та *y*, рівними 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.

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

**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 зелену стрілку біля імені методу (див. рис. 3.5). Вона означає, що *initialize* визначений у надкласі і перевизначений у вашому класі. Ще одна дія відбулась без нашої участі: середовище автоматично віднесло наш метод до протоколу *initialization*.

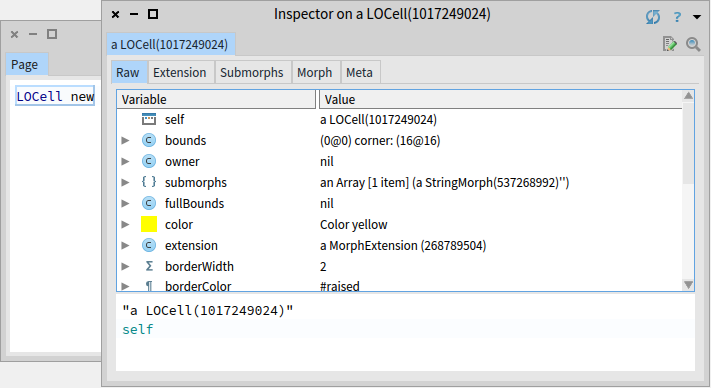


Рис. 3.6 Перевірка стану об’єкта *LOCell* за допомогою Інспектора

## 3.5 Інспектування об’єкта

Ви можете невідкладно випробувати дію написаного коду, створивши новий об’єкт класу *LOCell* і оглянувши його. Відкрийте Робоче вікно, введіть вираз «*LOCell new*» і оберіть *Inspect it* з контекстного меню (чи натисніть *CMD-i*). Лівий стовпець інспектора показує список полів, а правий – їхні значення (див. рис. 3.6).

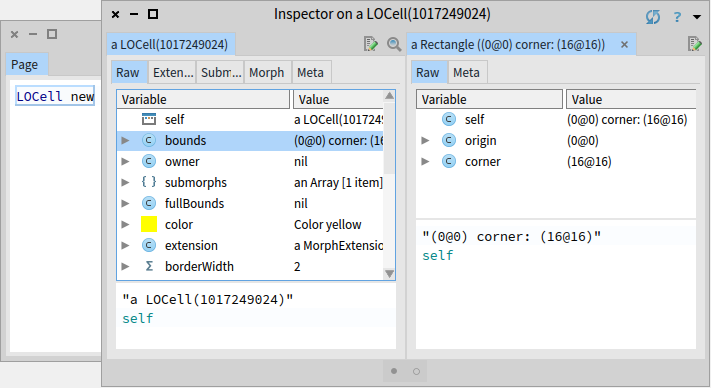


Рис. 3.7 Інспектування вкладеного об’єкта – змінної екземпляра

Якщо ви клацнете на одному з полів, Інспектор відкриє нову вкладку з деталями обраного поля (див. рис. 3.7).

**Виконання виразів.** Нижня панель інспектора працює як міні-playground. 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: (200@200 corner : 250@250)*» та виберіть *Do it*, щоб виконати його. Щоб оновити значення, відображені в Інспекторі, натисніть на кнопку «*update*» (пара синіх заокруглених стрілок) у правому верхньому куті вікна. Змінна *bounds* повинна змінитися в Інспекторі. Тепер наберіть текст «*self openInWorld*» у міні-playground та виконайте його. У лівому верхньому куті екрана повинна з’явитись клітинка, як показано на рис. 3.8. Вона з’явилась саме у тій позиції і того розміру, що вказані у змінній *bounds*.

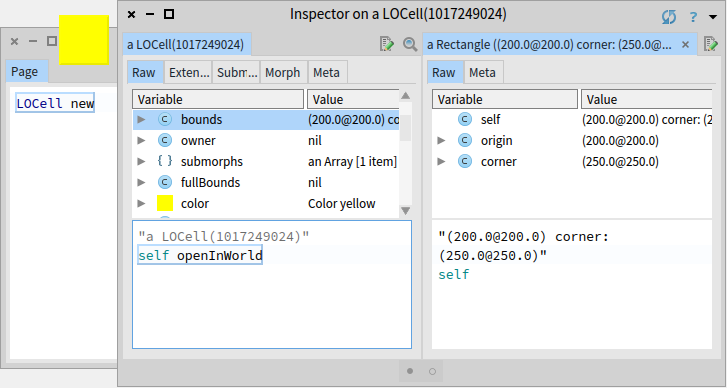


Рис. 3.8 Екземпляр *LOCell* з’явився в системі

Мета-клацніть на клітинці, щоб відкрити її меню-ореол. Перетягніть клітинку по екрану, натиснувши коричневий маніпулятор (вгорі другий від правого краю), змініть розмір клітинки жовтим маніпулятором (внизу праворуч). Прослідкуйте, як змінюються значення *bounds* в Інспекторі. (Можливо, вам доведеться натиснути кнопку «*update*».) Закрийте клітинку, клацнувши на рожевому маніпуляторі з хрестиком.

## 3.6 Визначення класу LOGame

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

### Створення класу

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 нижче, і збережіть його.

BorderedMorph subclass: #LOGame

instanceVariableNames: ''

classVariableNames: ''

package: 'PBE-LightsOut'

Тут ми наслідуємо клас *BorderedMorph*. Клас *Morph* є базовим для всіх графічних фігур у Pharo. 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.

### Ініціалізація гри

Тепер визначимо метод *initialize* для *LOGame*. Введіть наступний код у Оглядачі, як метод *LOGame* і спробуйте зберегти 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 ]

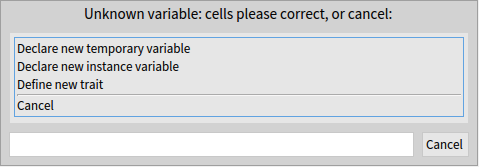


Рис. 3.9 Оголошення *cells* як нового поля даних екземпляра

Pharo поскаржиться, що він не знає, що означає *cells* (див. рис. 3.9), і запропонує вам декілька способів, як виправити ситуацію. Оберіть «*Declare new instance variable*», оскільки нам потрібне поле даних екземпляра.

### 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. Він повідомить, що нічого не знає про повідомлення *cellsPerSize* і відкриє Налагоджувач (див. рис. 3.10). Проте *cellsPerSize* не є помилкою: просто це метод, який ми ще не визначили. Ми дамо визначення вже дуже скоро.

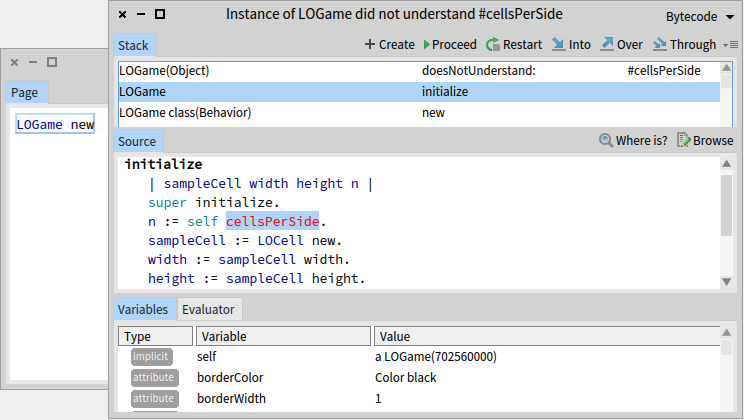


Рис. 3.10 Pharo знайшов невідомий селектор

Отож введіть «*LOGame new*» і виконайте. Не закривайте Налагоджувач, а натисніть на його кнопку «*Create*». У відповідь на запит про клас, який міститиме метод, оберіть *LOGame*. При запиті протоколу методу введіть «*accessing*» та натисніть *OK*. Налагоджувач створить метод *cellsPerSize* зі стандартним визначенням, яке ми вже бачили в п. 2.13. (див. рис. 3.11) і зупиниться, даючи можливість визначити поведінку методу.

