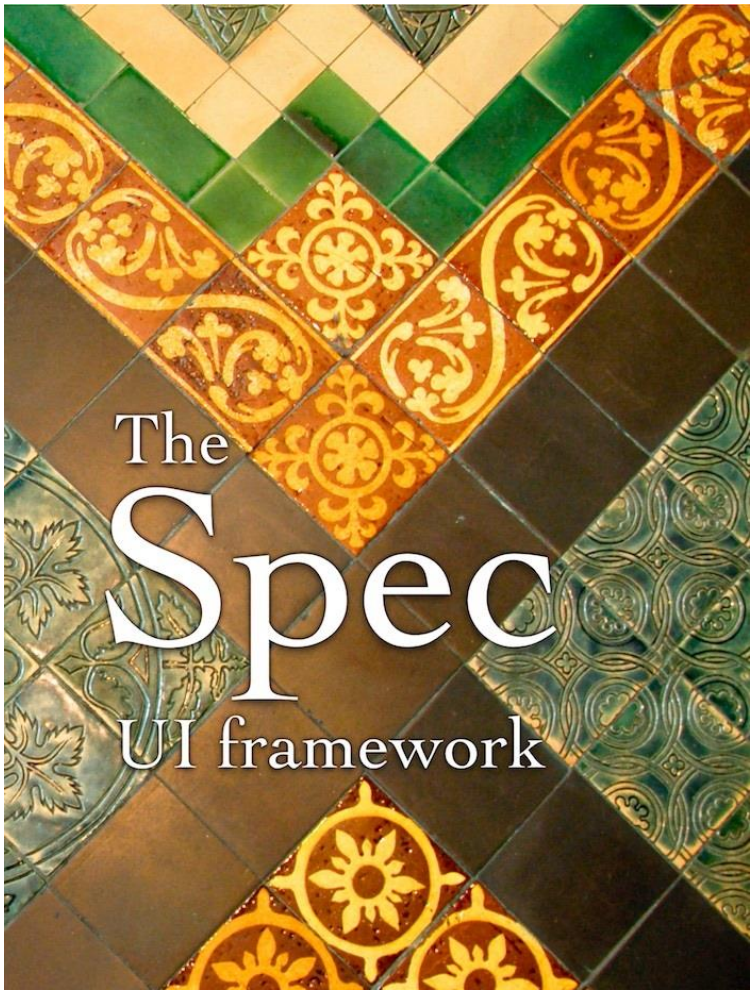

Побудова застосунків
засобами
Spec 2.0
К. Де Хондт
С. Дюкас
разом з
Е. Лоренцано
С. Джорданом-Монтаньо
Переклад з доповненнями
С. Ярошко

Побудова застосунків засобами Spec 2.0

Компонентно орієнтована розробка GUI



Application Building with Spec 2.0

K. De Hondt, S. Ducasse with S. Jordan-Montaño and
E. Lorenzano

October 26, 2024

BoD – Books on Demand

Побудова застосунків засобами Spec 2.0

К. Де Хондт, С. Дюкас разом з С. Джордан-Монтаньо та Е. Лоренцано

Переклад українською з доповненнями

Сергій Ярошко

ЛНУ ім. Івана Франка
2025

УДК 004.432.2
Х 77

Перекладено за оригіналом:

K. De Hondt, S. Ducasse with S. Jordan Montaño and E. Lorenzano
Application Building with Spec 2.0

Переклад:

Ярошко С. А. – канд. фіз.-мат. наук, доцент
(Львівський національний університет імені Івана Франка)

Рецензенти:

Іванчук Я. В. – д-р техн. наук, професор
(Вінницький національний технічний університет);

Міца О. В. – д-р техн. наук, професор
(Ужгородський національний університет);

Яремко Г. В. – канд. пед. наук, доцент
(Національний університет «Львівська політехніка»)

Рекомендовано до друку
Вченою радою Львівського національного університету імені Івана Франка.
Протокол № 82/5 від 01.05.2025 року

Коен Де Хондт

Х 77 Побудова застосунків засобами Spec 2.0/ К. Де Хондт, С. Дюкас [та ін.] ; пер. з англ. з допов. С. Ярошка. – Електрон. вид. – Львів : ЛНУ ім. Івана Франка, 2025. – 240 с.

ISBN 978-617-10-1094-9 (електрон. вид.)

Програмний каркас Spec призначено для побудови віконного інтерфейсу користувача застосунків, розроблених у середовищі програмування Pharo, сучасному втіленні класичної системи Smalltalk. Програмування у Spec можна назвати компонентно-орієнтованим, оскільки кожен візуальний компонент може стати і самостійним інтерфейсом, і частиною більшого компонента. Послідовно викладено правила побудови застосунку засобами Spec, описано демонстратори (стандартні і збудовані користувачем), макети, стилі, команди, перенесення даних, вікна та діалоги. Наведено багато прикладів конкретних застосунків.

Для здобувачів вищої освіти галузі знань F Інформаційні технології і всіх, хто цікавиться сучасними технологіями програмування.

УДК 004.432.2

ISBN 978-617-10-1094-9 (електрон. вид.)

Зміст

Глосарій

Application	*Застосунок*	Комп'ютерна програма з віконним графічним інтерфейсом користувача
Debugger	*Налагоджувач*	Програма, інструмент для дослідження коду, що виконується. У Pharo можна використовувати для написання програм на льоту
File Browser	*Оглядач файлів*	Інструмент для доступу до файлової системи комп'ютера, має можливості завантаження, швидкого перегляду файлів
Finder	*Шукач*	Програма, інструмент для відшукання методів, класів, програмного коду, анотацій, прикладів. Уміє шукати за зразком перетворень
Iceberg	*Менеджер пакетів*	Система контролю версій для взаємодії з Git
Image	*Образ системи*	Колекція всіх об'єктів системи, завантажується у віртуальну машину
Inspector	*Інспектор об'єктів*	Програма, інструмент, який дає змогу заглянути всередину об'єкта, вивчити та змінити його стан
Layout	*Макет*	Клас, який керує розташування візуальних компонентів у вікні застосунку
Morph	*Морфа*	Видимий елемент інтерфейсу користувача
Playground	*Пісочниця*	Вікно для безпечного виконання фрагментів коду, інтегроване з Інспектором об'єктів
Presenter	*Демонстратор*	Клас, що відображає візуальний компонент, частину графічного інтерфейсу користувача, реалізує логіку його поведінки
Spotter	*Навідник*	Вікно діалогу для швидкого відшукання будь-чого в системі
System Browser	*Оглядач класів, Системний оглядач*	Головний інструмент системи Pharo, який надає доступ до всіх пакетів класів, їхніх методів, дає змогу оголошувати нові класи та редагувати наявні
Transcript	*Консоль системи*	Текстове вікно, яке можна використовувати для виведення тестових повідомлень від об'єктів, публікує повідомлення про помилки під час інсталяції пакетів
World Menu	*Головне меню*	Pharo – окремий світ на комп'ютері, тому його головне меню називається «Світ»

Розділ 1

Вступ

Spec – це програмний каркас у Pharo для опису графічних інтерфейсів користувача. Він дає змогу створювати широкий спектр інтерфейсів користувача: від невеликих вікон із кількома кнопками до складних інструментів, як-от налагоджувач. Насправді багато інструментів Pharo написані в Spec, наприклад, Iceberg (менеджер Git), Change Sorter (хранитель змін коду), Critics Browser (оглядач якості коду) і налагоджувач Pharo. Важливим архітектурним рішенням у Spec є те, що він може використовувати різні графічні бібліотеки для зображення візуальних елементів (на момент написання цієї книги доступні GTK і Morphic).

1.1. Повторне використання інтерфейсів

Головним принципом Spec є повторне використання логіки поведінки графічного інтерфейсу користувача та компонування його вигляду[1]. Нові графічні інтерфейси створюють за допомогою повторного використання наявних: їх компонують і за потреби додатково налаштовують. Цей принцип діє, починаючи з найпростіших елементів інтерфейсу: навіть кнопка чи напис самі по собі є завершеним інтерфейсом користувача, який можна налаштувати і відкрити в окремому вікні, або використати для побудови більших вікон. Графічні елементи можна комбінувати, щоб сформувати складніші інтерфейси користувача, які знову можна повторно використовувати як частину більшого інтерфейсу і так далі. Такий підхід схожий на те, як поєднують різні кахельні плитки, зображені на обкладинці першої книги про Spec 1.0 (її зменшена копія є на форзаці цієї книги). Маленькі плитки підбирають за кольором або візерунком і з'єднують, щоб утворити більші прямокутні форми, які є частиною ще більшого дизайну підлоги.

Для того, щоб таке повторне використання стало можливим, розробники Spec використали шаблон Модель-Вигляд-Демонстратор (Model-View-Presenter, MVP), притаманний VisualWorks і Dolphin Smalltalk. Для функціонування Spec потрібний клас Демонстратора. Демонстратор описує зв'язки між доменом і візуальними компонентами, а також логіку взаємодії візуальних компонентів, що входять до складу застосунку.

У Spec 1.0 цю роль виконував клас *ComposableModel*, а тепер у Spec 2.0 – клас *SpPresenter*. Демонстратор керує логікою функціонування інтерфейсу користувача та зв'язками між візуальними компонентами й об'єктами домену. Насправді, під час написання коду Spec розробники не взаємодіють з компонентами інтерфейсу користувача безпосередньо. Натомість вони програмують Демонстратор, який містить логіку інтерфейсу користувача (взаємодії, макет тощо) і спілкується з об'єктами домену. Відповідні графічні компоненти Демонстратор створює автоматично під час відкривання інтерфейсу

[1] Відомо, що Microsoft Visual Studio надає засоби для побудови складених візуальних компонентів (компонентів користувача) на базі Windows Forms або WPF. Для цього створюють проєкт спеціального вигляду, компілюють компонент у окрему бібліотеку, яку згодом можна використовувати для створення віконних застосунків. Програмування зі Spec не потребує таких додаткових зусиль: кожен побудований візуальний елемент діє як компонент, придатний до використання у будь-якому застосунку. З огляду на це програмування зі Spec можна назвати «*компонентно-орієнтованим*» (прим. – *Ярошко С.*).

користувача. Тому така особливість неочевидна для розробників, і складається враження, ніби компоненти програмуються безпосередньо.

