Розділ 12
SUnit – модульне тестування у Pharo

SUnit – це невеликий але потужний інструмент, що підтримує створення та виконання тестів. Як можна здогадатися з його назви, SUnit розроблено для *модульного тестування* (для unit-тестів), але, насправді, його можна з успіхом використовувати і для інтеграційного, і для функціонального тестування. Початкову версію SUnit розробив Кент Бек (Kent Beck), її поступово доповнили Йозеф Пелріне (Joseph Pelrine) та багато інших розробників. Платформа SUnit материнська для інших xUnit платформ.

Розділ спеціально написано коротким, щоб переконати читача, що тестування просте. Детальніший опис SUnit і різних підходів до тестування можна прочитати в книзі «*Testing in Pharo*», доступній за адресою http://books.pharo.org.

У цьому розділі почнемо з обговорення того, чому тестуємо код, і що робить тести надійними. Далі продемонструємо низку невеликих прикладів як користуватися SUnit. Зазначимо, що в розділі описано версію SUnit3.3, яку тепер використовують у Pharo.

12.1. Вступ

Інтерес до тестування і розробки програм через тестування (Test Driven Development, TDD) не обмежується середовищем Pharo. Автоматизоване тестування стало невід'ємною ознакою *технології гнучкої розробки програмного забезпечення* (Agile software development), і кожен програміст, який турбується про підвищення якості свого програмного забезпечення, мав би взяти його на озброєння. Справді, розробники багатьма мовами програмування оцінили важливість модульного тестування, і середовища *xUnit* тепер існують для всіх мов програмування.

Ні тестування, ні побудова наборів тестів не є чимось новим – всі знають, що тестування є хорошим способом виявлення помилок. Технологія екстремального програмування (eXtreme Programming) піднесла тестування до рангу однієї з головних практик, наголошуючи на важливості *автоматизованих тестів*, і так перетворила тестування з рутини, яку так не люблять програмісти, на веселе і продуктивне заняття.

SUnit важливий, бо дає змогу писати *виконувані* тести, що містять визначення правильного результату, тому придатні для самоперевірки. Він допомагає також об'єднувати тести в групи, визначати контекст їхнього виконання й автоматично запускати групи тестів. За допомогою SUnit ви зможете за невеликий час створити вичерпний набір тестів для будь-якої мети, тому ми закликаємо вас замість покрокового тестування фрагментів коду в робочому вікні використовувати SUnit та всі його переваги зберігання і автоматичного виконання тестів.

12.2. Чому тестування важливе

Багато програмістів, на превеликий жаль, вважають, що написання тестів – це даремна трата часу. Адже це *інші* програмісти помиляються, а *вони* ніколи не пишуть з помилками. Багато хто з нас говорив: «*Я б писав тести, якби мав більше часу*». Якщо ви ніколи не робите помилок, і ваша програма ніколи не змінюватиметься у майбутньому, тоді тестування справді є даремним витрачанням вашого часу. Проте, це, найімовірніше, означає, що ваша програма або тривіальна, або її не використовуєте ні ви, ні будь-хто інший. Про тести потрібно думати як про інвестицію в майбутнє: корисний сьогодні набір хороших тестів стане просто *незамінним* у майбутньому, коли зазнає змін ваша програма чи середовище, в якому вона виконується.

Тести виконують кілька завдань одночасно. Найперше вони слугують документацією тієї функціональності програми, яку покривають. Ба більше, така документація активна: проходження тестів без помилок засвідчує її актуальність. По друге, тести допомагають розробникам переконатися, що щойно зроблені зміни певної частини коду не порушують функціонування решти системи, або локалізувати помилки в протилежному випадку. І нарешті, якщо ви пишете тести одночасно з програмою, або навіть перед нею, то починаєте думати про функціональність, яку проєктуєте, і ймовірно про те, *як це виглядатиме для користувача*, ніж як це реалізувати.

Якщо ви спершу пишете тести, а потім – код, то змушені визначати контекст, у якому виконуватиметься проєктована функціональність, спосіб взаємодії з кодом користувача й очікувані результати. Спробуйте так програмувати, і ви побачите, що ваш код стане кращим.