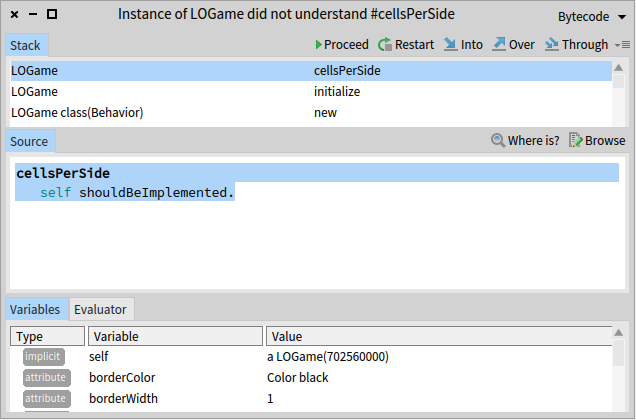


Рис. 3.11 Система створює метод, тіло якого потрібно перевизначити

Тепер можна написати метод. Цей метод важко зробити простішим: він повертає константу 10.

cellsPerSide

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

^ 10

Однією з переваг подання констант за допомогою як 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.

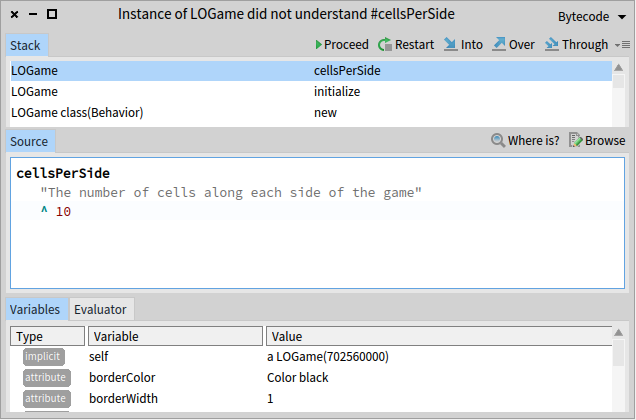


Рис. 3.12 Визначення методу *cellsPerSide* у Налагоджувачі

Визначте метод *cellsPerSide* у Налагоджувачі. Не забудьте скомпілювати його. Ви мали б отримати ситуацію, зображену на рисунку 3.12. Якщо ви натиснете кнопку «*Proceed*», програма продовжить своє виконання і зупиниться, бо ми не визначили метод *newCellAt:at:*. Ми можемо визначити його, як і попередній, але поки що зупинимось, щоб пояснити детальніше, що ми 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 (як на рис. 3.10) на звичайний чорний, і поменшало повідомлень про помилки від Помічника з якості. Червоним залишилося лише повідомлення *newCellAt:at:*.

### Вивчаємо метод *initialize*

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

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 \* self borderWidth)

).

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

* У 2 рядку 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.
* Якого розміру мало б бути поле для гри? Достатнього, щоб вмістити деяку невід’ємну кількість клітинок та намалювати межу навколо них. Скільки клітинок повинно бути? 5? 10? 100? Наразі ми не знаємо, яке число вибрати, але, якби й знали, то ймовірно ми могли б змінити свій вибір пізніше. Тому ми делегуємо відповідаль­ність за знання цієї кількості на інший метод, який називаємо *cellsPerSide* (його ще нема 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.
* Четвертий рядок використовує метод, відповідальний за розмір поля і відправляє повідомлення *cellsPerSide* до *self*, тобто самому собі. Відповідь, кількість клітинок на одній стороні ігрового поля, присвоюється змінній *n*.
* Наступні три рядки створюють новий *LOCell* об’єкт і присвоюють його ширину і висоту відповідним тимчасовим змінним.
* Рядок 8 задає межі нового об’єкта. Не вникаючи у деталі, повірте, 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, екземпляр класу *Matrix*, з правильною кількістю рядків та стовпців. Ми створили матрицю, надіславши повідомлення «*new:tabulate:*» до класу *Matrix*. Класи також є об’єктами, тому ми можемо надсилати їм повідомлення. Ми знаємо, що «*new:tabulate:*» приймає два аргументи, бо він має дві двокрапки (:) у імені. Аргументи вказують зразу після двокрапок. Якщо ви звикли до мов, що передають аргументи всі разом всередині дужок, то це спочатку може здатись дивним. Але не панікуйте, це всього лише син­таксис. І, виявляється, що це дуже навіть хороший синтаксис, оскільки назву методу можна використовувати для пояснення призначення 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*×*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*».

## 3.7 Поділ методів на протоколи

Перш ніж визначати інші методи, глянемо на третю панель угорі Оглядача. Так само, як перша панель Оглядача дозволяє нам класифікувати класи на пакети, панель протоколів дозволяє нам класифікувати методи, щоб нам не доводилось працювати з дуже довгим списком імен на панелі методів. Такі групи методів називають «протоколами». Оглядач також пропонує нам віртуальний протокол «*--all--*», який містить усі методи класу.

Якщо ви виконували приклад створення гри, то панель протоколів вашого Оглядача може містити протокол «*as yet unclassified*», до якого належать усі некласифіковані методи. (Попередні версії Pharo, на відміну від останньої, не класифікують методи автоматично.) Тоді, щоб виправити ситуацію, оберіть команду «*categorize all uncategorized*» з контекст­ного меню панелі протоколів та перемістіть метод *initialize* до запропонованого системою протоколу *initialization*.

Але як Системний оглядач знає, що це правильний протокол? У загальному випадку Pharo не може знати, але тут 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.

**Домовленість про позначення.** Програмісти Pharo часто використовують нотацію «*Class>>method*», щоб ідентифікувати клас, якому належить метод. Наприклад, щоб вказати метод *cellsPerSide* класу *LOGame*, записують так: *LOGame>>cellsPerSide*. Запис не є частиною синтаксису Pharo, а, швидше, зручною нотацією, щоб сказати «метод *cellsPerSide* екземпляра класу *LOGame*». Відповідна нотація для методу класу буде *LOGame class >> someClassSideMethod*.

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

## 3.8 Завершення розробки гри

А зараз давайте визначимо інші методи, використані в *LOGame>>initialize*. Давайте визначимо *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 ]

**Форматування**. Як ви можете бачити, у коді є порожні рядки та табуляція. Для того, щоб дотримуватись однакових правил форматування коду, ви можете вибрати команду «*Format*», першу команду контекстного меню панелі редагування методу, або використати *CMD-Shift-f*. Це автоматично відформатує текст вашого методу.

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 обробітника події клацання мишкою на клітинці. Відповідний метод також потрібно визначити. Для цього клацніть на якомусь протоколі, наприклад, на «*--all--*» і замініть шаблон визначення методу справжнім кодом.

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.

У який протокол потрапив ваш новий метод? Він або залишився некласифікованим, або потрапив. до *accessing*. У будь-якому випадку його потрібно перемістити у новий протокол, що називається «*game logic*». Щоб додати новий протокол, використайте відповідну команду контекстного меню панелі протоколів. Тепер, щоб перемістити метод до іншого протоколу, просто перетягніть його мишею (див. рис. 3.13).

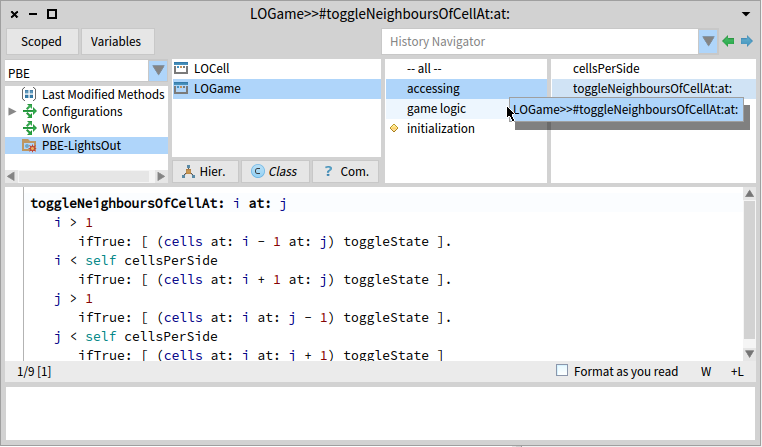


Рис. 3.13 Переміщення методу до нового протоколу перетягуванням

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

LOCell>>mouseAction: aBlock

^ mouseAction := aBlock

Цей метод не робить нічого особливого, лише присвоює змінній *mouseAction* клітинки свій аргумент і повертає нове значення. Будь-який метод, що змінює значення поля у такий спосіб, називається *модифікатором* (setter method). Метод, що повертає поточне значення поля, називається *селектором* (getter method). Не плутайте метод-селектор зі словом-селектором, частиною повідомлення.

