Розділ 5

Перший практикум. Розробка простого лічильника

Давайте для початку створимо у Pharo простий лічильник згідно з описаними далі вказівками. Ви навчитеся створювати пакети, класи, методи, екземпляри, модульні тести та ще дещо. Цей практикум охопить більшість важливих дій, які доводиться виконувати під час розробки у Pharo. Ви можете також переглянути відео, що ілюструють цей приклад, доступні в онлайновому курсі на http://mooc.pharo.org. Їхні назви мають формат *yy-Redo-xxx*. Ми покажемо також як за допомогою Iceberg зберегти свій код у хмарному сховищі, наприклад, у GitHub.

Зауважимо, що цей невеликий практикум описує *традиційний* процес розробки. Традиційний у тому сенсі, що спочатку ви створите пакет і оголосите клас, *тоді* оголосите змінні екземпляра, *тоді* визначите його методи і *лише тоді* виконаєте їх. Нині розробники зазвичай використовують у Pharo інший процес, який називається *розробка, керована тестами* (Test-Driven Development, TDD): вони *спочатку* виконують вираз. Відповідні методи ще не оголошені, тому вираз спричиняє помилку. Налагоджувач перехоплює її, і розробник програмує безпосередньо в налагоджувачі, дозволяючи системі визначити змінні екземпляра та методи на льоту, щоб виправити помилку.

Після закінчення практикуму ви ліпше освоїтеся з Pharo, тому ми наполегливо рекомендуватимемо вам виконати його ще раз з використанням TDD: напишіть модульні тести, запустіть їх, напишіть код, щоб тести проходили, знову запустіть тести. У Pharo можна застосовувати екстремальне програмування, кероване тестами, Extreme TDD: напишіть тести, запустіть їх, напишіть код у налагоджувачі, щоб тести пройшли. Екстремальне програмування – надпотужний підхід. Щоб відчути це, потрібно випробувати його власноруч. У тому ж онлайновому курсі є ще одне відео, що демонструє цей потужний спосіб написання програм. Ми справді закликаємо вас переглянути його та попрактикуватися.

5.1. Завдання

Ми хочемо мати змогу створити лічильник, збільшити його двічі (на одиницю), зменшити і переконатися, що він має очікуване значення – дорівнює одиниці. Наступний фраґмент коду описує сказане програмно. Він також стане чудовим модульним тестом, який визначимо трохи згодом.

```
| counter |
counter := Counter new.
counter increment; increment.
counter decrement.
counter count = 1
```

Ми напишемо всі класи і методи, потрібні для реалізації цього прикладу.

5.2. Створення пакета і класу

У цьому параграфі ви створите свій перший клас. У Pharo класи оголошують у пакетах, тому мусимо спочатку створити пакет, щоб помістити в нього клас. Такі дії виконують кожного разу при створенні класу, тому будьте уважні.

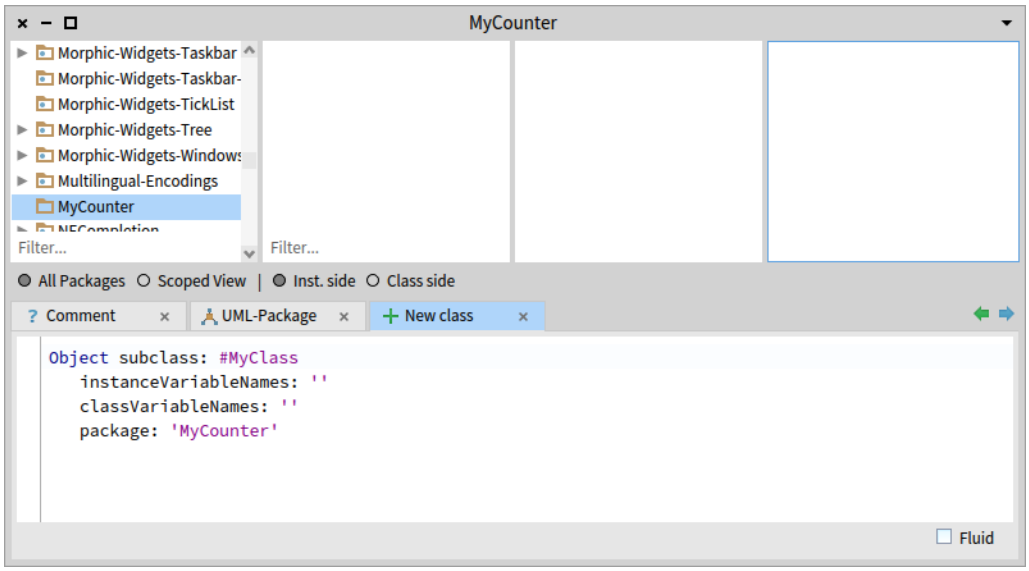

Рис. 5.1. Новостворений пакет і шаблон оголошення класу

Створення пакета

Пакети створюють за допомогою Системного оглядача: контекстно клацніть у його панелі пакетів і виберіть команду «*New Package*». Оглядач запитає у вас ім'я пакета – введіть «*MyCounter*». Одразу після створення новий пакет з'явиться у списку пакетів, його ім'я буде позначено. На рис. 5.1 зображено, що мало б вийти в результаті.

Створення класу

Нижня панель Оглядача мала б містити відкриту вкладку ***New class*** з шаблоном оголошення класу. Щоб створити новий клас, потрібно просто відредагувати шаблон і відкомпілювати написане. У шаблоні є п'ять частин, які можна змінювати.

- **Специфікація надкласу** задає базовий клас для того, який ви створюєте. За замовчуванням тут вказано *Object* – найбільш загальний з усіх класів Pharo, саме той, що потрібен нам для створення класу лічильника. Так буде не завжди: часто ви потребуватимете оголошувати класи на базі більш конкретизованих.
- **Ім'я класу.** Далі ви мали б вказати ім'я класу: замініть *#MyClass* на *#Counter*. Зверніть увагу на те, що ім'я класу починається з великої букви, а перед іменем обов'язково вказують знак #. Така форма запису зумовлена тим, що у Pharo клас називають за допомогою символу, екземпляра класу *Symbol*, унікального значення, зображеного рядком. Символи завжди починаються знаком #.
- **Специфікація змінних екземпляра.** Тепер вам потрібно заповнити імена змінних екземпляра класу *Counter* після слів *instanceVariableNames:*. Нам потрібно тільки одну змінну, що називається '*count*'. Не забудьте апострофи (одинарні лапки)! Перелік імен змінних задають рядком, екземпляром класу *String*, що містить імена, відокремлені пропуском. Рядки завжди обрамляють апострофами.

- **Специфікація змінних класу.** Їх оголошують після *classVariablesNames:*. Залиште тут порожній рядок (пару апострофів), як вказано у шаблоні, оскільки нам не потрібні змінні класу.
- **Специфікація пакета.** Вона вже містить правильне значення '*MyCounter*', яке ми не будемо зачіпати.