Ми не можемо всесторонньо протестувати жодного справжнього застосунку. Покриття тестами цілої програми просто неможливе, тому не мало б бути метою тестування. Навіть після застосування досконалого набору тестів окремі помилки можуть закрастися до аплікації і «залягти там на дно», очікуючи слушної нагоди, щоб зруйнувати всю систему. Якщо так справді трапиться, то скористайтеся моментом на свою користь! Як тільки знайдено не покриту тестами помилку, напишіть тест, що мав би її виявляти, запустіть його і переконайтеся, що він завершується невдачею. Тепер ви можете перейти до виправлення помилки. Успішне проходження тесту засвідчить, що ви зробили бажане.

12.3. Що робить тест хорошим?

Уміння писати хороші тести найлегше здобути на практиці. Розглянемо умови, за яких тести матимуть найбільше користі.

- *Тести мають бути повторювані*. У вас має бути змога запускати їх так часто, як би ви хотіли. Щоразу тести повинні повертати ті самі результати.
- *Тести повинні виконуватися без втручання людини*. У вас має бути змога запускати їх без особистого нагляду.
- *Тести мають бути інформативні*. Тест повинен «розповідати історію»: мав би діяти як сценарій, прочитавши який, і ви, і будь-хто інший зрозумів би функціональність цієї частини коду.
- *Тест повинен перевіряти один аспект коду*. Коли тест не проходить, він повинен показати, що неправильно працює щось одне. Справді, якщо тест охоплює кілька

аспектів, по-перше, він буде падати частіше, а по-друге, це змусить розробника під час виправлення брати до уваги більший набір параметрів.

- *Тести мають бути стабільними* і змінюватися рідше ніж код, який вони перевіряють, адже ніхто не хотів би переписувати весь набір тестів після кожної зміни в програмі. Єдиний спосіб досягнення такої властивості – писати тести, що взаємодіють з тестованим класом через його відкритий інтерфейс.

Як наслідок кількість тестів мала б бути приблизно пропорційна кількості тестованих функцій, методів. Зміна однієї властивості системи не має «завалювати» всі тести, а лише обмежену кількість з них. Це важливо, бо невиконання сотні тестів набагато серйозніше попередження, ніж невиконання десяти. Проте не завжди вдається досягти такого ідеалу, зокрема, якщо зміни зачіпають ініціалізацію об'єкта або налаштування тестів, то ймовірно всі тести зазнають невдачі.

12.4. SUnit крок за кроком

Написання тестів само собою не складне. Отож напишемо свій перший тест і продемонструємо переваги SUnit. Використаємо приклад тестування класу *Set*.

Виконаємо такі кроки:

- визначимо клас для групування тестів і використання можливостей SUnit;
- визначимо методи-тести;
- використаємо стандартні методи для перевірки очікуваних результатів;
- виконаємо тести.

Пишіть код і виконуйте тести по ходу читання.

12.5. Крок 1. Створіть клас тестів

Найперше потрібно створити новий підклас класу *TestCase*. Назвемо його *MyExampleSetTest* і додамо дві змінні екземпляра класу. Тоді новий клас матиме такий вигляд:

```
TestCase subclass: #MyExampleSetTest
   instanceVariableNames: 'full empty'
   classVariableNames: ''
   category: 'MySetTest'
```

Використаємо клас *MyExampleSetTest*, щоб об'єднати всі тести, які стосуються класу *Set*. Він визначає контекст їхнього виконання. Тут контекст описано двома змінними екземпляра *full* і *empty*, що міститимуть відповідно повну і порожню множини.

Ім'я класу тестів може бути довільним, але за домовленістю воно закінчується на *Test*. Якщо ви визначите клас *Pattern* і назвете відповідний клас тестів *PatternTest*, то в Оглядачі класів вони розташуються поруч в алфавітному порядку (у припущенні, що обидва класи належать до того самого пакета). Усі класи тестів *обов'язково* мають бути підкласами *TestCase*.

12.6. Крок 2. Налаштуйте контекст виконання тестів

Метод *TestCase >> setUp* визначає контекст, у якому працюватимуть тести. Він трохи схожий на метод ініціалізації. Метод *setUp* виконується автоматично перед викликом кожного методу, оголошеного в класі тестів.