**Домовленість про іменування селекторів та модифікаторів.** Якщо ви використо­вували селектори та модифікатори в інших мовах програмування, то ви можете очікувати, що ці методи 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 відносять до протоколу *accessing*. У Smalltalk 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. Насправді, заборона доступу не поширюється на підкласи: об’єкт має вільний доступ до всіх успадкованих полів.

Перейдіть до класу *LOCell*, визначте метод *LOCell>>mouseAction:* і перемістіть його у протокол *accessing*. Якщо у вас Pharo версії 6.1, класифікація відбудеться автоматично.

Нам залишилось визначити метод *mouseUp:*. Його автоматично викликатиме графічний інтерфейс користувача при відпусканні кнопки миші, коли курсор є над певною клітинкою. Додайте метод *LOCell>>mouseUp:* і виконайте автоматичну класифікацію мето­дів (*Categorize all uncategorized*), якщо Оглядач ще не зробив цього сам.

LOCell>>mouseUp: anEvent

mouseAction value

Цей метод надсилає повідомлення *value* об’єктові, що зберігається у полі *mouse­Action*. У *LOGame>>newCellAt:at:* ми створили блок, який перемикає усіх сусідів клітинки, та присвоїли його змінній *mouseAction* клітинки. Отже, відправлення повідомлення *value* спричинить виконання цього блоку і зміну станів усіх сусідніх клітинок.

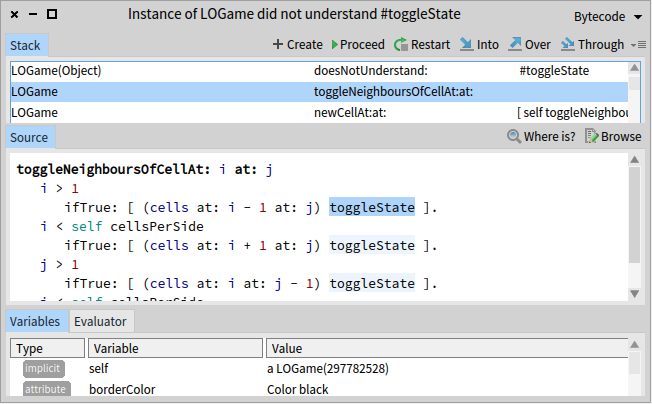


Рис. 3.14 Метод, що спричинив помилку, в Налагоджувачі

## 3.9 Випробуємо код

Це все: створення гри Lights Out завершене! Якщо ви виконали усі кроки, то зможете зіграти у гру, що складається з всього лише 2 класів та 7 методів. У Робочому вікні наберіть «*LOGame new openInWorld*» та виконайте (*Do it*).

Гра відкриється, і ви мали б клацати по клітинках і бачити, як усе працює. Принаймні так воно 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 Smalltalk, що виконується у цьому методі. Частина, що спричинила помилку, буде підсвічена.

Натисніть на рядку з написом «*LOGame toggleNeighboursOfCellAt:at:*» (вгорі списку). Налагоджувач покаже контекст виконання методу, де сталася помилка (див. рис. 3.14).

Унизу Налагоджувача є ділянка змінних. 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 ви можете побачити значення аргументів методу.

За допомогою Налагоджувача ви 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 Фаро-програмісти майже постійно працюють у Налагоджувачі – більше, ніж в Оглядачі класів. Перевагою такого підходу є те, що ви бачите, як буде виконано метод, який ви пишете, зі справжніми параметрами та в актуальному контексті виконання.

У нашому випадку заголовок вікна Налагоджувача (див. рис. 3.14) 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 Smalltalk повертає *self* – отримувача, який у випадку *newCellAt:at:* справді є екземпляром *LOGame*. Як ми уже говорили, для повернення значення з методу у Smalltalk використовують позначення «^ *вираз*».

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

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

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 мета-клацнути 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 openInWorld*» знову, бо якщо використати старий екземпляр гри, то працюватиме блок зі старою логікою.

Зараз гра повинна 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:

LOCell>>mouseMove: anEvent

Нарешті ми закінчили!

**Про налагоджувач.** Коли трапляється помилка у Pharo, система за замовчуванням відображає налагоджувач. Проте ми можемо повністю контролювати цю поведінку. Наприклад, ми можемо записати помилку у файл. Ми можемо навіть серіалізувати стек виконання до файлу, архівувати і відкрити його в іншому образі системи. Проте в режимі розробки налагоджувач доступний, щоб дозволити нам рухатись так швидко, як це можливо. У готових системах розробники часто налаштовують налагоджувач так, щоб приховувати свої помилки від клієнтів.

## 3.10 Удосконалення гри

Ми щойно закінчили, а вже виникли думки щодо удосконалення нашого продукту.[[4]](#footnote-4) Чи дуже зручно вам закривати гру маніпуляторами 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*.

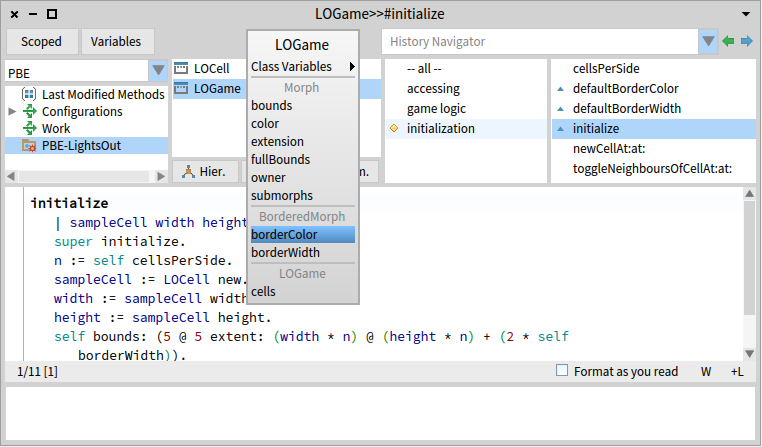


Рис. 3.15 Перелік змінних екземпляра класу *LOGame*

### 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*, позначте мишкою клас чи будь-який його метод і клацніть на кнопці «*Variables*» Оглядача (вгорі друга зліва). З’явиться невелике вікно з переліком усіх змінних екземпляра, структурованим за класами, в яких ці поля оголошені, як на рисунку 3.15. 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.

. . .

Якщо зберегти зроблені зміни і запустити гру, то побачимо, що все працює, гра набула саме такого вигляду, як на рисунку 3.1. Але не все так добре, як може здатися на перший погляд. Адже поля *borderColor* і *borderWidth* оголошені в класі *BorderedMorph*, і саме він повинен задавати їхні початкові значення. Схоже, ми запрограмували повторну ініціа­лізацію змінних *borderColor* і *borderWidth*. Це двічі неправильно: виконується зайва робота, і 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. Внизу панелі класів Оглядача зліва є кнопка «*Hier.*», що відображає в панелі ієрархію наслідування класів незалежно від їхньої приналежності до пакетів. Натисніть її, послідовно клацніть на імені класу *BorderedMorph*, пакету *initialization*, методу *initialize*. Ви мали б перейти до методу ініціалізації надкласу, як зображено на рисунку 3.16.

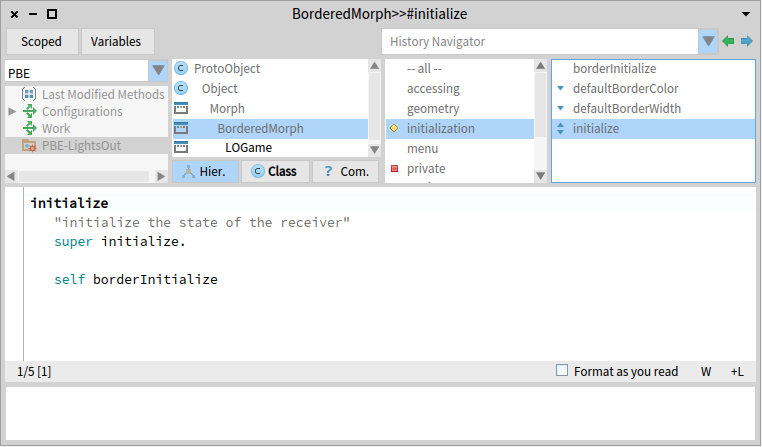


Рис. 3.16 Ієрархія наслідування класів

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

BorderedMorph>>borderInitialize

"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*. Зауважимо, що в мові Smalltalk 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 Робочого вікна). 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 ми спробуємо.