Ви мали б отримати таке оголошення класу:

```
Object subclass: #Counter
   instanceVariableNames: 'count'
   classVariableNames: ''
   package: 'MyCounter'
```

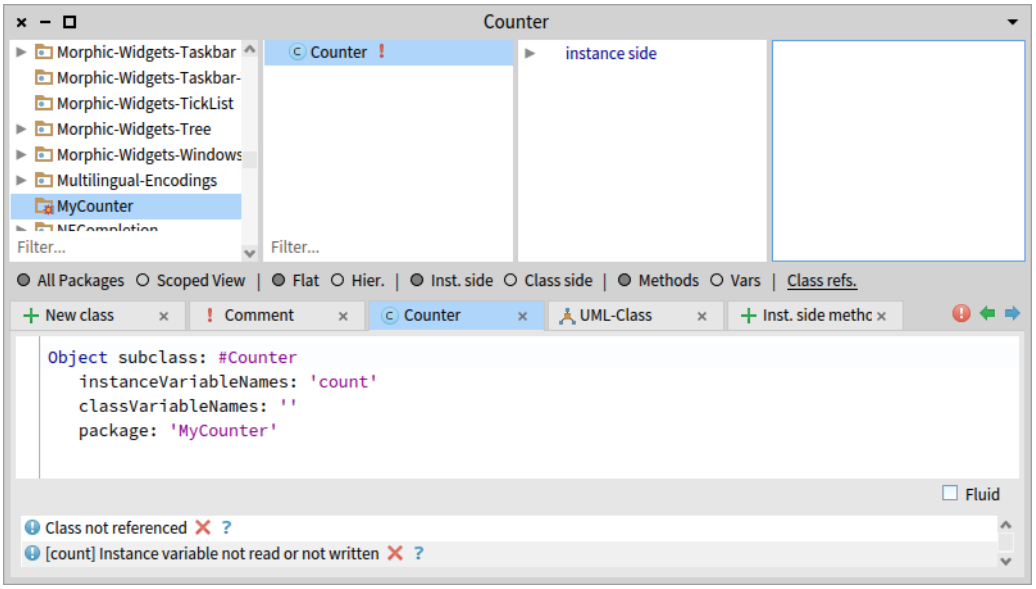

Рис. 5.2. Створено клас *Counter*, що наслідує *Object* і має одну змінну екземпляра *count*

Тепер ми маємо текст оголошення класу *Counter*. Щоб визначити клас у нашій системі, потрібно його *відкомпілювати* або через контекстне меню нижньої панелі командою «*Accept*», або комбінацією клавіш [*Cmd* + *S*]. (Скорочення від «*Save*»: зберігання оголошення класу означає його компіляцію). Після завершення компіляції клас одразу стане частиною системи.

На рис. 5.2 показано, як мав би виглядати Системний оглядач після компіляції.

Вбудований у Pharo інструмент аналізу якості коду запуститься автоматично і виведе кілька попереджень унизу вікна Оглядача. Поки що не турбуйтеся про них. Причина їхньої появи в тому, що наш клас ще ніде не використовується.

Ми з вами дисципліновані розробники, тому маємо додати до класу *Counter* коментар, що пояснює його призначення. Для цього відкрийте вкладку ***Comment*** нижньої панелі Оглядача та клацніть на перемикачі ***Toggle Edit/View comment***. Тепер ви можете замість стандартного тексту ввести, наприклад, такий коментар:

```
`Counter` is a simple concrete class which supports incrementing and
        decrementing.
Its API is
- `decrement` and `increment`
- `count`
Its creation message is `startAt:`
```

Для написання коментарів використовують розмітку *Microdown*, інтуїтивно зрозумілий діалект *Markdown*. Розмічений коментар гарно відображається у вікні оглядача. Знову збережіть зроблені вами зміни командою «*Accept*», або комбінацією [*Cmd* + *S*].

На рис. 5.3 зображено створений коментар до класу *Counter*.

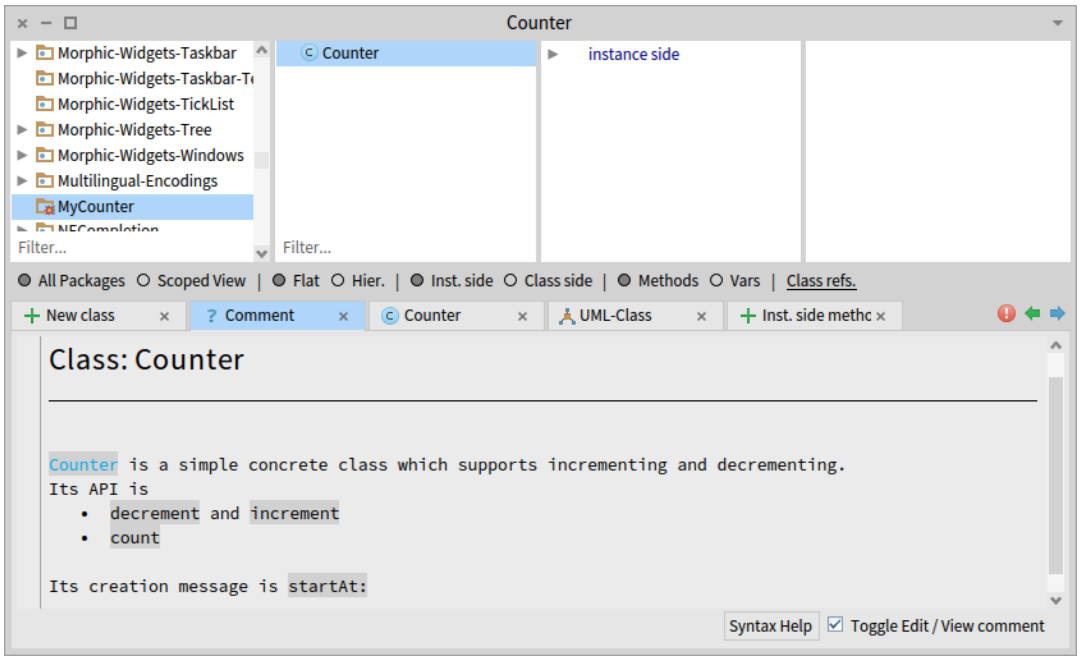

Рис. 5.3. Гарна робота: клас *Counter* тепер має коментар!

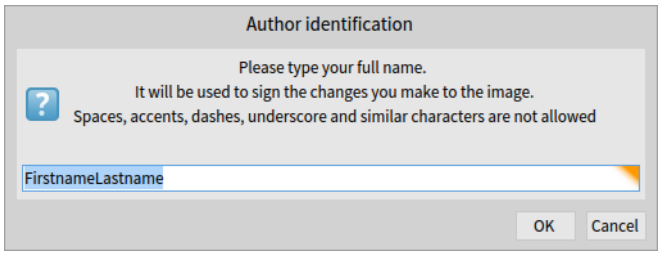

Рис. 5.4. Ідентифікація автора змін

Від перекладача. Pharo автоматично документує всі зміни, зроблені в бібліотеці класів. Під час першого зберігання зроблених вами змін (у тексті методу, в коментарі класу чи в оголошенні класу) система попросить назвати себе. Введіть своє ім'я та прізвище латинкою без пропусків, як у підказці в рядку введення діалогу ідентифікації, зображеному на рис. 5.3. Достатньо зробити це один раз. Збережіть імідж системи, і під час наступних сеансів розробки у Pharo збережений код автоматично підписуватиметься введеним іменем.

5.3. Визначення протоколів і методів

У цьому параграфі ви навчитеся додавати за допомогою Оглядача протоколи й методи.