Визначимо метод екземпляра *setUp*, як описано нижче, щоб змінна *empty* містила порожню множину, а змінна *full* – множину з двома елементами.

```
MyExampleSetTest >> setUp
  empty := Set new.
  full := Set with: 5 with: 6
```

На жаргоні тестування контекст тесту називають *фікстурою*.

12.7. Крок 3. Напишіть кілька методів тестування

Створимо кілька тестів, визначивши методи в класі *MyExampleSetTest*. Кожен метод представляє один тест. Ім'я методу має розпочинатись словом «*test*», щоб SUnit міг збирати їх в набори тестів. Тестовий метод не приймає аргументів.

Визначимо такі тестові методи. Перший називається *testIncludes* і перевіряє метод *includes:* класу *Set*. Тест говорить, що надсилання повідомлення «*includes: 5*» множині, яка містить *5*, мало б повернути *true*. Зрозуміло, що він покладається на той факт, що метод *setUp* уже виконано.

```
MyExampleSetTest >> testIncludes
  self assert: (full includes: 5).
  self assert: (full includes: 6)
```

Другий тест називається *testOccurrences*. Він стверджує, що кількість входжень елемента *5* в множину *full* дорівнює одиниці навіть після того, як ми додамо ще один елемент *5* до цієї множини.

```
MyExampleSetTest >> testOccurrences
  self assert: (empty occurrencesOf: 5) equals: 0.
  self assert: (full occurrencesOf: 5) equals: 1.
  full add: 5.
  self assert: (full occurrencesOf: 5) equals: 1
```

І нарешті ми переконаємося, що множина *full* більше не містить елемент *5* після того, як ми його вилучили.

```
MyExampleSetTest >> testRemove
  full remove: 5.
  self assert: (full includes: 6).
  self deny: (full includes: 5)
```

Використовуйте метод *TestCase >> deny:*, для підтвердження того, що метод повертає хибу. Вираз «*aTest deny: anExpression*» означає те саме, що й «*aTest assert: anExpression not*», але виглядає набагато зрозуміліше.

У браузері класів ліворуч біля імені класу тестів і біля імені кожного тестового методу можна бачити піктограму – сірий кружечок. Сірий колір свідчить про невизначений

статус тестів: поки що результат перевірки невідомий. Після запуску тестів колір піктограми зміниться.

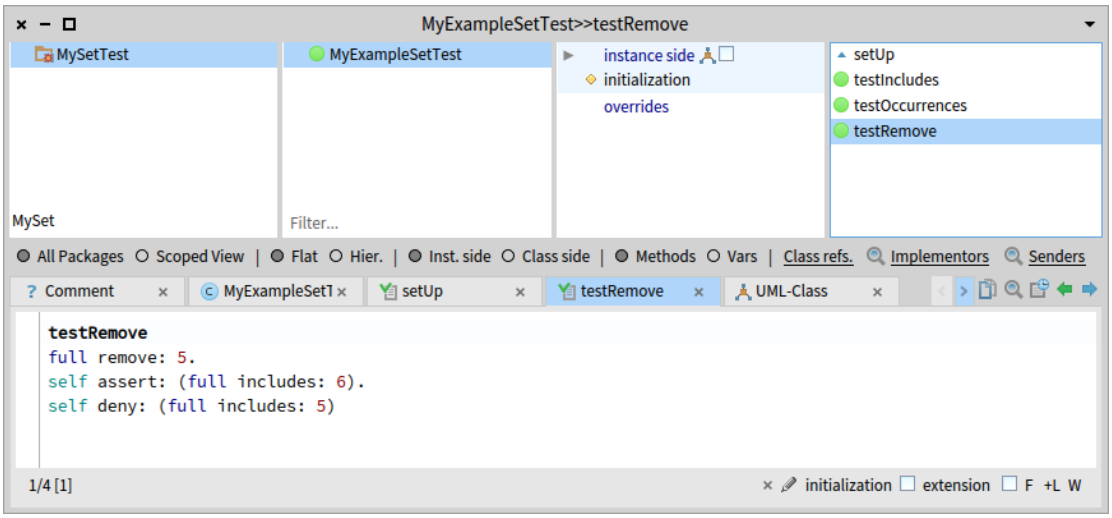

Рис. 12.1. Запуск модульних тестів з Оглядача класів

12.8. Крок 4. Запустіть тести

Найлегше запустити тести на виконання безпосередньо з Оглядача класів. Клацніть контекстно на імені пакета, класу чи окремого методу і виберіть з меню команду «*Run tests (Cmd+T)*», або просто клацніть на піктограмі класу чи методу. Колір піктограми методу зміниться на зелений або жовтий, залежно від того, завершився тест успіхом чи ні. Якщо всі тести успішні, то піктограма класу також стане зеленою, а в протилежному випадку – жовтою (див. рис. 12.1).