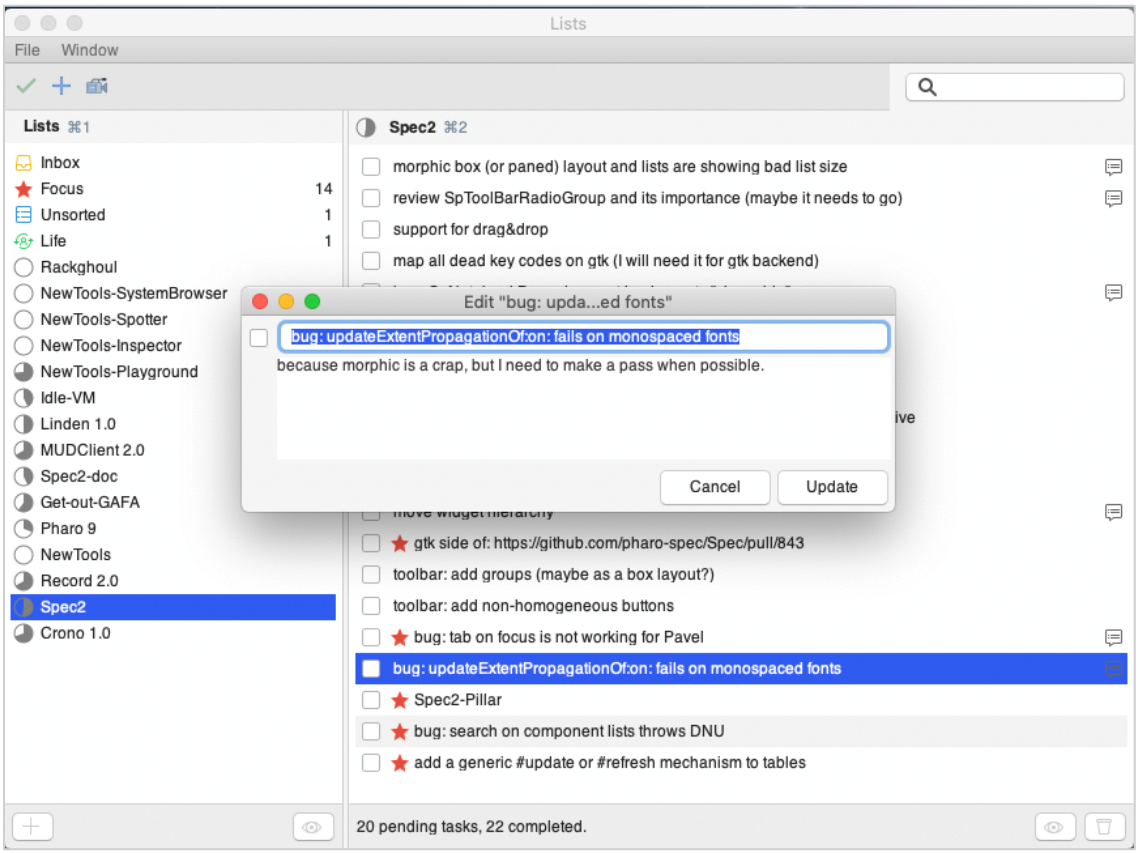


Рис. 1.1. Spec підтримує використання різних графічних бібліотек: Morphic і GTK3.0. Тут зображено GTK

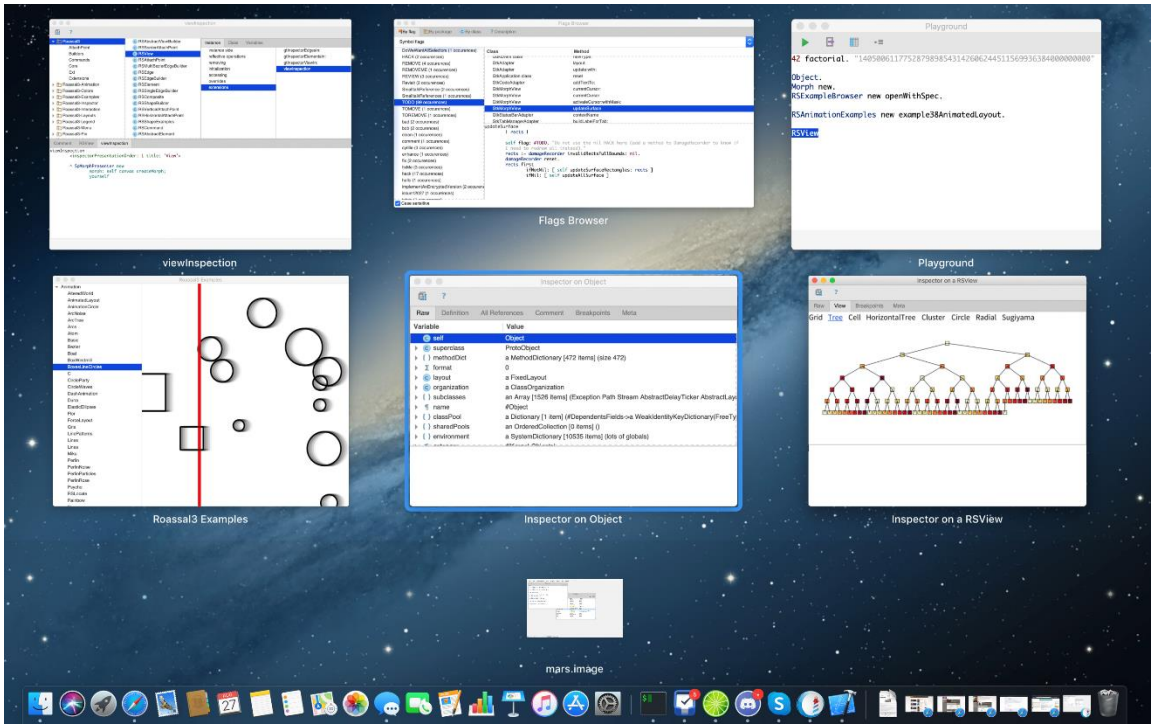


Рис. 1.2. Приклад інтеграції Spec і Roassal

Spec – це стандартний програмний каркас графічного інтерфейсу користувача у Pharo, який відрізняється від інших каркасів інтерфейсу Pharo (від Morphic, наприклад). Spec обмежений тим, що дає змогу створювати інтерфейс користувача для застосунків лише з типових візуальних компонентів: кнопок, списків тощо. Його не можна використовувати для довільних графічних побудов, але ви можете інтегрувати графічне полотно в компонент Spec.

Наприклад, ви можете вбудувати візуалізацію Roassal (див. рис. 1.2), або розширити сам Spec додатковими компонентами.

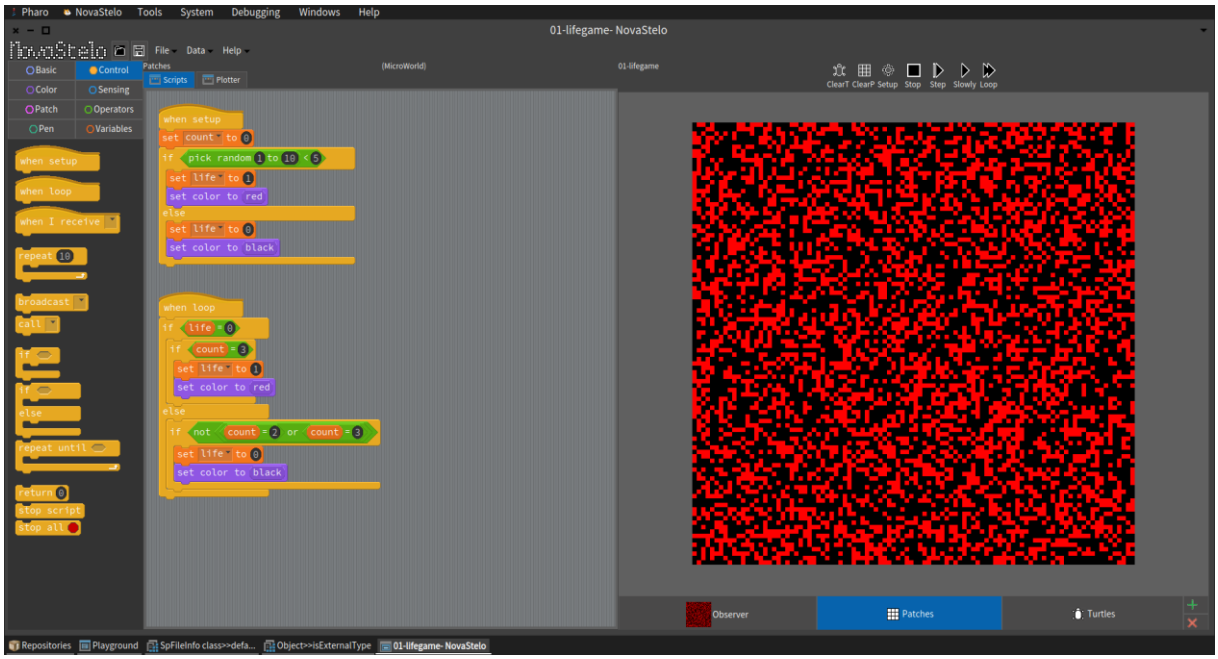


Рис. 1.3. Інтеграція графічних компонентів Morphic і Spec

Іншим прикладом інтеграції (див. рис. 1.3) є проєкт NovaStelo професора Е. Ітo. Він демонструє, що Spec можна використовувати для побудови загальної структури програми та вбудовувати необхідні елементи. На рис. 1.4 зображено копію екрана, зроблену членом спільноти на ім'я Walehead.

1.2. Spec 2.0

Починаючи зі Spec 2.0, для промальовування інтерфейсу ваших застосунків можна використовувати різні набори візуальних компонентів. На момент написання цієї книги Spec можна відобразити за допомогою графічних бібліотек Morphic або GTK. Spec 2.0 є великим наступним кроком розробки Spec 1.0. З'явилося багато вдосконалень: запроваджено новий спосіб опису макетів інтерфейсу користувача, переглянуто API, підтримано нові візуальні компоненти та додано інтеграцію з іншими проєктами, такими як *Commander*.

Мета Pharo – використовувати Spec для створення всіх власних графічних інтерфейсів. Такий підхід гарантує надійну підтримку Spec з часом і поліпшує стандартизацію інтерфейсів Pharo, а також їхню переносимість на нові графічні системи. Використання Spec 2.0 забезпечує незалежність від графічної реалізації та повторне використання логіки. Це означає, що написаний у Spec інтерфейс користувача відтворюватиметься іншими графічними бібліотеками, відмінними від GTK і Morphic. Як тільки з'являться нові бібліотеки, усі програми, написані на Spec, зможуть їх використовувати.