Відкрийте Сищика і в рядку пошуку наберіть «*world menu*». Єдиний результат пошуку – посилання на розділ «World Menu Items» довідкової системи – це саме те, що потрібно. В ньому написано, що для того, щоб додати новий пункт до головного меню, потрібно визначити *метод класу*, який називається *menuCommandOn*:. Довідка надає також декілька прикладів оголошення такого методу в деякому навчальному класі. Не зайвим буде також пошукати в Pharo справжні класи, які реалізують метод *menuCommandOn*:, і подивитися, як воно зроблено. Ви вже вмієте знаходити інформацію за допомогою Сищика чи Пошуковця.

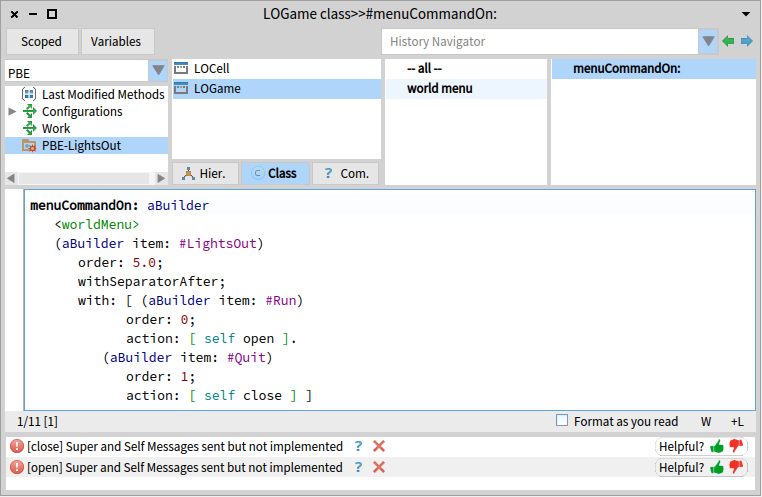


Рис. 3.17 Визначення методу класу

Давайте поміркуємо, чого ми хочемо від команди меню. Вона мала б виконувати той код, який ми набирали в Робочому вікні, 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 натисніть середню кнопку «*Class*» панелі класів (див. рис. 3.17). Оглядач відобразить перелік методів класу *LOGame* (він поки що порожній). Класи Pharo – також об’єкти, які можуть мати власні методи. Нам потрібно оголосити такий:

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 Smalltalk задають каскад повідомлень, які надходять до одного і того ж отримувача (до пункта 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.

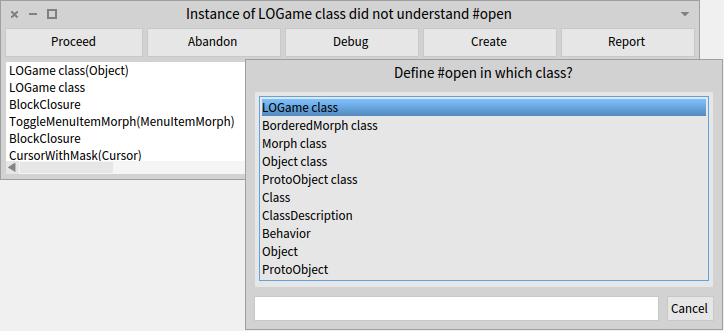


Рис. 3.18 Створення методу класу в Налагоджувачі

Зрозуміло, що команда меню «*Run*» повинна 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 сталася, тому можемо відразу натиснути кнопку «*Create*», вказати клас «*LOGame class*» (див. рис. 3.18), протокол «*instance creation*» і ввести текст методу.

Знайомий нам код «*LOGame new openInWorld*» 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 Smalltalk є свої змінні!), яку назвемо «*TheGame*».

У вікні редагування коду Налагоджувача введіть текст методу:

LOGame class>>open

TheGame isNil

ifTrue: [ TheGame := self new.

TheGame openInWorld ]

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*. 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. Оглядач класів класифікує його до протоколу «*initialize-release*».

LOGame class>>close

TheGame delete.

TheGame := nil

Знову натисніть на кнопку «*Proceed*» Налагоджувача – закриється і Налагоджувач, і гра. Тепер команди головного меню працюватимуть як слід. можете випробувати їхню дію (див. рис. 3.19).

|  |  |
| --- | --- |
| Рис. 3.19. Доповнене головне меню Pharo | Рис. 3.20 Виведення пакету класів до файла |

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.

## 3.11 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. Що вам потрібно, так це спосіб скопіювати код з вашого образу Pharo так, щоб інші програмісти змогли долучити його до їхніх власних образів.

**Зауваження.** Ми розглянемо різні способи збереження та поширення коду згодом у розділі «Поширення коду та контроль версій». Зараз ми лише оглянемо декілька доступних способів.

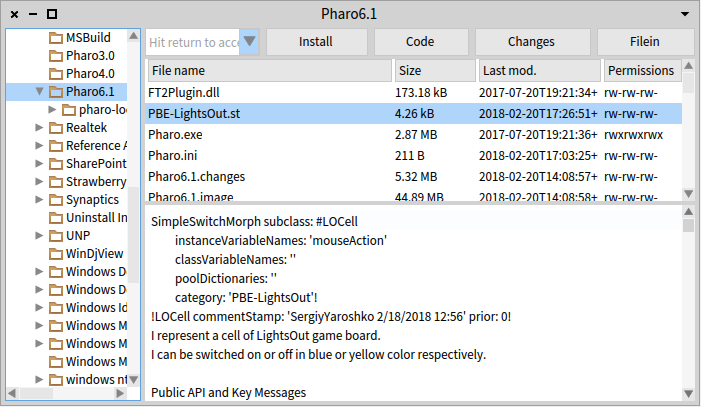


Рис. 3.21 Імпортування коду за допомогою оглядача файлів

### Збереження простого тексту

Найпростіший спосіб отримати текст створених класів – це записати пакет до файла. Позначте в панелі пакетів Системного оглядача *PBE-LightsOut* і виберіть команду «*File Out*» контекстного меню (див. рис. 3.20). Вона виведе весь пакет до файла «*PBE-LightsOut.st*», створеного в тій папці, де зберігається образ системи. Перегляньте отриманий файл за допомогою редактора текстів. Він більш-менш читабельний, але призначений він у першу чергу для комп’ютерів, а не для людей. Ви можете надіслати цей файл друзям і вони можуть увести його у власний образ Pharo за допомогою оглядача файлів.

Відкрийте новий образ Pharo та за допомогою оглядача файлів (*World* > *Tools* > *File Browser*) і кнопкою «*Filein*» імпортуйте файл *PBE-LightsOut.st* (див. рис. 3.21). Переконай­теся, що гра працює у новому образі.

### Пакети Monticello

Розглянутий вище спосіб виведення коду до файла є досить зручним, але він за означенням належить до «старої школи». Так само, як більшість проектів з відкритим кодом вважають зручним зберігати код у сховищах, використовуючи SVN чи Git, так і програмісти Pharo вважають, що зручніше керувати власним кодом за допомогою пакетів Monticello. Такий пакет зберігається у файлі з розширенням *.mcz*. Насправді він є архівова­ним пакунком (bundle), що містить весь код вашого пакету.

За допомогою менеджера Monticello ви можете зберігати пакети у сховищах на серверах різного типу, включно з FTP та HTTP серверами. Ви також можете записувати пакети до сховища у локальній файловій системі. Копія вашого пакету завжди кешується на локальному диску у папці *package-cache*. Monticello дає змогу зберігати кілька версій однієї програми, зливати версії, переходити до старих версій і переглядати різницю між версіями. Фактично Monticello – це розподілена система контролю версій. Це означає, що розробники можуть зберігати результати своєї роботи в різних місцях, а не в одному сховищі, як це є у випадку використання CVS чи Subversion.

Ви можете також пересилати *.mcz*-файли електронною поштою. Адресат помістить отриманий файл до своєї папки *package-cache* і зможе переглядати і завантажувати його за допомогою Monticello.

### Менеджер Monticello

Відкрийте менеджер Monticello через меню World (див. рис. 3.22).