Запустити тести можна також за допомогою Менеджера тестів SUnit, який відкривають командою *World > Browse > Test Runner*. Тут ви можете вибрати для запуску кілька наборів тестів і отримати детальний звіт про їхнє виконання.

Відкрийте Менеджер тестів, позначте пакет *MySetTest* і натисніть кнопку ***Run Selected***.

Можна також запустити окремий тест програмно, виконавши такий фрагмент коду «*MyExampleSetTest run: #testRemove*», наприклад, за допомогою команди «*Print it*» у Робочому вікні. Відповіддю буде звичайний звіт про проходження тесту.

Дехто вставляє в тестові методи виконувані коментарі, як показано нижче, щоб можна було запустити тест командою «*Do it*» в Оглядачі.

```
MyExampleSetTest >> testRemove
  "self run: #testRemove"
  full remove: 5.
  self assert: (full includes: 6).
  self deny: (full includes: 5)
```

Зробіть навмисно помилку в методі *MyExampleSetTest >> testRemove*, наприклад, замініть 6 на 7 і запустіть тест ще раз – він мав би завершитись невдачею. Імена тестів, що не пройшли, з'являються в правій частині вікна Менеджера тестів. Щоб перейти до налагодження котрогось з них і побачити, що спричинило помилку, клацніть на імені тесту. Ще один спосіб розпочати налагодження – виконати один з таких фрагментів коду.

```
(MyExampleSetTest selector: #testRemove) debug

MyExampleSetTest debug: #testRemove
```

12.9. Крок 5. Потрактуйте результати тестування

Метод *assert:*, визначений в класі *TestAsserter*. Це надклас *TestCase*, тому *TestCase* та його підкласи відповідальні за всі способи перевірки результатів тестування. Метод *assert:* приймає один аргумент логічного типу. Зазвичай це результат обчислення виразу, який перевіряють. Якщо він дорівнює *true*, то тест завершується успіхом, а в протилежному випадку – невдачею. Вираз з *assert:* називають твердженням тесту.

Насправді є три можливих результати виконання тесту: *завершився успіхом або пройшов*, *завершився невдачею або упав*, *спричинив помилку*.

- **Пройшов**. Ми сподіваємося, що всі твердження в тесті будуть істинні, і він пройде. Якщо всі тести набору проходять, верхня права панель вікна Менеджера тестів забарвлюється зеленим кольором. (Якщо ви запускаєте тести з Оглядача класів, то про їхнє завершення сигналізує поява інформаційного вікна – зеленого, якщо тести пройшли).
- **Упав**. Звичайна річ, коли одне з тверджень тесту виявляється хибним, і тест не проходить. Тести, що не пройшли, забарвлюють верхню праву панель вікна Менеджера у жовтий колір, а їхні імена відображаються в середній правій панелі.
- **Помилка**. В інших випадках трапляються помилки на етапі виконання коду тестового методу, наприклад, «*message not understood*» (об'єкт не знайшов методу опрацювання отриманого повідомлення), або «*index out of bounds*» (індекс вийшов за допустимі межі), або ще якісь. Якщо трапиться помилка, то перевірки в тестовому методі не будуть виконані до кінця, і ми не знатимемо, проходить тест, чи ні, але буде зрозуміло, що щось точно є неправильно. Про тести з помилками в коді сигналізує червоний колір, а їхні імена потрапляють у праву нижню панель вікна Менеджера.

Поекспериментуйте зі створеними раніше тестами так, щоб спричинити виникнення і помилок, і відмов.