За основу цієї книги використано її попередню версію Spec, але текст майже повністю переписано, щоб досягти вищої якості. Він охоплює всі нові засоби. Ми сподіваємося, що це буде корисно для розробників, які пишуть інтерфейси користувача у Pharo.

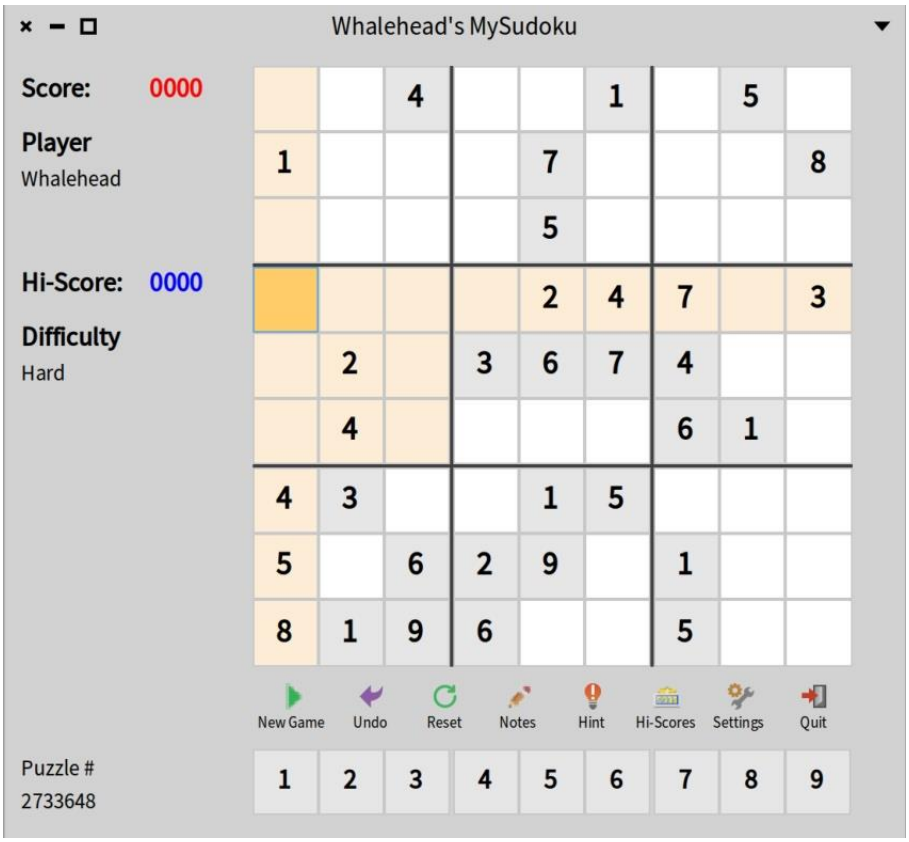


Рис. 1.4. Судоку

Зауваження. У книзі використано Pharo 12. Попередні версії Pharo оснащені іншими версіями Spec, тому окремі приклади коду з цієї книги у них можуть працювати неналежно. А втім, фундаментальні принципи розробки інтерфейсу користувача в Spec у всіх версіях ті самі.

Від перекладача. У книзі не обговорюють синтаксис Pharo, бібліотеку базових класів, способи використання інструментів середовища програмування тощо, тому читачам без досвіду розробки у Pharo варто спочатку прочитати книгу «Pharo 9 на прикладах», доступну за адресою https://books.pharo.org/pharo-by-example9/.

1.3. Програмний код прикладів

Програмний код усіх прикладів цієї книги збережено в репозиторії https://github.com/SquareBracketAssociates/CodeOfSpec20Book.

Ви можете інсталювати весь код одразу у свій образ Pharo, виконавши такий фрагмент:

```
Metacello new
   baseline: 'CodeOfSpec20Book';
   repository: 'github://SquareBracketAssociates/CodeOfSpec20Book/src';
   load
```

1.4. Подяки

Незважаючи на те, що через брак робочої сили не була проведена кампанія збору коштів, автори хотіли б висловити свою гарячу подяку таким людям за їхню фінансову підтримку: Масаші Фуджіта (Masashi Fujita), Рок-Александр Номіни (Roch-Alexandre Nominé), Ейічіро Ітo (Eiichiro Ito), Сумім (Sumim), Ілер Фернандес (Hilaire Fernandes), Домінік Дартуа (Dominique Dartois), Філіп Мужан (Philippe Mougin), Павел Криванек (Pavel Krivanek), Майкл Л. Девіс (Michael L. Davis), Іван Доусон (Ewan Dawson), Люк Фабресс (Luc Fabresse), Давід Байгер (David Bajger), Йорг Франк (Jörg Frank), Петтер Егесунд (Petter Egesund), П'єр Бюленс (Pierre Bulens), Томохіро Ода (Tomohiro Oda), Себастьян Хайдбрінк (Sebastian Heidbrink), Александр Бергель (Alexandre Bergel), Йонас Скучас (Jonas Skučas) і Марк Швенк (Mark Schwenk).

Хочемо подякувати I. Томас (I. Thomas) за її розділ про інспектора об'єктів та P. Де Вільмер (R. De Villemeur) за розділ про інтеграцію з *Athens*.

Нарешті, Стефан Дюкасс (Stéphane Ducasse) хоче подякувати Йохану Фабрі (Johan Fabry) за його співавторство першої книги про Spec 1.0. Без тієї першої книги не було б цієї. Він також хоче подякувати Коену (Koen), який із задоволенням долучився як співавтор і надзвичайно поліпшив книгу. Ще раз дякую, Коене. Це була весела пригода.

Хочемо подякувати ESUG і Pharo Association за спонсорську підтримку цієї книги. Це були справжні багаторічні зусилля.

Якщо ви підтримали нас, і вас немає в цьому списку, то зв'яжіться з нами або надішліть запит.

1.5. Від перекладача

Працездатність усіх прикладів з цієї книги перевірено у Pharo 12.0 на комп'ютері під управлінням ОС Windows 10. Окремі параграфи доповнено поясненнями та коментарями. Доповнено також третю частину книги. Тут з'явилися два розділи з прикладами використання Spec для побудови застосунків: у розділі 19 створено програму, що імітує роботу вуличного світлофора, а в розділі 20 – новий демонстратор, який об'єднує групу залежних перемикачів, суттєво спрощує їхнє налаштування і використання. Їхній програмний код можна завантажити зі сховищ LNUitTutor/TrafficLightsBySpec та LNUitTutor/RadioGroupProject, відповідно.

Переклад окремих, специфічних для Pharo термінів, зазначено в глосарії.

Перекладач книги висловлює подяку ESUG та Pharo Association за фінансову підтримку видання перекладу.

Частина I
Увесь Spec в одному прикладі

Розділ 2

Маленький приклад на 10 хвилин

Ми створимо невеликий, але завершений графічний інтерфейс користувача. Так ви навчитеся створювати базові інтерфейси.

Після завершення цього розділу ви можете перейти відразу до розділу 7. Він розповідає про повторне використання демонстраторів Spec, яке є ключем до потужності Spec. Після ознайомлення з цими двома розділами ви мали б отримати достатньо знань, щоби створювати у Spec інтерфейси користувача для різних потреб. Решту цієї книги можна використовувати як довідковий матеріал, але, все ж таки, ми рекомендуємо вам хоча б коротко ознайомитися з іншими розділами.

2.1. Вікно для опитування клієнтів

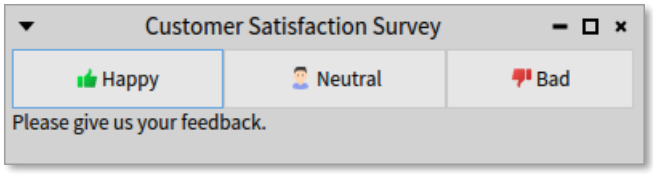


Рис. 2.1. Віконний інтерфейс опитування про задоволеність клієнтів

Ми створимо простий інтерфейс користувача для опитування про задоволеність клієнтів, який дає змогу залишити відгук про послугу, натиснувши одну з трьох кнопок. Отриману відповідь потрібно було б записати й опрацювати, але це виходить за межі прикладу. На рис. 2.1 зображено миттєвий знімок інтерфейсу користувача.

2.2. Створіть клас для інтерфейсу користувача

Усі інтерфейси користувача в Spec є підкласами *SpPresenter*, тому найперше потрібно наслідувати цей клас:

```
SpPresenter << #CustomerSatisfactionPresenter
   slots: { #buttonHappy . #buttonNeutral . #buttonBad . #result };
   package: 'CodeOfSpec20Book'
```

Змінні екземпляра класу міститимуть демонстратори, з яких складається інтерфейс користувача, так звані піддемонстратори, або вкладені демонстратори. У цьому випадку маємо три кнопки та напис, щоб відобразити результат опитування.

Вкладені демонстратори створюють і налаштовують у методах цього класу, наприклад, задають написи кнопок і реакції на натискання, а також логіку їхньої взаємодії. Базовий дизайн інтерфейсу, тобто те, як розташовані демонстратори, також визначає клас.

2.3. Створення та налаштування вкладених демонстраторів

Підкласи *SpPresenter* відповідальні за визначення методу *initializePresenters*, який створює та конфігурує демонстратори, що входять до складу інтерфейсу. Ми обговоримо

цей метод крок за кроком. Зауважте, що він може бути трохи довгим, тому розділимо його на частини, кожна з яких виконує певне завдання.

Створення демонстраторів

```
CustomerSatisfactionPresenter >> initializePresenters
   result := self newLabel.
   buttonHappy := self newButton.
   buttonNeutral := self newButton.
   buttonBad := self newButton
```

Клас *SpPresenter* визначає повідомлення для створення стандартних демонстраторів: *newButton, newCheckBox, newDropList* тощо. Усі вони визначені в протоколі *scripting - widgets* ознаки[2] *SpTPresenterBuilder*. Це методи-обгортки для зручного створення демонстраторів.

Нижче показано визначення методу *newButton*.

```
SpPresenter >> newButton
   ^ self instantiate: SpButtonPresenter
```

Зауважимо, що найменування методів може трохи збивати з пантелику, адже ми пишемо *newButton*, хоча насправді буде створено *демонстратор* кнопки, а не *візуальний компонент* кнопки, про який Spec подбає сам. Метод у Spec названо *newButton*, бо так легше записувати, ніж *newButtonPresenter*.