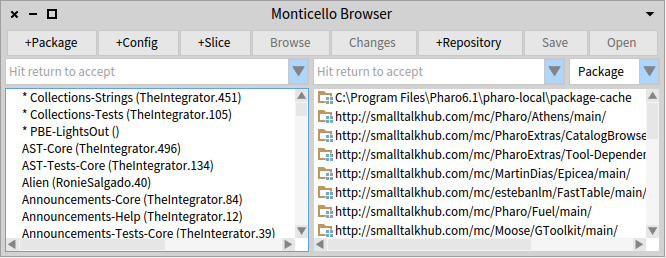


Рис. 3.22 Браузер Monticello. Пакет PBE-LightsOut поки що не збережено

У панелі праворуч відображено список сховищ Monticello, який включає усі сховища 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 були завантажені або опубліковані через мережу. Такий локальний кеш справді зручний, оскільки зберігає вашу особисту історію взаємодії зі сховищами. Він також дозволяє працювати в місцях, у яких немає доступу до інтернету, або доступ є настільки повільним, що ви не захочете часто зберігати зміни у віддаленому сховищі.

### Збереження та завантаження коду за допомогою Monticello

У панелі ліворуч у вікні Monticello відображено список пакетів, будь-яка версія яких завантажена в поточний образ системи. Назви пакетів, які були модифіковані після завантаження, позначені зірочкою. (Їх іноді називають зіпсутими пакетами.) Якщо ви виберете пакет, список сховищ показуватиме лише ті, що містять копію вибраного пакету.

Додайте пакет *PBE-LightsOut* до менеджера Monticello: натисніть кнопку «*+Package*» та введіть «*PBE-LightsOut*».

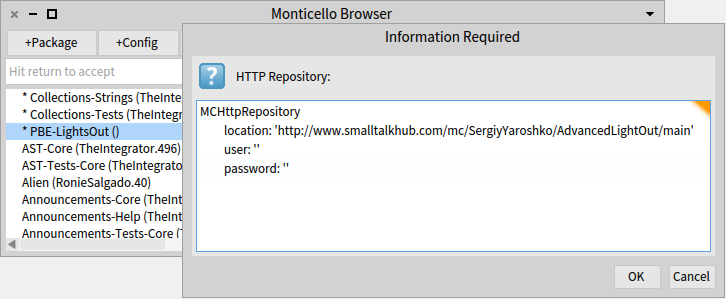


Рис. 3.23 Створення першого сховища на SmalltalkHub

### SmalltalkHub: Github для Pharo

На нашу думку, найкращим способом зберігати код та ділитись ним є створити обліковий запис для проекту на SmalltalkHub. SmalltalkHub схожий на Github: це веб-інтерфейс для HTTP-сервера Monticello, який дає змогу керувати вашими проектами. Публічний сервер доступний за адресою http://www.smalltalkhub.com/.

Щоб мати змогу використовувати SmalltalkHub, вам потрібно мати обліковий запис. Відкрийте сайт у браузері. Натисніть на *Join* та згідно з інструкціями створіть новий обліковий запис. Тоді увійдіть у нього. Клацніть на «*+ New project*», щоб створити новий проект. Заповніть необхідну інформацію та натисніть кнопку «*Register project*». Ви перейдете на сторінку вашого профілю, де зліва буде список ваших проектів та проектів інших програмістів, за якими ви слідкуєте. Натисніть на новостворений проект.

Під заголовком «*Monticello registration*» ви побачите рамку з виразом Smalltalk, схожим на

MCHttpRepository

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

user: 'YourName'

password: 'Optional\_Password'

Скопіюйте її вміст та поверніться до Pharo. Після створення проекту на SmalltalkHub потрібно вказати системі Pharo, що його можна використовувати. У менеджері Monticello позначте пакет *PBE-LightsOut*, натисніть кнопку «*+Repository*». Ви побачите список доступ­них сховищ різних типів. Щоб додати сховище SmalltalkHub, оберіть «*smalltalkhub.com*». Далі відкриється діалогове вікно, де потрібно ввести інформацію про сервер. Вставте код, скопійований раніше з SmalltalkHub (див. рис. 3.23). 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 сховищі на SmalltalkHub.

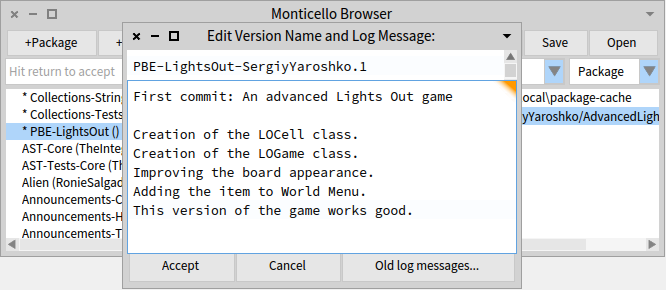


Рис. 3.24 Перша фіксація пакету

Як тільки ви натиснете кнопку «*OK*», адреса нового сховища з’явиться у правій панелі менеджера 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. рис. 3.24).

Щоб завантажити пакет у ваш образ, відкрийте Monticello, оберіть сховище, в якому розміщений пакет, та натисніть кнопку «*Open*». З’явиться нове вікно з двома панелями. У лівій буде пакет, який потрібно завантажити, а у правій – версії цього пакету. Оберіть пакет та версію, яку ви хочете завантажити, та натисніть кнопку «*Load*».

Відкрийте сховище PBE-LightsOut, яке ви щойно зберегли, в менеджері. Ви побачите пакет, який ви щойно зберегли. Зірочки перед його назвою більше нема!

Monticello має багато інших функцій, які ми детально розглянемо у розділі «Поширення коду та контроль версій».

**Про Git.** Якщо ви вже знайомі з Git і Github, то можете використати і його для ке­рування своїми проектами. Для цього є декілька рішень. Наступний блог пропонує стислий опис: http://blog.yuriy.tymch.uk/2015/07/pharo-and-github-versioning-revision-2.html#! Тут готується параграф про те, як використовувати Git з Pharo. Попросіть його в списку розсилання Фаро.

## 3.12 Підсумки розділу

У цьому розділі ви побачили, як створюють пакети, класи та методи. Також ви навчились користуватись Системним оглядачем, Інспектором, Налагоджувачем, Сищиком та менеджером Monticello.

* Пакети – це групи пов’язаних класів.
* Новий клас створюють, надсилаючи повідомлення його надкласу.
* Протоколи – це групи пов’язаних методів у класі.
* Новий метод створюють або змінюють редагуванням його визначення в Оглядачі з наступним збереженням змін.
* Інспектор надає простий графічний інтерфейс для спостереження та взаємодії з довільними об’єктами.
* Оглядач виявляє використання неоголошених змінних та пропонує можливі виправлення.
* Метод *initialize* автоматично виконується при створенні об’єкта у Pharo. Ви можете вставити в нього будь-який код ініціалізації.
* Налагоджувач надає високорівневий графічний інтерфейс для спостереження та зміни стану запущеної програми.
* Ви можете оголошувати змінні та методи і екземпляра, і класу.
* Ви можете ділитись кодом пакету, класу чи методу, записаним до файла.
* Кращий спосіб поділитись кодом – використати Monticello для керування зовнішнім сховищем, наприклад визначеним, як проект SmalltalkHub.

# Розділ 4. Синтаксис у двох словах

Синтаксис Pharo дуже близький до синтаксису його предка, Smalltalk. Синтаксис такий, що програмний код при читанні схожий на спрощену англійську.

Наступний метод класу Week демонструє приклад синтаксису. Він перевіряє чи DayNames містить аргумент, тобто чи аргумент є правильною назвою дня тижня. Якщо так, то цей аргумент присвоюється змінній StartDay.

startDay: aSymbol

(DayNames includes: aSymbol)

ifTrue: [ StartDay := aSymbol ]

ifFalse: [ self error: aSymbol, ' is not a recognised day name' ]

Синтаксис у Pharo простий. По суті, складається з відправки повідомлення (вирази). Складаються вирази з невеликого числа примітивних елементів. Є 6 ключових слів, і немає синтаксису для керування структурами та створення нових класів. Замість цього використовується відправка повідомлень об'єктам. Наприклад, замість керуючої структури if-then-else використовується посилка повідомлення ifTrue: логічному (Boolean) об'єкту. Нові класи створюються відправленням повідомлення суперкласу.