Ми визначили клас *Counter* з однією змінною екземпляра, яка називається *count* і призначена для зберігання рахунка. Ми хочемо збільшувати, зменшувати та демонструвати її поточне значення. Але у Pharo ми повинні пам'ятати три речі.

1. Усе є об'єктами.
2. Змінні екземпляра є цілком приватними для об'єкта.

3. Єдиний спосіб взаємодіяти з об'єктом – надсилати йому повідомлення.

Саме тому не існує іншого механізму доступу ззовні до змінної екземпляра нашого лічильника, як через надсилання повідомлень об'єктові. Нам потрібно оголосити метод, який повертатиме значення змінної екземпляра. Такі методи називають *методами читання* або *селекторами* (англійською – *getter*). Отже, давайте визначимо метод доступу до змінної екземпляра *count*.

Зазвичай метод поміщають у *протокол*. Протоколи в класі – просто групи методів. Вони не мають синтаксичного значення у Pharo, але надають читачам вашого класу важливу інформацію. Хоча протоколи можна називати довільно, проте розробники у Pharo дотримуються певних домовленостей щодо найменування протоколів. Якщо ви визначаєте метод і не впевнені, до якого протоколу його зачислити, перегляньте спочатку наявний код, чи не знайдеться потрібний протокол серед уже оголошених.

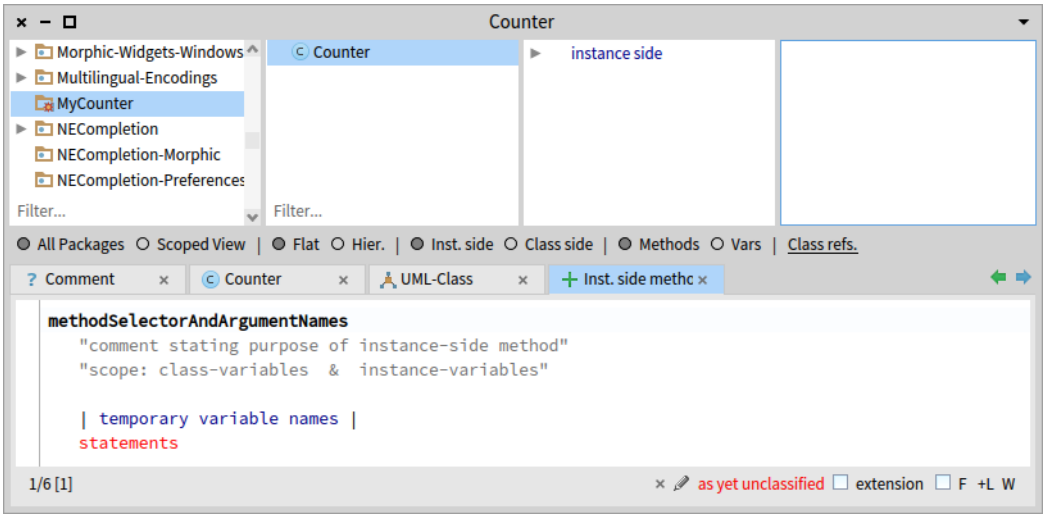

Рис. 5.5. Редактор коду готовий до визначення методу

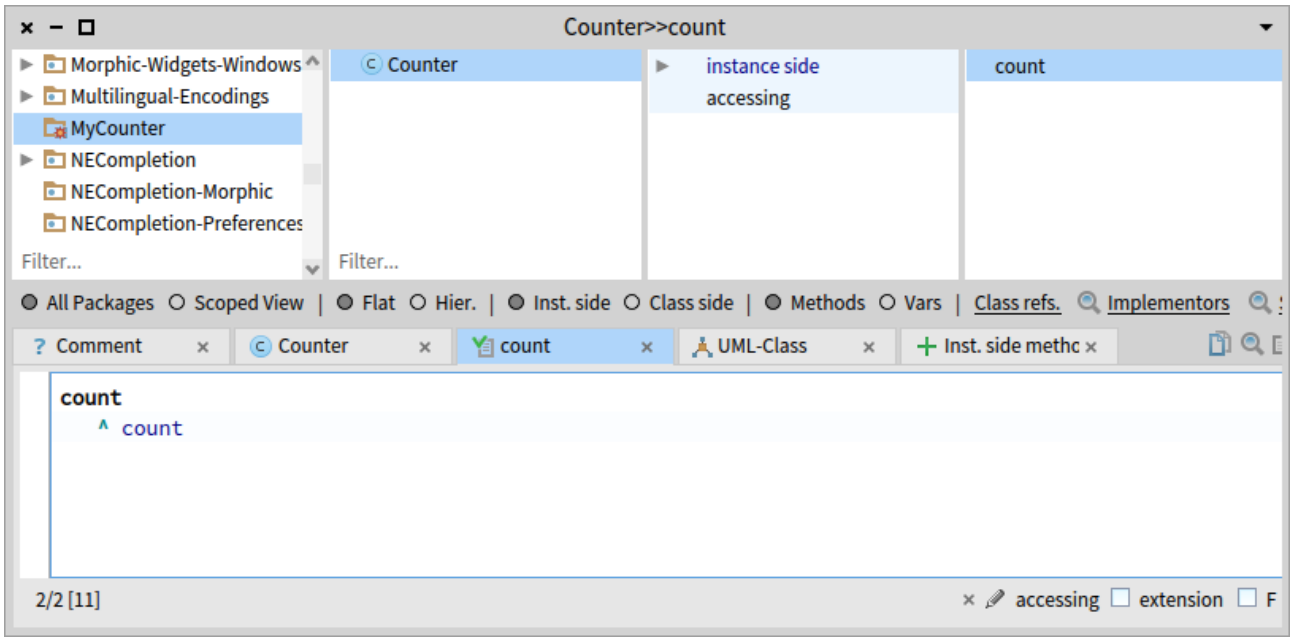

Рис. 5.6. Метод *count* визначено в протоколі *accessing*

5.4. Створення методу

Тепер давайте створимо метод для читання змінної екземпляра *count*. Почнімо з вибору класу *Counter* в Оглядачі та переконаймося, що ми редагуємо сторону екземпляра класу (тобто визначаємо методи для екземплярів нашого класу): посередині вікна Оглядача розташовано перемикач ***Inst. side/Class side***, позначеною має залишатися ліва частина. Далі оберіть вкладку ***Inst. side method*** та визначимо метод.

На рис. 5.5 показано редактор коду, готовий до визначення методу. Він містить шаблон оголошення методу. Нам потрібно замінити його справжнім оголошенням. Щоб позначити весь текст шаблону, двічі клацніть перед його початком або після закінчення. (Так можна позначати весь текст у будь-якому вікні). Почніть набирати текст, і виділення автоматично заміниться на введене.

Надрукуйте таке визначення методу:

```
count
   ^ count
```

Воно визначає метод *count*, що не приймає аргументів і повертає значення змінної екземпляра *count*. Щоб відкомпілювати метод, виберіть «*Accept*» з контекстного меню. Метод буде автоматично віднесено до протоколу *accessing*.

На рис. 5.6 зображено Оглядача після визначення методу.