Не викликайте *new*, щоб створити демонстратор, який є частиною вашого інтерфейсу користувача. Альтернативним способом створення демонстраторів є надсилання повідомлення *instantiate*: з аргументом – класом демонстратора. Наприклад, *result := self instantiate: SpLabelPresenter*. Такий спосіб дає змогу створювати стандартні та нестандартні демонстратори.

Налаштування демонстраторів

Наступний крок – конфігурування кнопок нашого інтерфейсу користувача. Повідомлення *label:* задає напис на кнопці, а повідомлення *icon:* визначає піктограму, яка буде зображена біля напису.

```
CustomerSatisfactionPresenter >> initializePresenters
 ... продовження ...
   result label: 'Please give us your feedback.'.
   buttonHappy
      label: 'Happy';
      icon: (self iconNamed: #thumbsUp).
   buttonNeutral
      label: 'Neutral';
      icon: (self iconNamed: #user).
   buttonBad
      label: 'Bad';
      icon: (self iconNamed: #thumbsDown)
```

[2] *Ознака (traits)* у Pharo – сутність, яка містить набір методів, здатних працювати у різних класах, не пов'язаних відношенням наслідування. Ознака схожа на модуль у інших мовах програмування, або на підмішаний клас у Python (прим. – *Ярошко С.*).

Метод *iconNamed:* класу *SpPresenter* використовує постачальника піктограм у Spec, щоб отримати значок з заданим іменем. Ви можете дослідити його, відкривши в системному оглядачі клас *SpPharoThemeIconProvider*, який є підкласом *SpIconProvider*. Кожна програма може визначити власного постачальника піктограм, оголосивши свій підклас *SpIconProvider*.

Логіка взаємодії демонстраторів

Тепер визначимо, що трапиться, коли користувач натисне кнопку. Опишемо це в окремому методі, який називається *connectPresenters*.

```
CustomerSatisfactionPresenter >> connectPresenters
   buttonHappy action: [ result label: buttonHappy label ].
   buttonNeutral action: [ result label: buttonNeutral label ].
   buttonBad action: [ result label: buttonBad label ]
```

Для того, щоб задати дію, яка буде виконана після натискання кнопки, використовують повідомлення *action:*. У нашій програмі зміниться текст напису *result*, щоб повідомити користувача, що його вибір зареєстровано. Зауважимо, що повідомлення *action:* є частиною API кнопки. В інших випадках уточнюють, яке повідомлення має бути надіслано вкладеному демонстратору, коли трапиться певна подія.

Підсумуємо:

- Спеціалізуйте метод *initializePresenters*, щоб визначити і налаштувати демонстратори, які є елементами вашого інтерфейсу користувача.
- Спеціалізуйте метод *connectPresenters*, щоб об'єднати ці демонстратори разом і визначити їхню взаємодію.

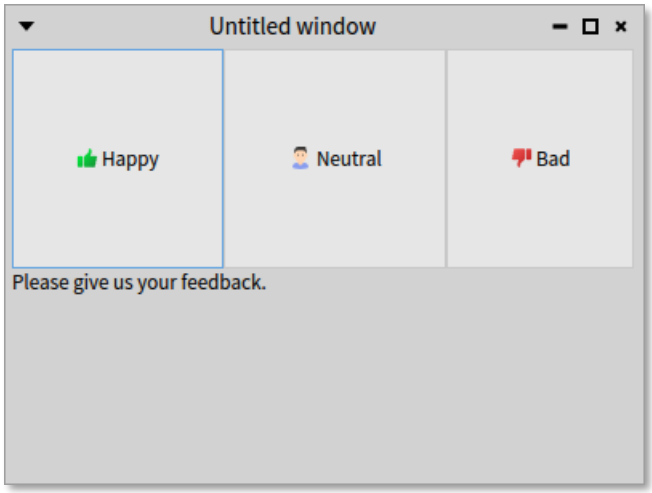


Рис. 2.2. Початкова версія інтерфейсу користувача для проведення опитування

Уточнення макета демонстратора

Вкладені демонстратори створено та налаштовано, але їхнє розташування у вікні застосунку все ще не визначено. Це завдання методу *defaultLayout*.

```
CustomerSatisfactionPresenter >> defaultLayout
   ^ SpBoxLayout newTopToBottom
      add: (SpBoxLayout newLeftToRight
         add: buttonHappy;
         add: buttonNeutral;
```

```
            add: buttonBad;
            yourself);
         add: result;
         yourself
```

У цьому макеті до інтерфейсу користувача додано два рядки: один з кнопками, а інший з написом, який містить запрошення або відповідь користувача. Визначення макета демонстратора є доволі складним процесом із багатьма можливими налаштуваннями, тому в цьому розділі не буде детального обговорення специфікації макета. Для отримання додаткової інформації зверніться до розділу 10.

Щойно метод *defaultLayout* визначено, можна відкрити графічний інтерфейс опитування за допомогою коду *CustomerSatisfactionPresenter new open*. Мало б з'явитися вікно як на рис. 2.2.

2.4. Визначення заголовка та розміру вікна, відкривання та закривання

Щоб задати заголовок вікна та початковий розмір демонстратора, потрібно спеціалізувати метод *initializeWindow:*, як показано нижче.

```
CustomerSatisfactionPresenter >> initializeWindow: aWindowPresenter
   super initializeWindow: aWindowPresenter.
   aWindowPresenter
      title: 'Customer Satisfaction Survey';
      initialExtent: 400@100
```

Щоб довідатися заголовок і розмір демонстратора, можна використовувати допоміжні методи. Коли знову відкрити демонстратор і натиснути кнопку ***Happy***, то вікно матиме вигляд як на рис. 2.3.

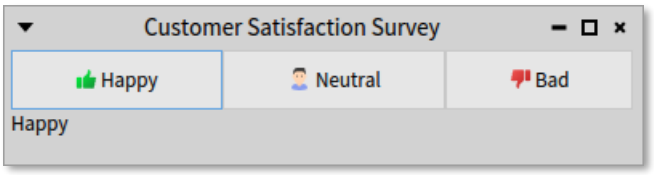


Рис. 2.3. Остаточний варіант графічного інтерфейсу для проведення опитування

Надсилання демонстраторові повідомлення *open* відкриває вікно і повертає екземпляр класу *SpWindowPresenter*. Це дає змогу закривати вікно програмно.

```
| ui |
ui := CustomerSatisfactionPresenter new open.
[ ... робити все, що треба,
      аж доки не виникне потреба закрити вікно ...]
ui close
```

Зазначимо, що для оновлення вмісту відкритого вікна, використовують метод *SpPresenter >> withWindowDo:*, але про нього йтиметься пізніше. Більше інформації про керування вікнами, наприклад, про відкриття діалогових вікон або про налаштування тексту «Про програму», можна знайти в розділі 9.

На цьому наш перший приклад інтерфейсу користувача Spec завершується. У наступному розділі продовжимо розглядати приклади того, як налаштувати різні демонстратори, які можна використовувати в інтерфейсі користувача.

2.5. Підсумки розділу

У цьому розділі наведено невеликий приклад побудови інтерфейсу користувача. Описано, які кроки потрібно виконати, щоб побудувати графічний інтерфейс користувача засобами Spec.

Від перекладача. Нагадаємо ці кроки:

- оголосити підклас класу *SpPresenter*, оголосити в ньому змінні екземпляра для вкладених демонстраторів і визначити методи:
  - *initializePresenters*, щоб створити та налаштувати візуальні компоненти;
  - *connectPresenters*, щоб описати взаємодію компонентів;
  - *defaultLayout*, щоб задати розташування компонентів;
  - *initializeWindow:*, щоб уточнити параметри вікна застосунку.

Більше прикладів інтерфейсів користувача Spec можна знайти в образі системи Pharo. Всі інтерфейси користувача Spec є підкласами *SpPresenter*, тому їх легко знайти, і кожен з них може бути прикладом. Ба більше, експериментувати з демонстраторами та інтерфейсами користувача дуже легко, бо кожного з них можна відкрити в окремому вікні.

Рекомендуємо прочитати принаймні розділ 7 про повторне використання демонстраторів Spec, що є основним секретом потужності Spec. Ці знання допоможуть вам швидше створювати інтерфейси користувача завдяки глибшому розумінню способів використання стандартних компонентів, а також дадуть змогу повторно використовувати ваші власні інтерфейси користувача.

Розділ 3

Більша частина Spec в одному прикладі

Цей розділ проведе через створення простого, але нетривіального додатка для зберігання інформації про фільми. Вікно додатка зображено на рис. 3.1. У розділі описано багато аспектів Spec, детальному розгляду яких присвячено решту цієї книги: застосунок, демонстратори, поділ між доменом і демонстратором, макет, перенесення даних з метою підключення візуальних компонентів і стилі.

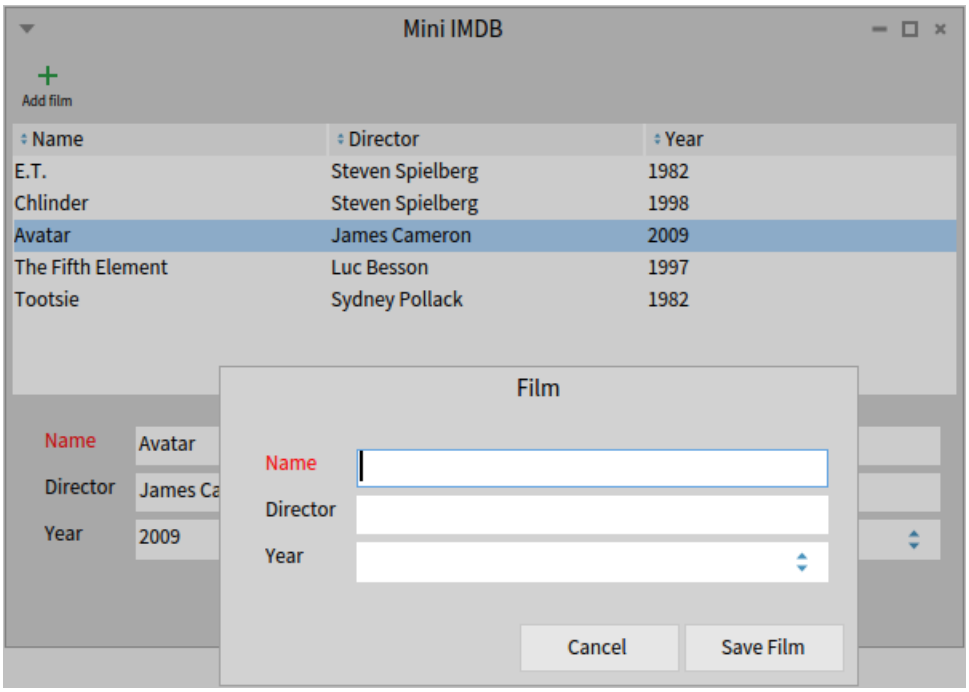


Рис. 3.1. Застосунок ImdbApp: повторне використання одного компонента для перегляду та редагування даних про фільм

3.1. Застосунок

Spec 2.0 запроваджує концепцію застосунку. Застосунок (application) – це невеликий об'єкт, який відповідає за збереження стану програми. Він керує, наприклад, вікнами, які становлять програму, та їхніми графічними бібліотеками (Morphic або GTK), а також може зберігати спільні властивості демонстраторів вікон.