## 4.1 Синтаксичні елементи

Вирази будуються за допомогою наступних елементів:

1. Шість зарезервованих ключових слів або псевдо змінних: self, super, nil, true, false, і thisContext
2. Константні вирази літеральних об’єктів, що включають числа, літери, рядки, символи та масиви
3. Оголошення змінної
4. Присвоювання
5. Блоки повідомлень
6. Повідомлення

Нижче наведено таблицю з прикладами різних синтаксичних елементів.

|  |  |
| --- | --- |
| **Синтаксис** | **Що означає** |
| startPoint | Назва змінної |
| Transcript | Назва глобальної змінної |
| self | Псевдо змінна |
| 1 | Десяткове ціле число |
| 2r101 | Двійкове число |
| 1.5 | Дійсне число |
| 2.4e7 | Експоненціальний запис числа |
| $a | Літера ‘а’ |
| ‘Hello’ | Рядок ‘Hello’ |
| #Hello | Символ #Hello |
| #(1 2 3) | Літеральний масив |
| { 1. 2. 1+2} | Динамічний масив |
| “a comment” | Коментар |
| |xy| | Тимчасові змінні |
| X:=1 | Присвоювання 1 змінній х |
| [:x|x+2] | Блок, який обчислює х+2 |
| <primitive: 1> | Віртуальний примітивний метод |
| 3 factorial | Унарне повідомлення |
| 3 + 4 | Бінарне повідомлення |
| 2 raisedTo: 6 module 10 | Ключове повідомлення |
| ^ true | Повернення значення true |
| Transcript show: ‘hello’. Transcript cr | Розділювач виразів (.) |
| Transcript show: ‘hello’;cr | Розділювач каскадних повідомлень (;) |

**Локальні змінні.** startPoint — це ім’я змінної або ідентифікатор. Умовно, ідентифікатори складаються зі слів в “camelCase” (тобто кожне слово, за виключенням першого, починається з великої літери). Перша літера змінної екземпляра, метода, аргумента блоку чи тимчасової змінної повинна бути малою. Це вказує читачеві, що змінна в приватній області видимості.

**Загальні змінні**. Ідентифікатори загальних змінних починаються з великої літери і до них відносяться глобальні змінні, змінні класу, словники та назви класів. Transcript є глобальною змінною і екземпляром класу TranscriptStream.

**Отримувач**. Ключове слово self посилається на об’єкт поточного метода, який виконується. Ми називаємо його “Отримувачем” тому, що це об’єкт, на який посилається self, є отримувачем повідомлення. self називається псевдо змінною так як ми не можемо призначити цій змінній інший об’єкт.

**Цілочислові**. В доповнення до звичайних десяткових цілих чисел, наприклад 42, Pharo використовує і інші системи числення. 2r101 - це 101 у двійковій системі числення (бінарній), яке рівне десятковому числу 5.

**Дійсні числа**. Можуть записуватися з використанням степені числа 10: 2.4e7 — це 2.4 × 10 7.

**Літерали**. Знак долара і наступний літерал позначають екземпляр класу Character, наприклад $a. Недруковані символи можна отримати, відправивши відповідні повідомлення класу Character, такі як Character space або Character tab.

**Рядки**. Одинарні лапки використовуються для позначення екземпляра класу String, який складається із послідовності букв и символів. Якщо потрібно в рядок (String) додати одинарну лапку, то її потрібно продублювати, наприклад 'G ''day '.

**Символи**. так як і рядки (Strings), містять послідовності букв. Але вони відрізняються від рядків тим, що вони є унікальними. В системі може бути тільки один символьний об’єкт #hello, а рядків 'hello ' скільки завгодно.

**Масиви, які створюються на етапі компілювання**. Визначаються за допомогою #() і між дужками вводять літеральні об’єкти, розділюючи їх пробілами. Все, що є між дужками, створюється під час компіляції. Наприклад, #(27 (true false) abc) складається з трьох елементів: цілого числа 27, масиву, який містить два логічних об’єкти и символу #abc. Цей самий приклад можна записати по-іншому: #(27 #(true false) #abc).

**Масиви, які створюються на етапі виконання**. Фігурні дужки {} визначають (динамічний) масив на етапі виконання. Елементи масиву – вирази, що розділяються крапкою. Наприклад, {1. 2. 1+2.} визначає масив з трьох елементів: 1, 2 и результат виразу 1+2.

**Коментарі**. Поміщені у подвійні лапки “ ”. "hello" – це коментар, не рядок (String) і ігнорується компілятором Pharo. Коментарі можуть містити кілька рядків.

**Визначення локальних змінних**. Вертикальні полоски | | містять оголошення однієї чи кількох локальних змінних в методі (також і в блоці).

**Присвоювання обєкта змінній** здійснюється оператором :=.

**Блоки**. Квадратні дужки [] визначають блок (block), також відомий як block closure чи lexical closure, обєкт який представляє собою функцію. Як ми побачимо далі, блоки можуть мати аргументи і локальні змінні.

**Примітиви**. <primitive: ...> позначає посилання на примітив віртуальної машини. Наприклад, <primitive: 1> - це VM примітив для Smallinteger. Весь код, який розташований за примітивом, виконується тільки, якщо примітив закінчився з помилкою. Аналогічний синтакс використовується для анотації методів.

**Унарні повідомлення**. Вони складаються з одного слова (наприклад, factorial), яке посилається отримувачу (наприклад 3). У виразі «3 factorial» 3 це отримувач, а factorial це селектор повідомлення.

**Бінарні повідомлення**. Це оператори (+), які посилаються отримувачу і приймають один аргумент. У виразі 3 + 4 отримувачем є 3, селектором повідомлення є + і аргументом є 4. Результатом такого виразу є ціле число 7.

**Ключові повідомлення**. Складаються з кількох слів (наприклад, raisedTo:modulo:), кожне з яких закінчується двокрапкою і приймає один аргумент. У виразі «2 raisedTo: 6 modulo: 10» селектор повідомлення «raisedTo:modulo:» приймає два аргументи 6 и 10, які розміщені після двокрапки. Повідомлення відправляється отримувачу 2.

**Повернення значення з метода**. Оператор ^ використовується для повернення відповіді з метода.

**Послідовності повідомлень**. Крапка (.) є розділювачем повідомлень. Якщо між двома виразами поставити крапку, то вони перетворюються у два незалежні повідомлення.

**Каскадні повідомлення**. Крапка з комою (;) використовується для відправки кількох повідомлень одному отримувачу. У виразі Transcript show: 'hello '; cr. спочатку відправляється повідомлення show: 'hello ' отримувачу Transcript, після цього відправляється унарне повідомлення cr цьому ж отримувачу.

Класи Number, Character, String I Boolean описані більш детально у розділі 10: Базові класи.

## 4.2 Псевдо змінні.

У Pharo є 6 зарезервованих слів або псевдо змінних: nil, true, false, self, super та thisContext. Вони називаються псевдо змінними тому, що їх значення наперед визначені і не можуть бути змінені. true, false i nil є константами, а значення змінних self, super та thisContext є динамічними, бо залежать від коду.

* ***true*** i ***false*** є унікальнми екземплярами логічних класів True та False відповідно. Див. Розділ 10: Базові класи.
* ***self*** завжди посилається на отримувача поточно виконуваного метода.
* ***super*** також посилається на отримувача поточного метода, але відмінність в тому, що ми посилаємо повідомлення суперкласу. Для більш детальної інформації дивіться Розділ 6: Обєктна модель Pharo.
* ***nil*** це невизначений обєкт. Є унікальним екземпляром класу UndefinedObject. Змінні екземплярів, класів і локальні змінні при ініціалізації встановлюються в значення nil.
* ***thisContext*** це псевдо змінна, яка представляє верхню частину стеку виконання. thisContext переважно не представляє інтересу для більшості програмістів, але він потрібен для розробки інструментів, таких як відладжувач (debugger), та для реалізації виключень.

## 4.3 Відправка повідомлення.

Є три типи повідомлень у Pharo:

1. Унарні повідомлення складаються тільки з селектора без аргументів. «1 factorial» - відправляє повідомлення factorial обєкту 1.
2. Бінарні повідомлення складаються з селектора та одного аргумента. «1+2» - відправляє повідомлення + з аргументом 2 обєкту 1.
3. Ключові повідомлення складаються з довільної кількості аргументів. «2 raisedTo: 6 modulo: 10» - відправляє повідомлення, яке складається з селектора raisedTo:modulo і аргументів 6 та 10, обєкту 2.

Селектори унарних повідомлень складаються з букв та цифр і починаються з малої букви.