12.10. Використання assert:equals:

У випадку відмови тесту повідомлення *assert:equals:* надає кращу діагностику ніж звичайне *assert:*. Наприклад, наведені нижче тести еквівалентні, проте другий з них повідомлятиме очікуване в тесті значення. Це полегшує розуміння причин відмови. У прикладах припускається, що *aDateAndTime* змінна екземпляра тестованого класу.

```
testAsDate
   self assert: aDateAndTime asDate =
      ('February 29, 2004' asDate translateTo: 2 hours).
```

```
testAsDate
   self assert: aDateAndTime asDate
   equals: ('February 29, 2004' asDate translateTo: 2 hours).
```

12.11. Як пропустити тест

Якщо під час розробки програми у вас виникне потреба призупинити перевірку окремого тесту, то замість його вилучення з набору чи перейменування краще пропустити його. Для цього достатньо надіслати повідомлення *skip* екземплярові класу тестів. У прикладі нижче його використано для визначення умовного тесту.

```
OCCompiledMethodIntegrityTest >> testPragmas
   | newCompiledMethod originalCompiledMethod |
   (Smalltalk globals hasClassNamed: #Compiler) ifFalse: [ ^ self skip ].
    ...
```

Це зручно, автоматично запускається велика кількість тестів, і важливо отримати звіт про їхнє успішне виконання.

12.12. Перевірка виникнення винятків

SUnit надає два додаткові важливі методи *TestAsserter >> should:raise:* та *TestAsserter >> shouldnt:raise:* для тестування виникнення винятків.

Наприклад, ви можете використати «*self should: aBlock raise: anException*», щоб переконатися, чи виконання блока *aBlock* запускає виняток *anException*. Використання методу *should:raise:* демонструє тестовий метод нижче.

```
MyExampleSetTest >> testIllegal
   self should: [ empty at: 5 ] raise: Error.
   self should: [ empty at: 5 put: #zork ] raise: Error
```

Спробуйте запустити цей тест. Пам'ятайте, що першим аргументом повідомлень *should:raise:* та *shouldnt:raise:* мав би бути блок, який містить вираз до виконання.

12.13. Програмний запуск тестів

Зазвичай тести запускають за допомогою Менеджера тестів або Оглядача класів. Програвач можна відкрити командою головного меню або програмно, виконавши «*TestRunner open*».

Запуск одного тесту

Як вже було сказано, окремий тест можна запустити програмно.

```
MyExampleSetTest run: #testRemove
>>> 1 ran, 1 passed, 0 skipped, 0 expected failures, 0 failures,
    0 errors, 0 passed unexpected
```

Запуск усіх тестів тестового класу

Кожен підклас класу *TestCase* у відповідь на повідомлення *suite* повертає набір тестів, який містить усі методи цього підкласу, чиї імена починаються на «*test*». Щоб запустити на виконання цей набір тестів, йому надсилають повідомлення *run*. Наприклад, як показано нижче.

```
MyExampleSetTest suite run
```

```
>>> 4 ran, 4 passed, 0 skipped, 0 expected failures, 0 failures,
    0 errors, 0 passed unexpected
```

12.14. Підсумок розділу

Пояснено, чому тести є важливою інвестицією в майбутнє програмного коду. Описано покроково як визначити кілька тестів для класу *Set*.

- Щоб максимізувати свій потенціал, модульні тести мають бути швидкими, повторюваними, незалежними від будь-якої прямої взаємодії з людиною та охоплювати одну функціональну одиницю.
- Тести для класу, що називається *MyClass*, мають належати до класу *MyClassTest*, оголошеного підкласом *TestCase*.
- Контекст виконання тестів налаштовують методом *setUp*.
- Назва кожного тестового методу має розпочинатися словом *test*.
- Для запису тверджень тесту використовують методи класу *TestCase – assert:*, *deny:* та інші.
- Запускайте тести!

Кілька методологій розробки програмного забезпечення, такі як екстремальне програмування та розробка, керована тестуванням (TDD), рекомендують писати тести перед написанням коду. Може видатися, що це суперечить глибинним інстинктам розробників програмного забезпечення. Все, що можна сказати: спробуйте! З'ясувалося, що написання тестів перед програмою допомагає дізнатися, що саме потрібно кодувати, допомагає зрозуміти, коли робота закінчена, і допомагає концептуалізувати функціональність класу та розробити його інтерфейс. Крім того, розробка на основі тестування заохочує просуватися швидко, тому що не страшно забути щось важливе.