Почнемо з визначення класу застосунку:

```
SpApplication << #ImdbApp
   package: 'CodeOfSpec20Book'
```

3.2. Базова модель фільму

Програма керуватиме інформацією про фільми, тому визначимо клас *ImdbFilm* так, щоб він зберігав назву фільму, рік і режисера. Pharo допоможе автоматично згенерувати відповідні селектори та модифікатори.

```
Object << #ImdbFilm
   slots: { #name . #year . #director };
   package: 'CodeOfSpec20Book'
```

Потрібно визначити спосіб, як програма зберігатиме інформацію про фільми. Адже треба мати змогу доповнювати, редагувати її. Можна було б покликати на допомогу Voyage (https://github.com/pharo-nosql/voyage) і налаштувати зв'язок з базою даних MongoDB, або, простіше, скористатися можливістю Voyage імітувати базу даних у пам'яті комп'ютера. Але у Pharo можна зробити ще простіше: дані можна зберігати в образі системи разом з класом. Для цього визначимо змінну *films* класу. Вона міститиме колекцію фільмів. До речі, так у Pharo реалізують патерн проєктування Одинак.

```
Object class << ImdbFilm
   class slots: { #films }
```

Визначимо метод-селектор, який виконує відкладену ініціалізацію змінної *films*.

```
ImdbFilm class >> films
   ^ films ifNil: [ films := OrderedCollection new ]
```

І на завершення визначимо метод додавання фільму до колекції.

```
ImdbFilm class >> addFilm: aFilm
   films add: aFilm
```

Тепер ми готові визначити перший демонстратор, який відображатиме список фільмів.

3.3. Список фільмів

Щоб визначити демонстратор для керування списком фільмів, оголосимо новий клас *ImdbFilmListPresenter*, який наслідує *SpPresenter*. Змінна екземпляра *filmList* міститиме вкладений демонстратор таблиці для структурованого відображення списку фільмів.

```
SpPresenter << #ImdbFilmListPresenter
   slots: { #filmList };
   package: 'CodeOfSpec20Book'
```

Розташування вкладених демонстраторів визначає метод *defaultLayout*. Використаємо простий послідовний вертикальний макет з єдиним елементом *filmList*.

defaultLayout

```
ImdbFilmListPresenter >> defaultLayout
   ^ SpBoxLayout newTopToBottom
      add: filmList;
      yourself
```

Якщо не визначено жодних інших методів, які описують макет, то *defaultLayout* автоматично викликається інфраструктурою Spec.

Демонстратор може містити піддемонстратори. *ImdbFilmListPresenter* містить демонстратор таблиці, і пізніше ви побачите, що:

1) демонстратор може мати кілька макетів;
2) макети можна визначати динамічно.

У Spec макети за замовчуванням динамічні, їх визначають методом екземпляра. Задля сумісності з попередньою версією Spec все ще можна замість методу екземпляра *ImdbFilmListPresenter >> defaultLayout* визначити метод, який повертає макет, як метод класу *ImdbFilmListPresenter class >> defaultLayout*, але це не рекомендований спосіб.

initializePresenters

Вкладений демонстратор *filmList* поки що не ініціалізовано.

Місцем для ініціалізації піддемонстраторів є метод *initializePresenters*, як показано нижче. Там визначено, що *filmList* – це таблиця з трьома стовпцями. Повідомлення *newTable* створює екземпляр *SpTablePresenter*.

```
ImdbFilmListPresenter >> initializePresenters
   filmList := self newTable
      addColumn: (SpStringTableColumn title: 'Name'
         evaluated: #name) beSortable;
      addColumn: (SpStringTableColumn title: 'Director'
         evaluated: #director) beSortable;
      addColumn: (SpStringTableColumn title: 'Year'
         evaluated: #year) beSortable;
      yourself
```

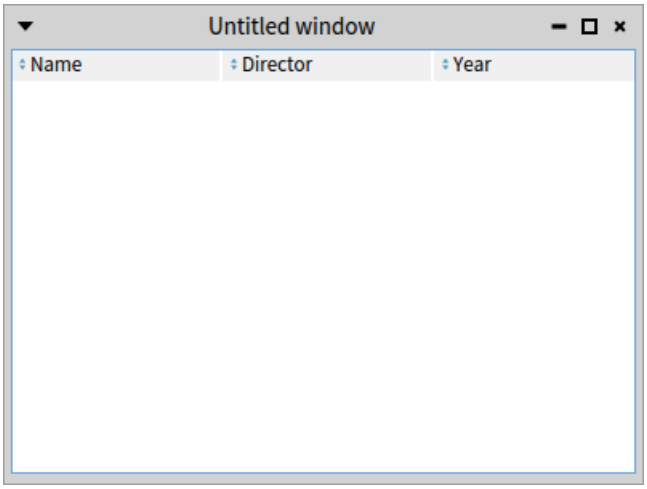


Рис. 3.2. Порожній список фільмів, створений методом *initializePresenters*

Наступний вираз створює екземпляр демонстратора списку фільмів і відкриває його. Отримаємо вікно, як на рис. 3.2.

```
ImdbFilmListPresenter new open
```

3.4. Заповнення списку фільмів

Spec влаштовано так, що після методу *initializePresenters* автоматично викликається метод *updatePresenter*. Визначимо його так, щоб він отримував з моделі даних (*ImdbFilm*) список записаних фільмів і заповнював вкладену таблицю. На цей момент *ImdbFilm* не зберігає жодного фільму в колекції, тому таблиця залишається порожньою.

```
ImdbFilmListPresenter >> updatePresenter
   filmList items: ImdbFilm films
```

За бажання можна додати фільм до колекції і повторно відкрити демонстратор. Доданий фільм мав би відобразитися в таблиці.

```
ImdbFilm addFilm: (ImdbFilm new
   name: 'E.T.';
   director: 'Steven Spielberg';
   year: '1982';
   yourself)
```

3.5. Відкривання демонстраторів за допомогою застосунку

Зазвичай безпосереднє створення демонстратора практикують під час розробки, а усталений спосіб відкрити демонстратор – попросити застосунок за допомогою повідомлення *newPresenter:*, як показано нижче.

```
| app |
app := ImdbApp new.
(app newPresenter: ImdbFilmListPresenter) open
```

Застосунок відповідає за керування вікнами та іншою інформацією, тому важливо використовувати його для створення демонстраторів, які входять до складу програми.

3.6. Покращення вигляду вікна

Демонстратор можна вбудувати в інший демонстратор, про що йтиметься пізніше. Його також можна помістити у вікно, що саме й робить повідомлення *open*. Spec надає ще один метод-зачіпку, *initializeWindow:*, щоб налаштувати параметри вікна, в якому відображатиметься демонстратор.

Рис. 3.3. Демонстратор списку фільмів у вікні, оздобленому панеллю інструментів

Метод *initializeWindow:* дає змогу визначити заголовок, усталений розмір (повідомлення *initialExtent:*) і панель інструментів.

```
ImdbFilmListPresenter >> initializeWindow: aWindowPresenter
| addButton toolbar |
addButton := self newToolbarButton
   label: 'Add film';
   icon: (self iconNamed: #smallAdd);
   action: [ self addFilm ];
   yourself.
toolbar := self newToolbar
   add: addButton;
   yourself.
aWindowPresenter
   title: 'Mini IMDB';
   initialExtent: 600@400;
   toolbar: toolbar
```

Ви мали б отримати вікно з панеллю інструментів, як на рис. 3.3. Щоб переконатися, що кнопка ***Add film*** не спричиняє помилки через виклик методу *addFilm*, оголосимо його порожнім. Згодом, щоб мати змогу задати інформацію про новий фільм, створимо окремий демонстратор.

```
ImdbFilmListPresenter >> addFilm
   "empty for now"
```

Як описано в розділі 18, панелі інструментів можна автоматично створювати з команд. Можна було б і тут застосувати такий підхід, зробити панель інструментів частиною *ImdbFilmListPresenter*, використавши для цього змінну екземпляра. Це можливо, бо, подібно до демонстратора таблиці чи інших попередньо визначених демонстраторів, панель інструментів також є демонстратором. Але робити так – означає робити програмний код менш модульним. Зауважимо також, що створену панель інструментів можна описати окремим класом, щоб полегшити її повторне використання.

3.7. Застосунок керує піктограмами

З визначення методу *initializeWindow:* видно, що застосунок керує піктограмами за допомогою повідомлення *iconNamed:*. Насправді, клас демонстратора делегує опрацювання повідомлення *iconNamed:* своєму застосункові. До того ж, кожен застосунок може визначити власний набір піктограм за допомогою повідомлення *iconProvider:*.