Селектори бінарних повідомлень складаються з одного з наступних символів: +=/\\*~<>=@%|&?,

Селектори ключових повідомлень складаються з серії ключових слів, кожне з яких складається з букв, цифр і починається маленькою буквою і закінчується двокрапкою.

Унарні повідомлення мають вищий пріоритет ніж бінарні, а бінарні вищий ніж ключові повідомлення.

2 raisedTo: 1 + 3 factorial -> 128

Спочатку ми надсилаємо factorial обєкту 3, потім надсилаємо + 6 обєкту 1, і потім надсилаємо raisedTo: 7 обєкту 2.

Зауважимо, що обчислення виразу відбувається зліва направо, тому:

1 + 2 \* 3 -> 9.

Для зміни порядку обчислення використовують круглі дужки:

1 + (2 \* 3) –> 7

Надсиалння повідомлень може комбінуватися з крапкою та крапкою з комою. Послідовність виразів розділених крапкою буде обчислюватись по черзі.

Transcript cr.

Transcript show: ‘hello world’.

Transcript cr.

Спочатку відправиться повідомлення cr обєкту Transcript, потім відправиться show: ‘hello world’ і потім cr.

Коли повідомлення відправляється одному і тому ж отримувачу, то краще використовувати каскадні повідомлення. Отримувач описується один раз, а послідовність повідомлень розділяється крапкою з комою.

Transcript cr; show: ‘hello world’; cr

## 4.4 Синтаксис методу.

Слід зауважити, що вирази виконуються всюди (workspace, debugger, browser і т.д.) а методи зазвичай створюються (редагуються) в браузері класів, або у відладчику (debugger). Методи також можна створювати з файлів, але такий шлях використовується рідко.

Програми розробляються зміною або створенням нових методів у певному класі. (Новий підклас створюється відправленням повідомлення існуючому класу).

Наведений нижче метод lineCount розташований у класі String. (Далі при описі методів ми будемо використовувати запис виду ClassName >> MethodName, тому ми викликаємо цей метод так – String >> lineCount).

String >> lineCount

"Answer the number of lines represented by the receiver, where every cr adds

one line."

| cr count |

cr := Character cr.

count := 1 min: self size.

self do:

[:c | c == cr ifTrue: [count := count + 1]].

^ count

Синтаксично метод складається з:  
1. заголовка методу, що включає в себе ім'я (ie, lineCount) і аргументів (відсутні в даному прикладі);  
2. коментарів (можуть розташовуватися в будь-якій частині методу, але за згодою зазвичай йдуть після заголовка і пояснюють, що робить метод);  
3. визначення локальних змінних;  
4. кількох пропозицій розділеними крапками, в даному випадку 4.

На початку пропозиції може розташовуватися символ ^, який означає, що після виконання даної пропозиції, метод поверне відправнику результат. При відсутності в методі символу ^, одержувач отримає як результат значення self.

Аргументи і локальні змінні слід завжди починати з малої літери. Імена, що починаються з великої літери, використовуються для глобальних змінних. Імена класів, наприклад Character, є звичайними глобальними змінними і мають посилання на об'єкт, який представляє клас.

## 4.5 Синтаксис блока.

Блоки забезпечують механізм відкладеної послідовності дій. Блок, по суті, анонімна функція. Блок обчислюється, посилаючи повідомлення value. Результатом виконання зазвичай є значення останнього виразу в тілі блока, але тільки якщо немає явного повернення результату (^).

[1 + 2] value → 3

Блоки можуть мати аргументи, які починаються з символу (:) двокрапки. Вертикальна смуга відокремлює тіло блоку від аргументів. При виконанні блоку з одним аргументом, потрібно відправити повідомлення value: argument. Якщо аргументів у блоку 2, тоді повідомлення зміниться на value: argument1 value: argument2 і т.д. до 4 аргументів.

[: X | 1 + x] value: 2 → 3

[: X: y | x + y] value: 1 value: 2 → 3

Якщо блок більш ніж з чотирма параметрами, тоді треба використовувати повідомлення valueWithArguments: argumentArray, де argumentArray - масив аргументів. (Не рекомендується використовувати блоки з великим числом параметрів).

Блок дозволяє визначати локальні змінні, які розділяються вертикальними смугами як і при декларуванні змінних методу. Розташовані вони після аргументів блоку.

[: x: y | | z | z: = x + y. z] value: 1 value: 2 → 3

Блоки можуть посилатися на змінні розташовані поза нього. У наступному прикладі блок посилається на змінну x розташовану в методі, не в блоці.

| x |

x: = 1.

[: y | x + y] value: 2 → 3

Також вони є екземплярами класу BlockClosure. Це означає, що вони є об'єктами і можуть бути приєднані до змінної і відправлені як аргумент будь-якого об'єкта.

## 4.6 Порівняння і цикли у двох словах.

У Фаро відсутній спеціальний синтаксис для керуючих структур. Замість них використовуються пропозиції з посилкою повідомлень булевим, числовим і колекціям. А в якості аргументів використовуються блоки.

Порівняння - це пропозиція, де надсилається повідомлення ifTrue :, ifFalse: або ifTrue: ifFalse результату булевого виразу. Більш докладно про операції порівняння у розділі 8.

(17 \* 13> 220)

ifTrue: ['bigger']

ifFalse: ['smaller'] → 'bigger'

Цикли теж реалізовані через відправку повідомлень блокам, цілочисельним або колекціям. Так як в циклах умови виходу можуть бути обраховані повторно, то вони сконструйовані за допомогою блоків а не булевих об'єктів. Нижче представлений приклад циклу whileTrue.

n: = 1.

[N <1000] whileTrue: [n: = n \* 2].

n → 1024

whileFalse - поки хибна умова триває цикл.

n: = 1.

[N> 1000] whileFalse: [n: = n \* 2].

n → 1024

timesRepeat: - повторити тіло циклу фіксовану кількість разів.

n: = 1.

10 timesRepeat: [n: = n \* 2].

n → 1024

Також можна надсилати повідомлення to: do: числу для створення циклу з параметром. Початкове значення циклу - одержувач, перший аргумент повідомлення - кінцеве значення циклу, другий - тіло циклу.

result: = String new.

1 to: 10 do: [: n | result: = result, n printString, ''].

result → '1 2 3 4 5 6 7 8 9 10'

**Ітератори вищих порядків**. Колекції складаються з великої кількості класів, багато з яких мають однаковий протокол. Більшість повідомлень для ітерацій складаються з: do, collect, select :, reject :, detect: і inject: into :. Ці повідомлення визначають ітерації вищих порядків, які дозволяють писати більш компактний код.

**Interval** - колекція, яка дозволяє ітерувати послідовність чисел від початкового числа і до кінцевого. 1 to: 10 реалізує інтервал від 1 до 10. Так як Interval є  
колекцією то можна послати повідомлення do :. Як аргумент використовується блок, який виконує певні дії для кожного елемента колекції.

result: = String new.

(1 to:10) do: [:n | result:= result, n printString, ''].

result → '1 2 3 4 5 6 7 8 9 10'

**collect**: створює нову колекцію такого ж розміру, трансформуючи кожен елемент.

(1 to: 10) collect: [: each | each \* each] → # (1 4 9 16 25 36 49 64 81 100)

**select**: і **reject**: створює нову колекцію, що включає в себе підмножину елементів, які задовольняють (чи ні) булевий блоку порівняння.

**detect**: повертає перший елемент, який задовольняє умову. Слід зауважити, що рядки теж є колекціями (символів), тому можна ітерувати по всіх символах.

'Hello there' select: [: char | char isVowel] → 'eoee'

'Hello there' reject: [: char | char isVowel] → 'hll thr'

'Hello there' detect: [: char | char isVowel] → $e

## 4.7 Примітиви і прагми.

У Pharo все є об'єктом і вся взаємодія між об'єктами здійснюється посилкою повідомлень. Проте певні об'єкти можуть викликати віртуальні машинні примітиви.

Наприклад, всі нижче наведені повідомлення реалізовані як примітиви:   
- виділення пам'яті (new, new :)  
- маніпуляції бітами (bitAnd :,ditOr :, ditShift :)  
- точки (Point) та целочисленная арифметика (+ - <> \* /= ==. . . )  
- доступ до масивів (at :, at: put).

Синтаксис при виклику примітиву такий <primitive: aNumber>. Метод який викликає примітив також може містити Smalltalk код, який отримає управління, якщо примітив завершитися помилкою.