Тепер ви можете випробувати новий метод, просто надрукуйте в Робочому вікні та виконайте вираз:

```
Counter new count
>>> nil
```

Цей вираз спочатку створить новий екземпляр класу *Counter*, а тоді надішле йому повідомлення «*count*», яке поверне поточне значення лічильника. Воно має повернути *nil* – значення за замовчуванням неініціалізованої змінної екземпляра. Пізніше ми створюватимемо екземпляри з розумнішим початковим значенням.

5.5. Додавання методу запису значення

Доповненням до методу читання є *метод запису* або *модифікатор* (*setter* англійською). Його використовують для того, щоб ззовні об'єкта змінити значення його змінної. Наприклад, вираз «*Counter new count: 7*» спочатку створить новий екземпляр класу *Counter*, а тоді зробить сім його значенням за допомогою повідомлення «*count: 7*». Селектори та модифікатори разом називають *методами доступу* (*accessors*).

Приклад використання модифікатора на практиці:

```
| c |
c := Counter new count: 7.
c count
>>> 7
```

Метод запису поки що не існує, тому як вправу створіть його. Метод *count:* має приймати один аргумент, число, і записувати його у змінну екземпляра. Метод можна випробувати в Пісочниці, виконавши наведений вище приклад.

Підказка: модифікатор може мати такий вигляд:

```
count: anInteger
   count := anInteger
```

Після компіляції модифікатор також мав би потрапити до протоколу *accessing*.

5.6. Визначення класу тестів

У наші дні написання тестів перестало бути необов'язковим заняттям. Ви можете писати їх перед створенням коду чи після, але писати треба обов'язково. Колекція добре продуманих тестів підтримуватиме розвиток вашого застосунку і даватиме впевненість, що ваша програма робить саме те, чого ви від неї очікуєте. Написання тестів є хорошою інвестицією. Один раз написаний код тестів виконується тисячі разів. Наприклад, якщо ми перетворимо на тест наведений раніше приклад, то зможемо автоматично перевірити, чи працює новий модифікатор так, як треба.

Модульні тести пишуть у вигляді методів окремого класу, успадкованого від *TestCase*. Тому ми визначимо клас *CounterTest* так:

```
TestCase subclass: CounterTest
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'MyCounter'
```

Традиційно ім'я класу тестів складається з імені тестованого класу і суфікса *Test*: *Counter* + *Test* дає *CounterTest*.

Тепер можемо написати свій перший тест як метод цього класу. Імена тестових методів мають розпочинатися з «*test*», щоб система правильно розпізнала їх і наділила спеціальними можливостями: тестові методи можна автоматично виконати за допомогою спеціального інструмента – *Test Runner*; Оглядач розташовує перед іменем тестового методу круглу піктограму, яка здатна змінювати колір, а клацання на ній запускає метод на виконання.

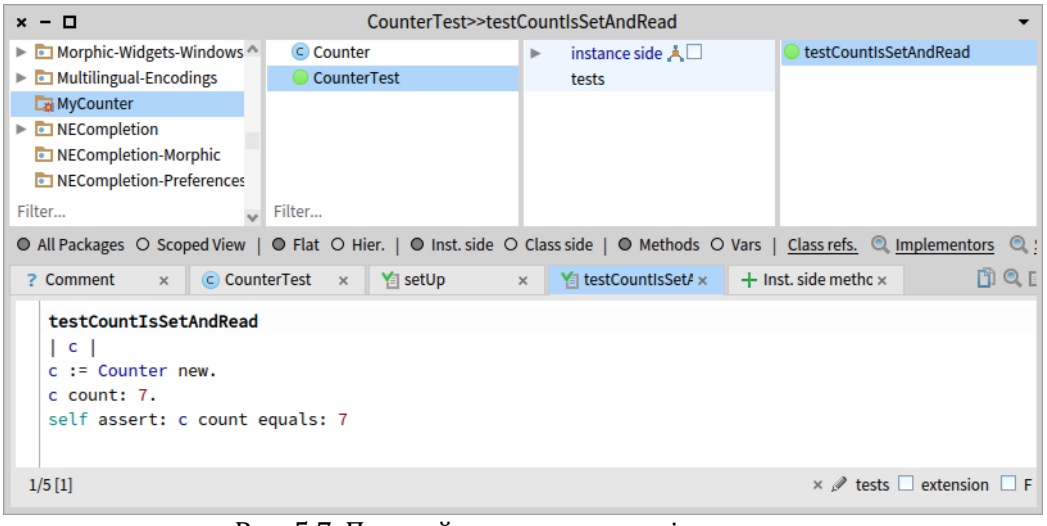

Рис. 5.7. Перший тест визначено і виконано

Визначимо метод для нашого модульного тесту. Найперше він створює екземпляр класу *Counter*, задає його значення, а тоді перевіряє чи містить він задане значення.

Повідомлення «*assert:equals:*» зрозуміле всім нащадкам *TestCase*. Воно перевіряє істинність твердження про те, що два об'єкти рівні між собою. Якщо це не так, то тест завершиться невдачею.

```
CounterTest >> testCountIsSetAndRead
  | c |
  c := Counter new.
  c count: 7.
  self assert: c count equals: 7
```

На рис. 5.7 показано визначення методу *testCountIsSetAndRead* в класі *CounterTest*.

Нагадаємо домовленості щодо позначень методів. Pharo-розробники часто використовують запис «*ClassName >> methodName*», щоб вказати клас, до якого належить метод. Наприклад, раніше ми визначили метод *count* у класі *Counter*. Посилання на нього матиме вигляд «*Counter >> count*». Пам'ятайте, що такий запис *не є* частиною синтаксису Pharo. Просто ми домовилися так записувати те, що «метод екземпляра *count* належить до класу *Counter*».

Надалі будемо записувати імена методів у такій формі щоразу, коли згадуватимемо їх у цій книзі. Зрозуміло, що вам не потрібно друкувати ім'я класу чи знак «>>», коли ви вводитимете текст методу в редакторі коду. Натомість переконайтеся, що на панелі класів Оглядача вибрано належний клас.

Щоб запустити тестовий метод на виконання та переконатися, що тест завершується успіхом, клацніть на іконці ліворуч від імені методу (див. рис. 5.7) або використайте *Test Runner* через меню *World* > *Browse*.

Ви щойно отримали перший зелений тест і добру нагоду зберегти свої напрацювання.

5.7. Зберігання коду в git-репозиторії за допомогою Iceberg

Зроблене можна зберігати в іміджі Pharo. Це хороший спосіб, проте не ідеальний. Він не підходить для налагодження співпраці з іншими розробниками та для поширення коду. Багато сучасних розробників програмного забезпечення взаємодіють через Git, відкриту розподілену систему керування версіями файлів. Збудовані із застосуванням Git сервіси, такі як GitHub, надають розробникам простір для спільного виконання проєктів з відкритим вихідним кодом таких, наприклад, як Pharo.

Pharo взаємодіє з Git за допомогою окремого інструмента – *Iceberg*. Цей параграф продемонструє вам, як створити локальний репозиторій для програмного коду, заносити в нього зміни та переносити їх до віддаленого репозиторію, розташованого, наприклад, на GitHub.