3.8. Демонстратор фільму

Настав час визначити простий демонстратор для відображення інформації про фільм. Використаємо його, щоб додати до колекції новий фільм або показати наявний. Для цього створимо підклас *SpPresenter*'а і назвемо його *ImdbFilmPresenter*. Він матиме три змінні екземпляра: *nameText*, *directorText* і *yearNumber*.

```
SpPresenter << #ImdbFilmPresenter
   slots: { #nameText . #directorText . #yearNumber };
   package: 'CodeOfSpec20Book'
```

Як і раніше, визначимо стандартний макет демонстратора. Цього разу скористаємося макетом сітки. З його допомогою можна вибрати клітинку сітки, в якій відображатиметься вкладений демонстратор.

```
ImdbFilmPresenter >> defaultLayout
^ SpGridLayout new
   add: 'Name' at: 1@1; add: nameText at: 2@1;
   add: 'Director' at: 1@2; add: directorText at: 2@2;
   add: 'Year' at: 1@3; add: yearNumber at: 2@3;
   yourself
```

Зауважимо, що тепер не потрібно створювати всі методи доступу до елементів демонстратора, як це було в Spec 1.0. Достатньо визначити лише селектори, оскільки вони знадобляться під час створення відповідного екземпляра *ImbdFilm*.

```
ImdbFilmPresenter >> year
^ yearNumber text

ImdbFilmPresenter >> director
^ directorText text

ImdbFilmPresenter >> name
^ nameText text
```

Для зручності *SpGridLayout* підтримує ще один спосіб побудови, за якого елементи до макета додають у тому порядку, в якому вони з'являтимуться. Попереднє визначення макета можна переписати так:

```
ImdbFilmPresenter >> defaultLayout
^ SpGridLayout build: [ :builder |
   builder
      add: 'Name'; add: nameText; nextRow;
      add: 'Director'; add: directorText; nextRow;
      add: 'Year'; add: yearNumber ]
```

Зверніть увагу на таку особливість: не додавайте сюди повідомлення *yourself*, бо тоді повернеться клас, а не екземпляр макета.

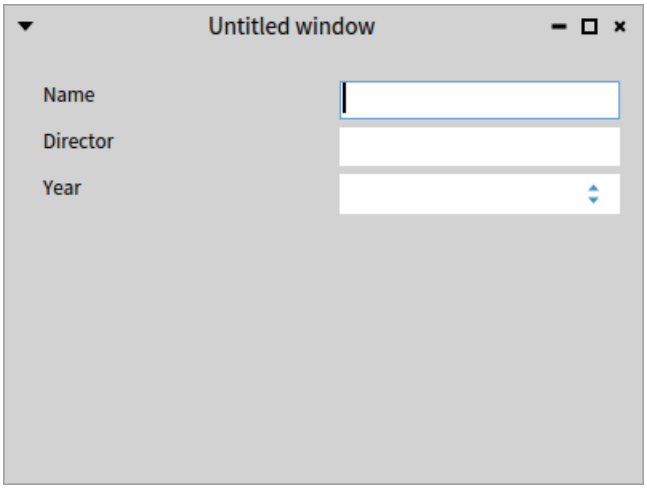


Рис. 3.4. Окремий демонстратор фільму

Як і раніше, визначимо метод *initializePresenters*, щоб ініціалізувати змінні екземпляра відповідними вкладеними демонстраторами. Тут *nameText* і *directorText* міститимуть компоненти для введення тексту, а *yearNumber* – для введення числа.

```
ImdbFilmPresenter >> initializePresenters
   nameText := self newTextInput.
   directorText := self newTextInput.
   yearNumber := self newNumberInput
      rangeMinimum: 1900 maximum: Year current year;
      yourself
```

Тепер можна випробувати наш застосунок, виконавши наведений нижче фрагмент коду, й отримати вікно, подібне до того, що зображено на рис. 3.4.

```
| app |
app := ImdbApp new.
(app newPresenter: ImdbFilmPresenter) open
```

3.9. Покращення вигляду демонстратора фільму

Щоб покращити зовнішній вигляд демонстратора фільмів, можна належно налаштувати поведінку стовпців і задати властивості вікна. Видно, що написи на формі для відображення даних фільму займають занадто багато місця, а компоненти для введення даних відірвані від них. Макет сітки усталено розподіляє стовпці однакового розміру, тому написи займають половину ширини форми. Проблему можна вирішити, використавши неоднорідні стовпці та попросивши другий стовпець зайняти найбільшу можливу ширину за допомогою повідомлення *column:expand:*, як у коді нижче. Результат зображено на рис. 3.5.

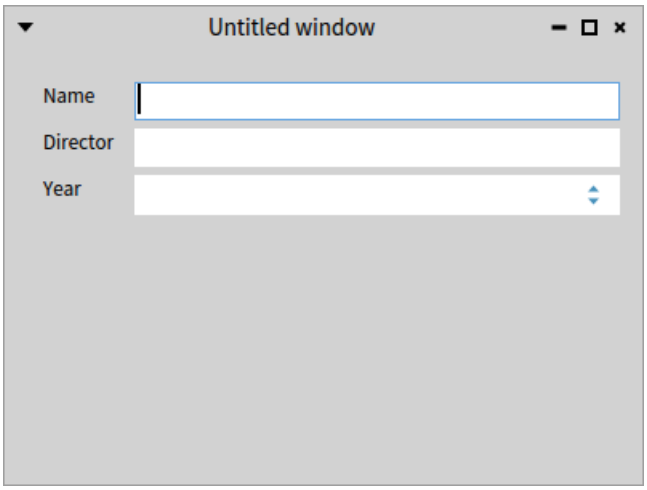


Рис. 3.5. Використання макета з неоднорідною сіткою

```
ImdbFilmPresenter >> defaultLayout

^ SpGridLayout build: [ :builder |
   builder
      beColumnNotHomogeneous;
      column: 2 expand: true;
      add: 'Name'; add: nameText; nextRow;
      add: 'Director'; add: directorText; nextRow;
      add: 'Year'; add: yearNumber ]
```

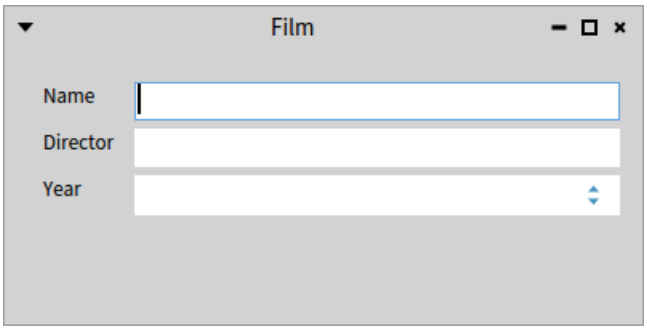


Рис. 3.6. Покращене вікно

А тепер встановимо властивості вікна, оголосивши новий метод *initializeWindow:*, як записано нижче. Отримаємо вікно, зображене на рис. 3.6.

```
ImdbFilmPresenter >> initializeWindow: aWindowPresenter
aWindowPresenter
   title: 'Film';
   initialExtent: 400@250
```

3.10. Відкривання демонстратора фільму в вікні модального діалогу

Окрім того, що демонстратор фільму можна відкривати у вікні застосунку, опишемо, як його відкрити у вікні модального діалогу. Модальний діалог блокує інтерфейс користувача, доки користувач не підтвердить або не скасує діалог. Модальне вікно не має стандартних кнопок звичайних вікон і його не можна переміщувати по екрану. Відомо, що звичайне вікно відкривають надсиланням повідомлення *open* екземплярові демонстратора, натомість діалог відкривають повідомленням *openModal*.

```
| app |
app := ImdbApp new.
(app newPresenter: ImdbFilmPresenter) openModal
```

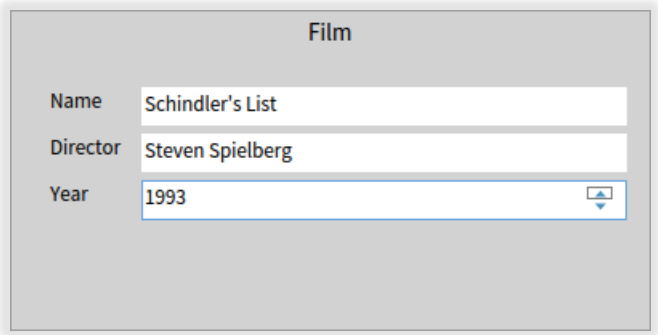


Рис. 3.7. Модальний діалог

Рис. 3.7 демонструє результат. Зауважте, що немає компонентів інтерфейсу користувача для закривання діалогу. Натисніть клавішу [*Esc*], щоб закрити його.

3.11. Налаштування модального діалогу

Spec дає змогу пристосувати діалогове вікно до власних потреб, наприклад, додати кнопки для взаємодії з користувачем. Для цього перевизначають метод *initializeDialog-Window:*. Додамо дві кнопки, які керуватимуть завершенням діалогу, як показано на рис. 3.8. Задамо також розташування діалогового вікна в центрі екрана, надіславши повідомлення *centered* демонстратору діалогу.

```
ImdbFilmPresenter >> initializeDialogWindow: aDialogPresenter
   aDialogPresenter centered.
   aDialogPresenter
      addButton: 'Cancel' do: [ :button | button close ];
      addButton: 'Save Film' do: [ :button | button beOk; close ]
```

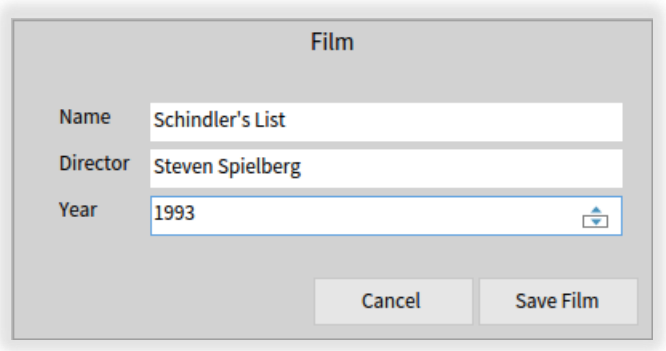


Рис. 3.8. Модальний діалог з додатковими налаштуваннями

3.12. Виклик діалогу

Все готово для того, щоб використати демонстратор фільму в демонстраторі списку фільмів. Наповнимо справжнім змістом раніше визначений «порожній» метод *addFilm* у класі *ImdbFilmListPresenter*. Він спрацьовує, коли користувач натискає кнопку ***Add film***. Отож створимо новий демонстратор фільму і пов'яжемо його з поточним застосунком. Відкриємо демонстратор фільму як модальне діалогове вікно за допомогою повідомлення *openModal*. Коли користувач натисне кнопку ***Save Film*** діалогу, новий фільм буде додано до імпровізованої бази даних, і демонстратор списку фільмів оновить свій зовнішній вигляд.

```
ImdbFilmListPresenter >> addFilm

| dialog windowPresenter film |
dialog := ImdbFilmPresenter newApplication: self application.
windowPresenter := dialog openModal.
windowPresenter isOk ifFalse: [ ^ self ].

film := ImdbFilm new
   name: dialog name;
   director: dialog director;
   year: dialog yearNumber.
ImdbFilm addFilm: film.
self updatePresenter
```

Тепер можна відкрити застосунок зі списком фільмів і натиснути кнопку ***Add film***, щоб відкрити діалог.

```
app := ImdbApp new.
(app newPresenter: ImdbFilmListPresenter) open
```