Нижче показаний код методу SmallInteger >> +. Якщо примітив завершиться помилкою, то виконається наступна пропозиція super + aNumber.

+ ANumber

"Primitive. Add the receiver to 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: 1>

^ super + aNumber

У Pharo подібний синтаксис також використовується для анотацій методу - прагми.

## 4.8 Короткий огляд розділу.

* Pharo використовує тільки 6 зарезервованих ідентифікаторів, які називаються псевдозмінними: true, false, nil, self, super, thisContext.
* Є 5 видів літеральних обєктів: числа (5, 2.5, 1.9, е15, 2r111), літери ($a), рядки (‘hello’), символи (#abc) і масиви (#(‘hello’, #ab)).
* Рядки (String) поміщаються в одинарні лапки, коментарі у подвійні. Одинарні лапки всередині рядка дублюються.
* Подібні рядкам, сиволи унікальні у середовищі.
* Для визначення літерального масиву використовується #(…), а для динамічного масиву – {…}. Потрібно зауважити, що #(1 + 2) size -> 3, a {1 + 2} size -> 1.
* Повідомлення бувають трьох видів : унарні (1 asString, Array new), бінарні (1+2) та ключові (‘hi’ at: 1 put: #a).
* Каскадні повідомлення – послідовність повідомлень одному отримувачу, розділені крапкою з комою: OrderedCollection new add: #calvin; add: #hobbes; size -> 2.
* Локальні змінні поміщаються у вертикальні смуги. := використовується для присвоєння: |x| x:= 1.
* Вирази складаються з відправки повідомлень, каскадних повідомлень і присвоєнь, які обчислюються зліва на право. Також можна використовувати дужки для групування або зміни пріоритету.
* Блоки – вирази заключені у квадратні дужки. Можуть мати аргументи і тимчасові змінні. Вираз у блоці не виконується поки йому не буде відправлено повідомлення value з правильною кількість аргументів.  
  [:x | x + 2] value: 4 -> 6.
* Відсутній вбудований синтаксис для керуючих структур, все реалізовано через відправку повідомлень з використанням блоків.

[Розділ 1. Передмова 2](#_Toc507182797)

[1.1 Що таке Фаро 2](#_Toc507182798)

[1.2 Для кого ця книга 2](#_Toc507182799)

[1.3 Порада 3](#_Toc507182800)

[1.4 Відкрита книга 3](#_Toc507182801)

[1.5 Спільнота Фаро 3](#_Toc507182802)

[1.6 Приклади і вправи 4](#_Toc507182803)

[1.7 Подяки 4](#_Toc507182804)

[1.8 Спеціальні подяки 4](#_Toc507182805)

[Розділ 2. Короткий огляд Фаро 5](#_Toc507182806)

[2.1 Встановлення Pharo 5](#_Toc507182807)

[Завантаження Pharo 5](#_Toc507182808)

[Встановлення Pharo 5](#_Toc507182809)

[2.2 Файли компонентів Pharo 5](#_Toc507182810)

[Пара Image/Change 6](#_Toc507182811)

[Спільне встановлення 6](#_Toc507182812)

[2.3 Запуск Фаро 6](#_Toc507182813)

[Запуск Pharo з командного рядка 7](#_Toc507182814)

[2.4 Програма PharoLauncher 7](#_Toc507182815)

[2.5 Меню «World» 9](#_Toc507182816)

[Взаємодія з Pharo 9](#_Toc507182817)

[2.6 Надсилання повідомлень 11](#_Toc507182818)

[2.7 Зберігання, завершення та повторний запуск сесії Pharo 12](#_Toc507182819)

[2.8 Вікна Playground і Transcript 13](#_Toc507182820)

[2.9 Гарячі клавіші 13](#_Toc507182821)

[Виконання VS виведення 14](#_Toc507182822)

[Інспектування 14](#_Toc507182823)

[Інші дії 15](#_Toc507182824)

[2.10 Оглядач класів 16](#_Toc507182825)

[Відкривання Оглядача на заданому класі чи методі 16](#_Toc507182826)

[Мандрівка системою за допомогою Оглядача класів 17](#_Toc507182827)

[2.11 Відшукання класів 18](#_Toc507182828)

[Використання повідомлення *browse* 18](#_Toc507182829)

[Використання *CMD-b* для перегляду 18](#_Toc507182830)

[Використання Сищика (*Spotter*) 19](#_Toc507182831)

[Перехід за результатами пошуку 19](#_Toc507182832)

[Використання команди «*Find class*» в Системному оглядачі 20](#_Toc507182833)

[Використання засобу пошуку *Finder* 20](#_Toc507182834)

[2.12 Відшукання методів 20](#_Toc507182835)

[Spotter 21](#_Toc507182836)

[Finder 21](#_Toc507182837)

[Пошук методів за зразками 21](#_Toc507182838)

[2.13 Визначення нового методу 21](#_Toc507182839)

[Визначення нового тестового методу 22](#_Toc507182840)

[Виконання тестового методу 23](#_Toc507182841)

[Створення тестованого методу в Налагоджувачі 24](#_Toc507182842)

[2.14 Підсумки розділу 25](#_Toc507182843)

[Розділ 3. Перший застосунок 25](#_Toc507182844)

[3.1 Гра Lights Out 26](#_Toc507182845)

[3.2 Створення нового пакету класів 26](#_Toc507182846)

[3.3 Визначення класу LOCell 27](#_Toc507182847)

[Створення нового класу 27](#_Toc507182848)

[Про коментарі 28](#_Toc507182849)

[Про категорії та пакети 28](#_Toc507182850)

[3.4 Додавання методів до класу 29](#_Toc507182851)

[3.5 Інспектування об’єкта 30](#_Toc507182852)

[3.6 Визначення класу LOGame 32](#_Toc507182853)

[Створення класу 32](#_Toc507182854)

[Ініціалізація гри 32](#_Toc507182855)

[Використання переваг налагоджувача 32](#_Toc507182856)

[Вивчаємо метод *initialize* 34](#_Toc507182857)

[3.7 Поділ методів на протоколи 35](#_Toc507182858)

[3.8 Завершення розробки гри 36](#_Toc507182859)

[3.9 Випробуємо код 38](#_Toc507182860)

[3.10 Удосконалення гри 39](#_Toc507182861)

[Зміна методу ініціалізації 40](#_Toc507182862)

[Зміна способу запуску 41](#_Toc507182863)

[3.11 Зберігання та поширення коду Pharo 44](#_Toc507182864)

[Збереження простого тексту 45](#_Toc507182865)

[Пакети Monticello 45](#_Toc507182866)

[Менеджер Monticello 45](#_Toc507182867)

[Збереження та завантаження коду за допомогою Monticello 46](#_Toc507182868)

[SmalltalkHub: Github для Pharo 46](#_Toc507182869)

[3.12 Підсумки розділу 48](#_Toc507182870)

[Розділ 4. Синтаксис у двох словах 48](#_Toc507182871)

[4.1 Синтаксичні елементи 48](#_Toc507182872)

[4.2 Псевдо змінні. 50](#_Toc507182873)

[4.3 Відправка повідомлення. 51](#_Toc507182874)

[4.4 Синтаксис методу. 52](#_Toc507182875)

[4.5 Синтаксис блока. 52](#_Toc507182876)

[4.6 Порівняння і цикли у двох словах. 53](#_Toc507182877)

[4.7 Примітиви і прагми. 54](#_Toc507182878)

[4.8 Короткий огляд розділу. 55](#_Toc507182879)

Проблеми:

1. «Мишачі» комбінації клавіш не працюють
2. Не працює #ref у Spotter
3. !!! Розповісти про методи класу в Оглядачі

1. Alec Sharp, Smalltalk by Example. McGraw-Hill, 1997 〈URL: <http://stephane.ducasse.free.fr/FreeBooks/ByExample/>〉. [↑](#footnote-ref-1)
2. <http://alanknightsblog.blogspot.com/2011/10/principles-of-oo-design-or-everything-i.html> [↑](#footnote-ref-2)
3. Class Responsibility Collaborators – «Клас Відповідальність Взаємодія», метод аналізу при проектуванні об’єктно-орієнтованого програмного забезпечення [↑](#footnote-ref-3)
4. Цей параграф написав перекладач книги. [↑](#footnote-ref-4)