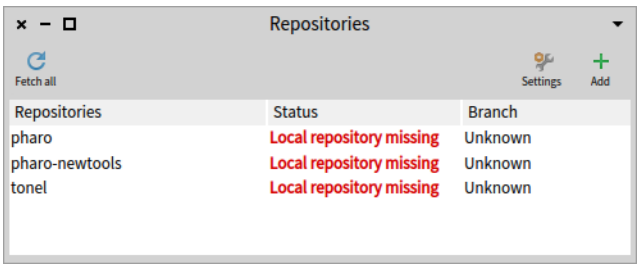

Рис. 5.8. Оглядач репозиторіїв інструмента Iceberg, відкритий на свіжому образі Pharo показує проєкти, для яких не знайдено локальні сховища. Їх треба буде створити, якщо ви захочете змінити версію самого Pharo, але це не зараз

Відкрийте Iceberg

Відкрийте Iceberg через меню *World* > *Sources* або комбінацією клавіш [*Cmd* + *O,I*].

Ви мали б побачити щось схоже на зображене на рис. 5.8 – головне вікно Iceberg. Воно містить проєкт Pharo та деякі інші, отримані разом з образом, та повідомляє рядком «*Local repository missing*», що не може знайти їхні локальні репозиторії. Ви можете не турбуватися про проєкт Pharo та його сховище, якщо не збираєтеся брати участі в його розробці.

Ми плануємо створити свій власний новий проєкт.

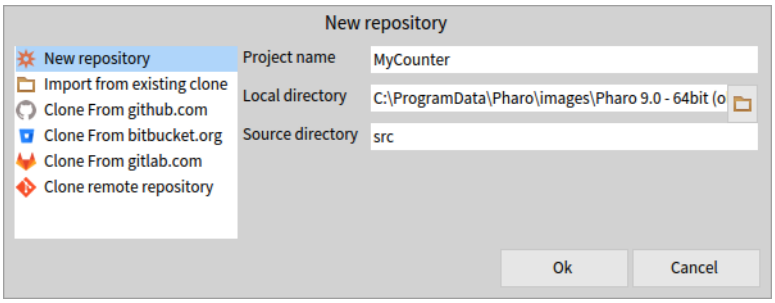

Рис. 5.9. Створення нового проєкту, що називається *MyCounter* та містить підкаталог *src*

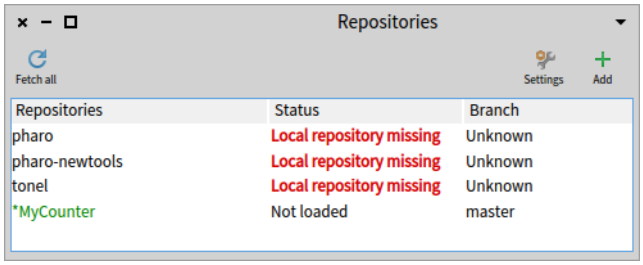

Рис. 5.10. Новостворений проєкт з незбереженими змінами

Додайте та налаштуйте проєкт

Щоб створити новий проєкт, натисніть кнопку ***Add***. Вона відкриє вікно налаштування проєкту (див. рис. 5.9). У списку ліворуч буде обрано рядок «*New repository*» (якщо ні, то оберіть його), а нам залишиться тільки задати ім'я проєкту, уточнити папку для його зберігання та вказати ім'я вкладеної папки для програмного коду. За домовленістю вона називається «*src*».

Натисніть кнопку «*Ok*» і діалог закриється, а в головному вікні Iceberg з'явиться створений проєкт, як на рис. 5.10.

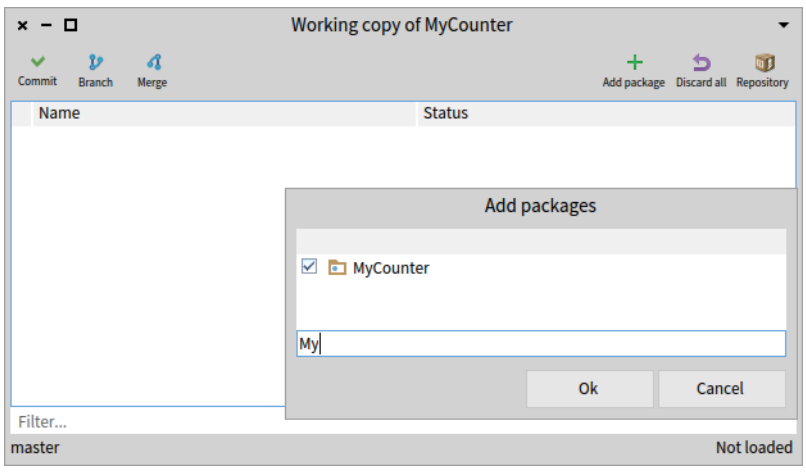

Рис. 5.11. Додавання пакета до проєкту кнопкою ***Add package***

Додайте до проєкту пакет

Вміст проєкту можна переглянути в оглядачі робочої копії репозиторію інструмента Iceberg. Щоб відкрити його, двічі клацніть на рядку з проєктом *MyCounter* у головному вікні Iceberg. Щойно створений проєкт не містить пакетів, тому ви побачите порожнє вікно, як на рис. 5.11. Клацніть на кнопці ***Add package*** та виберіть пакет *MyCounter* на модальній панелі, що відкриється (його легко відшукати серед багатьох пакетів за допомогою фільтра).

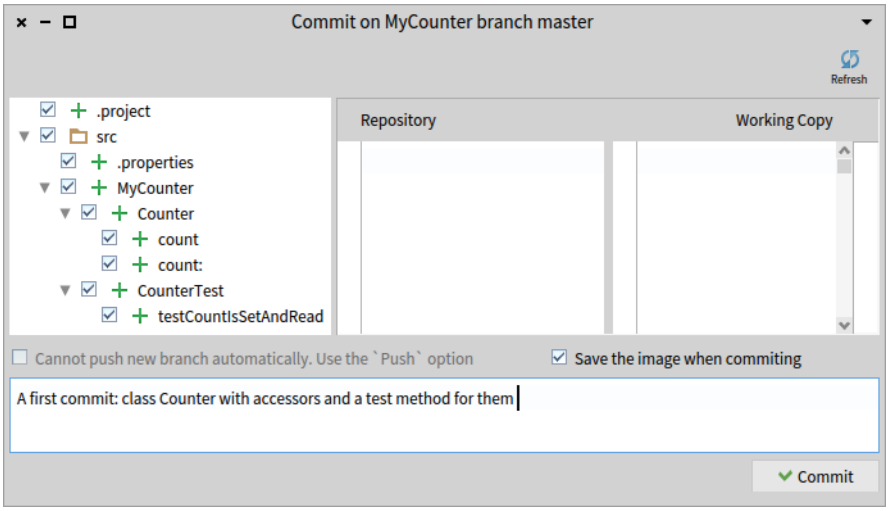

Рис. 5.12. Iceberg показує зміни, які належить перенести до сховища

Збережіть пакет

Одразу після додавання до проєкту пакет матиме статус «*Uncommitted changes*». Так Iceberg повідомляє нам, що код в пакеті відрізняється від того, що є у сховищі. Так і має бути, адже в сховищі ще немає коду! Збережіть його, клацнувши кнопку ***Commit***. Iceberg покаже вам у новому вікні всі зміни, які потрібно зберегти (див. рис. 5.12). Уведіть коротке пояснення щодо змісту змін і перенесіть їх до сховища ще однією кнопкою ***Commit***.