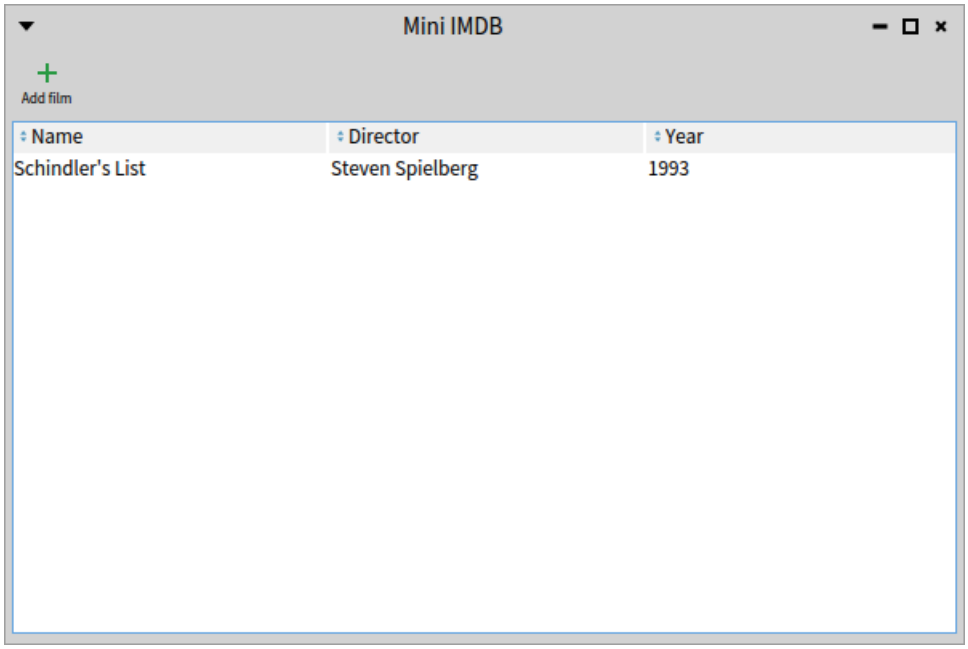


Рис. 3.9. Оновлений список фільмів

Натискання клавіші [*Esc*], або кнопки ***Cancel*** тільки закриває діалог і ніяк не впливає на перелік фільмів у вікні застосунку. Якщо ж ввести дані фільму та натиснути кнопку ***Save Film***, то ви побачите, що екземпляр *ImdbFilmListPresenter* оновлено доданим фільмом, як зображено на рис. 3.9.

3.13. Вбудовування демонстратора фільму у демонстратор списку фільмів

Маємо два основні візуальні компоненти: список фільмів і деталі фільму. Можна припустити, що корисно було б бачити деталі фільму в тому самому контейнері, що й список, наприклад тому, що деталі можуть потребувати більше місця, ніж є у стовпці таблиці.

Щоб розташувати демонстратор фільму поруч з таблицею, додамо нову змінну екземпляра *detail* до класу *ImdbFilmListPresenter*.

```
SpPresenter << #ImdbFilmListPresenter
   slots: { #filmList . #detail };
   package: 'CodeOfSpec20Book'
```

Перевизначимо стандартний макет. Згодом продемонструємо, що можна використовувати різні макети.

```
ImdbFilmListPresenter >> defaultLayout
^ SpBoxLayout newTopToBottom
   add: filmList;
```

```
        add: detail;
        yourself
```

Демонстратор фільму в діалозі використовують для введення деталей фільму, а у вікні застосунку – тільки для відображення, тому потрібно додати метод до класу демонстратора, щоб контролювати режим редагування.

```
ImdbFilmPresenter >> editable: aBoolean
   nameText editable: aBoolean.
   directorText editable: aBoolean.
   yearNumber editable: aBoolean
```

Доповнимо метод *initializePresenters* класу *ImdbFilmListPresenter*.

- По-перше, потрібно створити екземпляр *ImdbFilmPresenter*.
- По-друге, налаштувати його в режим лише для читання повідомленням *editable: false*.
- По-третє, демонстратор фільму мав би відобразити деталі, коли вибрано елемент списку фільмів. Таку поведінку можна задати і в методі *initializePresenters*, але у Spec вважається за краще робити це в методі *connectPresenters* (див. параграф 3.14).

```
ImdbFilmListPresenter >> initializePresenters
filmList := self newTable
   addColumn: (SpStringTableColumn title: 'Name'
      evaluated: #name);
   addColumn: (SpStringTableColumn title: 'Director'
      evaluated: #director);
   addColumn: (SpStringTableColumn title:
      'Year' evaluated: #year);
   yourself.
detail := self instantiate: ImdbFilmPresenter.
detail editable: false
```

3.14. Визначення взаємодії компонентів

Щоб мати змогу передати деталі фільму у демонстратор і відповідно заповнити його, додамо допоміжний метод *setModel:* у клас *ImdbFilmPresenter*.

```
ImdbFilmPresenter >> setModel: aFilm
   aFilm
      ifNil: [
         nameText text: ''.
         directorText text: ''.
         yearNumber number: '' ]
      ifNotNil: [
         nameText text: aFilm name.
         directorText text: aFilm director.
         yearNumber number: aFilm year ]
```

Важливо перевірити значення *nil*, інакше надсилання повідомлень *name*, *director* чи *year* може завершитися помилкою. Якщо переданий аргумент *aFilm* дорівнює *nil*, то потрібно очистити всі три піддемонстратори.

Зауважимо, що метод *setModel:* не потрібний, якщо ви наслідуєте клас демонстратора від *SpPresenterWithModel*. Якщо ж ви наслідуєте підклас від *SpPresenter*, то це єдиний спосіб ініціалізувати модель перед налаштуванням демонстратора й уникнути помилок під час його відкриття.

Взаємодію демонстраторів визначають у методі *connectPresenters*. Оголосимо його так, щоб, коли користувач вибере елемент списку, демонстратор фільму відобразив інформацію про нього. Варто приділити трохи часу вивченню методу *whenSelectionChangedDo:*.

Метод *whenSelectionChangedDo:* приймає як аргумент блок щонайбільше з одним параметром. Параметр блока містить не безпосередньо виділений елемент, а складніший об'єкт, який описує виділення. Справді, виділення в списку з одиничним вибором відрізняється від такого в списку з множинним вибором. Тому Spec визначає концепцію режиму вибору у формі підкласів *SpAbstractSelectionMode*.

```
ImdbFilmListPresenter >> connectPresenters
   filmList whenSelectionChangedDo: [ :selectedItemMode |
      detail setModel: selectedItemMode selectedItem ]
```

Коли *connectPresenters* відкомпільовано, вибір елемента в списку призводить до відображення деталей вибраного елемента, як показано на рис. 3.10.

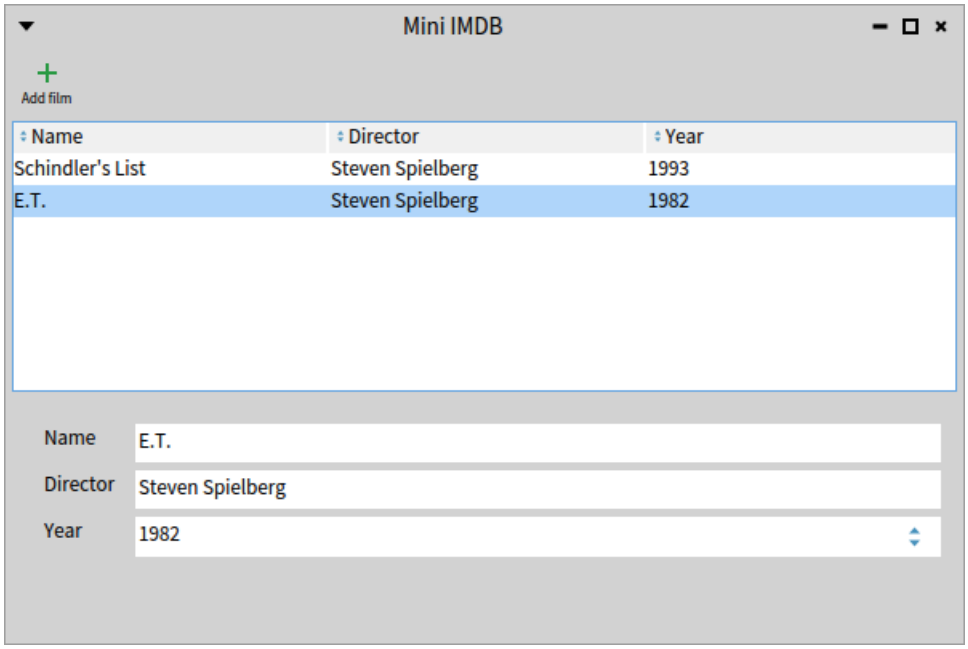


Рис. 3.10. Вбудовування опису фільму у вікно застосунку: вибір елемента списку заповнює візуальний компонент деталізації

3.15. Тестування графічного інтерфейсу користувача застосунку

Однією з сильних властивостей Spec є можливість писати тести для перевірки взаємодії та логіки інтерфейсу користувача. Підтвердимо, що написання тестів для інтерфейсу користувача на стільки ж просте, на скільки потужно допомагає створювати гарні проєкти та виявляти помилки.

Визначимо *ImdbFilmListPresenterTest* як підклас *TestCase*.

```
TestCase << #ImdbFilmListPresenterTest
   package: 'CodeOfSpec20Book'

ImdbFilmListPresenterTest >> testWhenSelectingOneFilmThenDetailIsUpdated
   | presenter detail |
"Arrange"
   presenter := ImdbFilmListPresenter new.
   presenter open.
   detail := presenter detail.
   self assert: detail name isEmpty.
"Act"
   presenter clickFilmAtIndex: 1.
"Assert"
   self deny: detail name isEmpty.
   presenter delete
```

Видно, що для підтримки належного тестування доведеться визначити два методи в класі *ImdbFilmListPresenter*: селектор для *detail* і метод взаємодії *clickFilmAtIndex:*. Зачислимо їх до протоколу «*testing support*», щоб зазначити, що вони потрібні лише для цілей тестування.

```
ImdbFilmListPresenter >> detail
   ^ detail

ImdbFilmListPresenter >> clickFilmAtIndex: anIndex
   filmList clickAtIndex: anIndex
```

Це трохи недосконалий тест, бо він не перевіряє конкретне значення назви фільму у відповідному полі демонстратора. Зроблено так заради простоти налаштування тесту, а також частково тому, що *ImdbFilm* зберігає колекцію фільмів у полі класу. Одинак – не найліпше рішення, до того ж він ускладнює тестування.