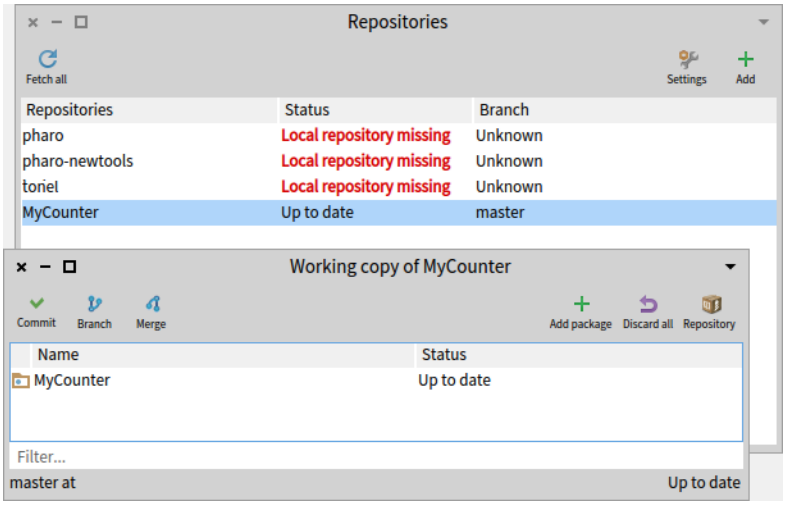

Рис. 5.13. Iceberg підтверджує, що зміни збережено

Код збережено

Як тільки ви завершите зберігання, Iceberg відзначить, що ваша система і локальний репозиторій синхронізовані (рис. 5.13).

Гарна робота! Незабаром ми побачимо, як перенести зміни коду до віддаленого сховища, а зараз повернемося до нашого класу *Counter*.

5.8. Додавання нових методів

Перш ніж додати до класу *Counter* наступні методи, напишемо тести, що їх перевіряють. Почнемо з тесту для повідомлення *increment*.

```
CounterTest >> testIncrement
  | c |
  c := Counter new.
  c count: 0; increment; increment.
  self assert: c count equals: 2
```

Тепер ваша черга! Напишіть визначення методу *increment* так, щоб цей тест завершився успіхом. Коли закінчите, спробуйте написати тест для повідомлення *decrement* і визначте в класі *Counter* метод, що задовольнить його. Нові методи розташуйте в протоколі *operations*.

Розв'язок

```
Counter >> increment
  count := count + 1

Counter >> decrement
  count := count - 1
```

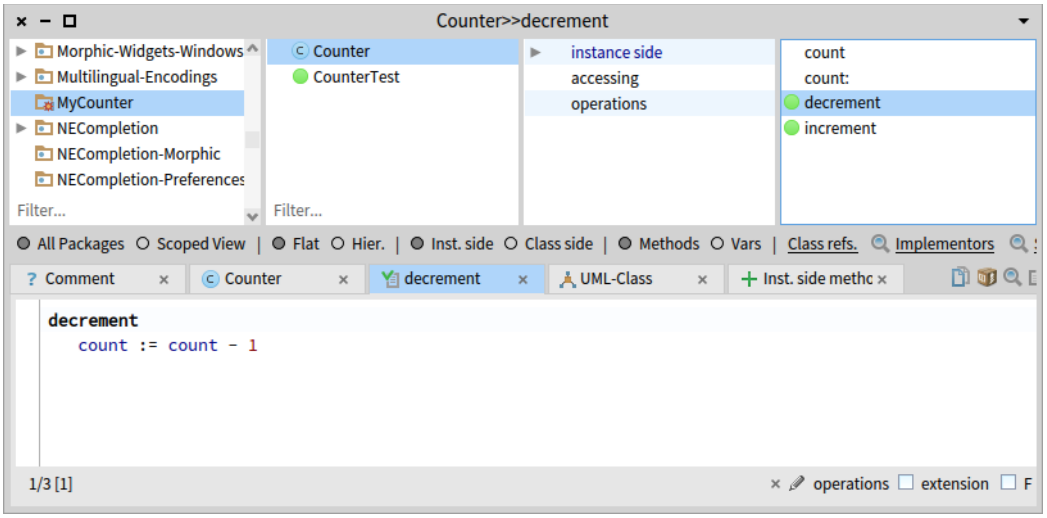

Рис. 5.14. У класу *Counter* стало більше методів і зелених тестів

Чи зауважили ви, що тест для методу *increment* називається *testIncrement*? Це не випадково, адже такий спосіб найменування дає змогу середовищу пов'язувати метод з його модульним тестом. Якщо ви все зробили правильно, то біля імен методів *increment* та *decrement* в Оглядачі класів з'являться такі ж піктограми, як біля методів-тестів (див. рис. 5.14). Клацніть на кожній з них, і ви запустите тести на виконання.

Ваші тести мали б завершитися успіхом. І знову було б доречно зберегти свою роботу. Збереження у ті моменти, коли всі тести зелені, є хорошою практикою. Щоб зберегти зміни, занесіть їх до сховища за допомогою Iceberg.

5.9. Метод ініціалізації екземпляра

На цей момент початкове значення нашого лічильника залишається невизначеним. У цьому легко переконатися, виконавши вираз:

```
Counter new count
>>> nil
```

Давайте напишемо тест, який стверджує, що новий екземпляр класу *Counter* містить значення 0 у змінній *count*:

```
CounterTest >> testInitialize
  self assert: Counter new count equals: 0
```

Клацніть на піктограмі методу, щоб запустити його на виконання, і отримаєте вікно з повідомленням «*Got nil instead of 0*» (отримано *nil* замість *0*). Це вікно можна поки що закрити, але зауважте, що піктограма стала *жовтою*. Це ознака того, що тест завершився невдачею: усе було виконано, проте твердження виявилося хибним. Така ситуація відрізняється від отримання *червоного* тесту, коли тест не виконується через помилку, наприклад, якщо потрібний метод не визначено. Незабаром ми побачимо і такі.

5.10. Визначення методу ініціалізації

Тепер потрібно написати метод ініціалізації, який задає початкове значення змінної *count* екземпляра лічильника. Ми вже знаємо, що екземпляри створюють надсиланням класові повідомлення *new*. Процес влаштовано так, що кожен новостворений об'єкт автоматично отримує повідомлення *initialize*. Це дає нагоду екземплярові налаштувати свій початковий стан. Визначимо метод *initialize*, який задаватиме правильне початкове значення лічильника.

Повідомлення *initialize* надсилають екземплярові, тому відповідний метод потрібно визначити на *стороні екземпляра* так само, як інші методи, що опрацьовують повідомлення до екземпляра (такі як *increment* чи *decrement*). Метод *initialize* відповідає за налаштування значень за замовчуванням змінних екземпляра.

Отже, визначте в класі *Counter* на стороні екземпляра у протоколі *initialization* зображений нижче метод (тіло методу поки що порожнє, заповніть його самостійно).

```
Counter >> initialize
  "Set the initial value of the count to 0"

  "Тут має бути ваш код"
```

Якщо ви все зробили правильно, то тест *testInitialize* тепер завершиться успіхом. Як завжди, збережіть свою роботу перш ніж рухатися далі.

5.11. Визначення методу створення нового екземпляра

Щойно ми зазначили, що метод *initialize* визначають *на стороні екземпляра*, бо його виконує новостворений екземпляр класу *Counter*. Метод *initialize* є методом екземпляра і змінює значення його полів. Давайте тепер розглянемо визначення методу *на стороні класу* – методу класу. Метод класу виконується тоді, коли повідомлення отримує сам клас, а не його екземпляри (таким повідомленням, наприклад, є *new*). Щоб визначити

метод класу, потрібно перемкнути Оглядач класів на сторону класів. Увімкніть для цього перемикач ***Class side***.

Визначте новий метод для створення екземплярів, що називається *startingAt:*. Він отримує ціле число як аргумент і повертає новий екземпляр класу *Counter*, який у змінній *count* містить задане число.

З чого почати? Звичайно ж, з тесту.

```
TestCounter >> testCounterStartingAt5
  self assert: (Counter startingAt: 5) count equals: 5
```

Тут повідомлення *startingAt: 5* надсилають самому класові *Counter*.

Ваша реалізація методу може виглядати якось так:

```
Counter class >> startingAt: anInteger
  ^ self new count: anInteger.
```

Тут у тексті ми використали позначення вигляду «*ClassName class >> methodName*», щоб вказати на *метод класу*. Перший рядок коду означає «*startingAt:* є методом класу у класі *Counter*».

На що вказує *self* у наведеному коді? Як завжди, *self* вказує на об’єкт, що виконує метод, і тому тут він вказує на сам клас *Counter*.

Давайте напишемо ще один тест, щоб просто переконатися, що все працює належно.

```
CounterTest >> testAlternateCreationMethod
  self assert: ((Counter startingAt: 19) increment; count) equals: 20
```

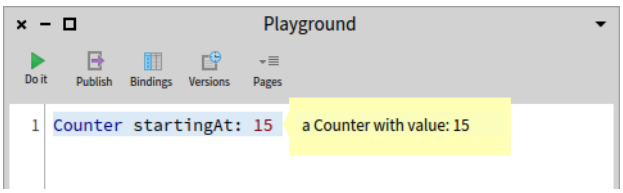

Рис. 5.15. Інформативне представлення лічильника

5.12. Покращення опису об’єкта

Коли ви інспектуватимете екземпляр *Counter* або Налагоджувачем, або Інспектором, відкривши його командою [*Cmd* + *I*] на виразі «*Counter new*», або, навіть, коли ви просто виконаєте команду «*Print it*» на виразі «*Counter new*», отримаєте дуже спрощене зображення свого лічильника: лише '*a Counter*':

```
Counter new
>>> a Counter
```

Ми б хотіли мати більш інформативне зображення, наприклад, таке, що показує значення лічильника. Реалізуйте наступний метод у протоколі *printing* класу *Counter*:

```
Counter >> printOn: aStream
  super printOn: aStream.
  aStream nextPutAll: ' with value: '.
  count printOn: aStream
```

Зауважте, що повідомлення *printOn:* отримує кожен об'єкт, який друкують командою «*Print it*» (див. рис. 5.15), або який відкривають у Інспекторі. За допомогою визначення методу *printOn:* для екземпляру класу *Counter* ми власноруч задаємо спосіб відображення лічильника на екрані. Ми *перевизначаємо* успадкований метод *Object >> printOn:*, що діє за замовчуванням, і який до тепер виконував усю роботу. Пізніше детальніше розглянемо успадкування та перевизначення, а також вивчимо використання потоків і змінної *super*.

Модульний тест для методу *printOn:* визначте самі. Підказка: щоб отримати зображення лічильника у вигляді рядка, таке ж, яке створює метод *printOn:*, надішліть повідомлення «*printString*» до «*Counter new*».

```
Counter new printString
>>> a Counter with value: 0
```

Давайте знову збережемо свою роботу, але цього разу – на віддаленому Git-сервері.

5.13. Зберігання коду на віддаленому сервері

До тепер ви зберігали свій код на локальному диску комп'ютера. Зараз ми покажемо, як його зберегти у віддаленому репозиторії, який ви можете створити на GitHub https://github.com або GitLab.

Створіть проєкт на віддаленому сервері

Спочатку вам треба створити репозиторій на віддаленому Git-сервері. Залишіть його порожнім, щоб не виникало плутанини. Назвіть його якось просто й зрозуміло, наприклад, «*Counter*» або «*Pharo-Counter*». Це те місце, куди ми плануємо відправити наш проєкт з Iceberg.

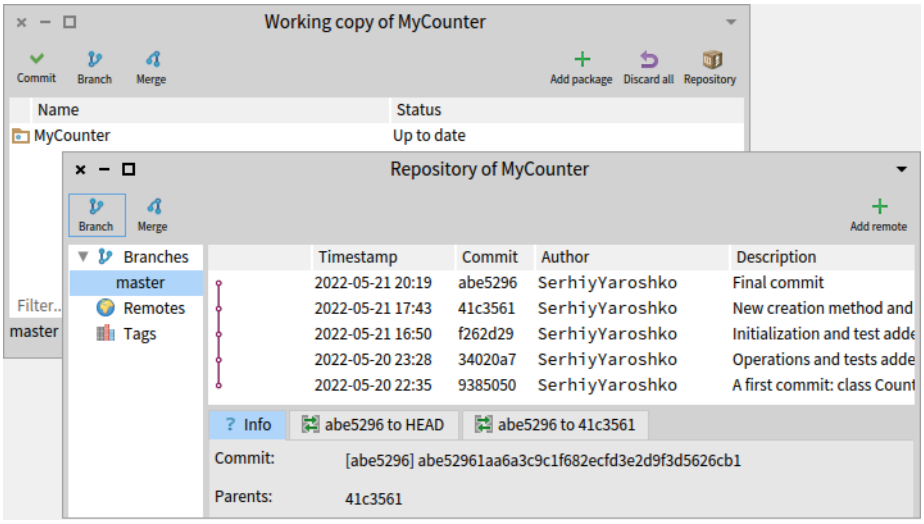

Рис. 5.16. Оглядач репозиторіїв вашого проєкту

Додайте віддалений репозиторій

У головному вікні Iceberg двічі клацніть на рядку з вашим проєктом *MyCounter*, щоб відкрити оглядач робочої копії репозиторію. Тоді клацніть на кнопці ***Repository*** (вона оздоблена піктограмою з зображенням ящика). Вона відкриє оглядач репозиторіїв проєкту *MyCounter*, як показано на рис. 5.16.

Тоді вам просто потрібно додати віддалене сховище для проєкту, що так само просто, як клацнути кнопку ***Add remote***. Вам запропонують вказати ім'я віддаленого сховища і його URL-адресу. Ім'я можна вказати довільне, воно є лише міткою, яку Git використовує локально для ідентифікації. Адреса ж має бути точною і вести до створеного перед цим віддаленого репозиторію. Ви можете використовувати доступ HTTPS (URL-адреса, яка починається з https://github.com для GitHub) або доступ SSH (URL-адреса, яка починається з git@github.com). SSH потребуватиме налаштування агента SSH на вашому комп'ютері з правильними обліковими даними (будь ласка, зверніться до свого постачальника Git, щоб дізнатися, як цього досягти).

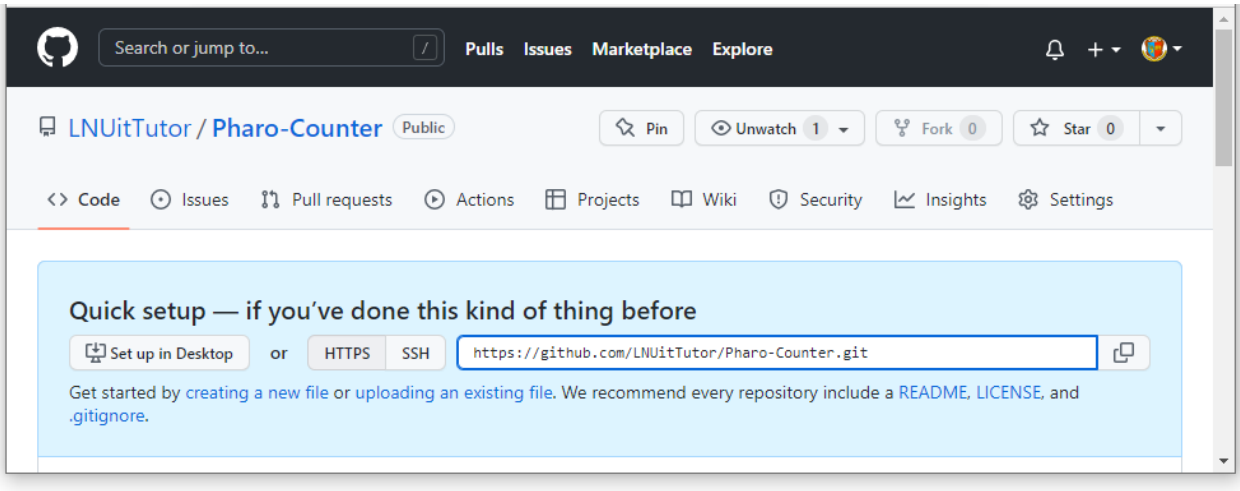

Рис. 5.17. HTTPS адреса нашого репозиторію на GitHub

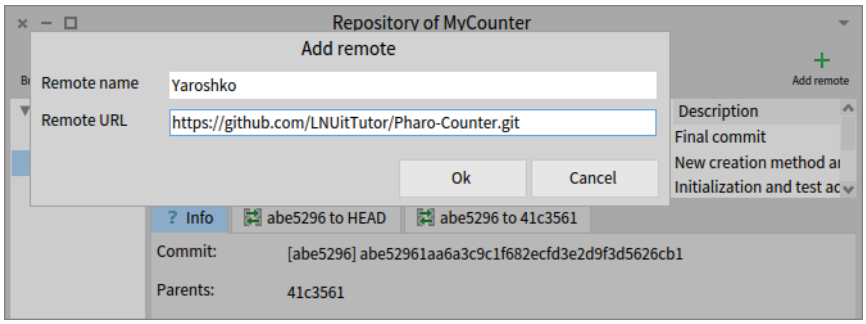

Рис. 5.18. Використання HTTPS адреси з GitHub

Ми використаємо HTTPS доступ. Адресу скопіюємо зі сторінки репозиторію на GitHub (рис. 5.17) і вкажемо її в модальному діалозі *Add remote* (рис. 5.18).

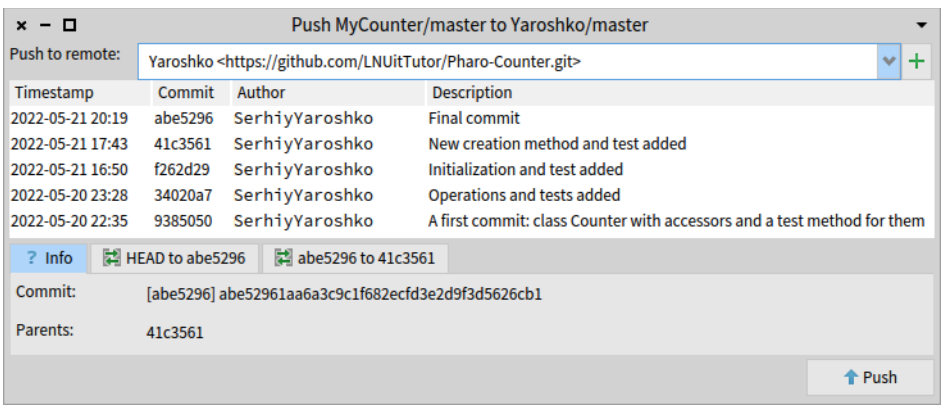

Рис. 5.19. Перелік збережень, які буде перенесено на віддалений сервер

Перенесіть проєкт у віддалене сховище

Як тільки ви вкажете справжню адресу віддаленого сервера, Iceberg відобразить на кнопці ***Push*** помітний червоний індикатор. Він сигналізує, що у вашому локальному сховищі є зміни, ще не перенесені до віддаленого сховища. Все, що потрібно зробити, клацнути на кнопці ***Push***. Iceberg покаже вам список збережених змін (commits), які буде перенесено на сервер (рис. 5.19). Вам треба натиснути ще одну кнопку ***Push***.

Коли ви зробите це вперше, HTTPS зажадає від вас ввести ім'я користувача на GitHub та токен доступу. Pharo збереже їх для вас на майбутнє.

Тепер ви *справді* зберегли свій код і зможете оновити його з іншої машини або місця. Це вміння дасть змогу вам працювати віддалено, а також ділитися програмами та співпрацювати з іншими розробниками.

Від перекладача. Віднедавна GitHub перестав надавати доступ для сторонніх програм через ім'я користувача та пароль, натомість потрібно використовувати ім'я і персональний токен доступу. Перш ніж переносити проєкт на GitHub, згенеруйте собі такий токен: увійдіть у свій обліковий запис, відкрийте сторінку налаштувань (Settings), у самому низу лівої панелі сторінки відшукайте посилання на підрозділ налаштувань розробника (Developer settings) і відкрийте його. Тут ви знайдете підрозділ керування токенами доступу (Personal access tokens), де обліковано всі наявні токени та можна згенерувати нові. Зверніть увагу на те, що згенерований токен не можна побачити двічі. Як тільки ви закриєте сторінку, текстове зображення токена зникне, і ви не зможете відкрити його знову. Тому одразу збережіть його в окремому файлі або перенесіть одразу до Pharo.

Увесь процес генерування докладно описаний у документації, доступній за адресою https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token

---

5.14. Підсумки розділу

У цьому практикумі ви дізналися, як визначати пакети, класи, методи та тести. Він був першим, тому ми використали традиційний для більшості мов програмування процес розробки програм. Однак у Pharo розробники використовують інший робочий процес, розумний і гнучкий: розробку, керовану тестуванням (TDD). Ми пропонуємо вам повторити всю цю вправу в стилі TDD: спочатку визначте тест, запустіть його, отримайте повідомлення про помилку та визначте метод у налагоджувачі, а потім повторіть такі ж дії для наступного методу. Перегляньте друге відео «Counter» у Pharo MOOC, доступне на http://mooc.pharo.org, щоб краще зрозуміти робочий процес.