Визначимо допоміжні методи класу *ImdbFilm* для скидання збережених фільмів і додавання фільму «E.T.». Клас доведеться доповнити також полем *copy* для зберігання робочої колекції фільмів на час виконання тестів[3].

```
ImdbFilm class >> reset
   copy := films.
   films := OrderedCollection new

ImdbFilm class >> addET
   films add: self ET

ImdbFilm class >> ET
   ^ self new
      name: 'E.T.';
      director: 'Steven Spielberg';
      year: '1982';
      yourself

ImdbFilm class >> restore
   films := copy.
   copy := nil
```

[3] Змінна *copy* та метод *reset* запозичені з попередньої версії книги (прим. – *Ярошко С.*).

Тепер можна визначити метод *setUp*.

```
ImdbFilmListPresenterTest >> setup
   super setUp.
   ImdbFilm reset.
   ImdbFilm addET
```

Оновимо тест, щоб зберегти відкритий демонстратор у змінній екземпляра. Це дасть змогу визначити метод *tearDown*, який завжди закриватиме демонстратор, незалежно від того, чи завершився тест успішно, чи ні.

```
ImdbFilmListPresenterTest >> testWhenSelectingOneFilmThenDetailIsUpdated
   | detail |
"Arrange"
   presenter := ImdbFilmListPresenter new.
   presenter open.
   detail := presenter detail.
   self assert: detail name isEmpty.
"Act"
   presenter clickFilmAtIndex: 1.
"Assert"
   self assert: detail name equals: 'E.T.'

ImdbFilmListPresenterTest >> teardown
   presenter ifNotNil: [ presenter delete ].
   ImdbFilm restore.
   super tearDown
```

3.16. Більше тестів

Тести забезпечують спокійне життя, тому що можна змінювати програму і швидко перевіряти, чи вона все ще працює, зменшуючи стрес програміста. Тож напишемо ще кілька.

Додамо ще один метод-селектор для підтримки тестів.

```
ImdbFilmListPresenter >> filmList
   ^ filmList
```

Перевіримо, чи список містить один фільм, чи очищується назва фільму в демонстраторі, якщо вибирати неіснуючий індекс.

```
ImdbFilmListPresenterTest >> testNoSelectionClearsDetails
"Arrange"
   presenter := ImdbFilmListPresenter new.
   presenter open.
"Act"
   presenter clickFilmAtIndex: 1.
"Assert"
   self assert: presenter filmList listSize equals: 1.
   self deny: presenter detail name isEmpty.
"Act"
   presenter clickFilmAtIndex: 2.
"Assert"
   self assert: presenter detail name equals: ''
```

Множинний вибір елементів списку не підтримується. Тому перевіримо, чи *filmList* налаштовано для одиночного вибору. Немає методу *isSingleSelection*, тому замість підтвердження можливості одиночного вибору пересвідчимося, що у множинному виборі відмовлено.

```
ImdbFilmListPresenterTest >> testListIsSingleSelection
   presenter := ImdbFilmListPresenter new.
   presenter open.
   self deny: presenter filmList isMultipleSelection
```

Видно, що досить легко перевірити, чи належно застосунок взаємодіє з користувачем.

3.17. Заміна макета

У Spec демонстратор може мати кілька макетів, навіть створених на льоту, як побачимо незабаром у прикладі динамічної зміни макета. Який макет використати, можна вирішити навіть під час відкривання демонстратора. Проілюструймо це. Уявіть, що ми надаємо перевагу розташуванню списку фільмів під деталями фільму, або хочемо бачити лише список.

```
ImdbFilmListPresenter >> listBelowLayout
   ^ SpBoxLayout newTopToBottom
      add: detail;
      add: filmList;
      yourself
```

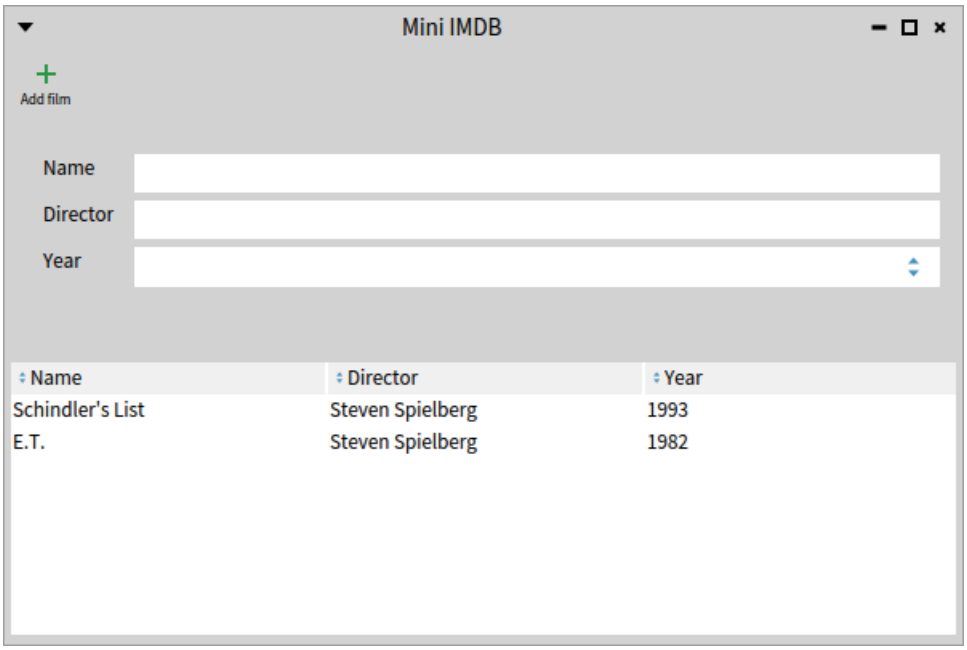


Рис. 3.11. Демонстратор може використовувати кілька макетів для керування розташуванням вкладених компонентів

Приклад демонструє, що можна відкрити *ImdbFilmListPresenter* за допомогою щойно визначеного макета *listBelowLayout* (див. рис.3.11).

```
| app presenter |

app := ImdbApp new.
presenter := app newPresenter: ImdbFilmListPresenter.
presenter openWithLayout: presenter listBelowLayout
```

Можна також визначити макет із частиною піддемонстраторів. Тут *listOnlyLayout* показує лише список.

```
ImdbFilmListPresenter >> listOnlyLayout
   ^ SpBoxLayout newTopToBottom
      add: filmList;
       yourself
```

Наступний приклад демонструє, що можна відкрити *ImdbFilmListPresenter* з одним макетом і динамічно замінити його іншим. Виконайте в Пісочниці Pharo фрагмент коду без оголошення тимчасових змінних. Так вони будуть пов'язані з робочим вікном і зберігатимуться аж до його закриття.

```
app := ImdbApp new.
presenter := app newPresenter:ImdbFilmListPresenter.
presenter open
```

Демонстратор відкриється зі стандартним макетом. Тепер виконайте в тій же Пісочниці рядок.

```
presenter layout: presenter listOnlyLayout
```

Тієї ж миті з вікна застосунку зникне демонстратор деталей фільму, і залишиться тільки список. Видно, що макет з одним лише списком було застосовано динамічно.

3.18. Використання перенесень

Spec 2.0 впроваджує чудову концепцію перенесення виокремлення від одного демонстратора до іншого, уділяючи більше уваги «потокові» інформації, ніж деталям реалізації цього перенесення, які можуть змінюватися від демонстратора до демонстратора.

За наявності перенесень кожен демонстратор може визначити набір вихідних портів (порти для передавання інформації) і вхідних портів (порти для отримання інформації). У демонстраторах візуальних компонентів уже визначено вихідні/вхідні порти, які можна використовувати для взаємодії з ними, але також можна додавати порти до власних демонстраторів.

Найпростіший спосіб оголосити перенесення – надіслати повідомлення *transmitTo:* від одного демонстратора до іншого. Тепер можна змінити метод *connectPresenters*: використаймо в ньому перенесення.

```
ImdbFilmListPresenter >> connectPresenters
   filmList transmitTo: detail
```

Тут *filmList* – демонстратор таблиці, який переноситиме виокремлення свого рядка до демонстратора деталей фільму.

Подальші визначення методів потребують пояснень. *ImdbFilmPresenter* – це спеціальний демонстратор. Spec не знає, як «заповнити» його вхідними даними. Тому потрібно повідомити Spec, що вхідним портом *ImdbFilmPresenter* буде модель, і вхідні дані отримуватиме вона. З цією метою вхідним портом призначимо екземпляр *SpModelPort*.

```
ImdbFilmPresenter >> inputModelPort
   ^ SpModelPort newPresenter: self

ImdbFilmPresenter >> defaultInputPort
   ^ self inputModelPort
```

Зауважимо, що замість двох методів можна було б оголосити один *defaultInputPort*, якщо зробити його тілом визначення методу *inputModelPort*.

Вхідні дані буде опрацьовано методом *setModel:*, який вже визначили раніше в класі *ImdbFilmPresenter*. Екземпляр *SpModelPort* подбає про це.

Тепер можна відкрити застосунок і пересвідчитися, що все працює належно.

```
| app |
app := ImdbApp new.
(app newPresenter: ImdbFilmListPresenter) open
```

3.19. Використання стилів для оформлення застосунку

Візуальні компоненти інтерфейсу користувача застосунку можуть мати свої правила налаштування і відображення, наприклад, щоб задати розмір або колір шрифту для заголовків. Щоб підтримати таку можливість, Spec вводить концепцію *стилів* для компонентів.

У Spec застосунок визначає таблицю стилів (або їх набір). Таблиця стилів визначає набір «класів стилю», які можна призначити візуальним компонентам демонстратора. Однак для кожної графічної бібліотеки клас стилю визначають по-різному. Тоді, як GTK для стилізації компонентів приймає (зазвичай) традиційний CSS, Morphic використовує власний каркас стилів.

Новостворений застосунок має усталену конфігурацію і стандартну таблицю стилів. Якщо немає потреби стилізувати програму особливо, то немає потреби їх перевизначати. В описаному прикладі хотілося б визначити стиль заголовка, щоб налаштувати деякі написи. У Spec кожен демонстратор розуміє повідомлення *addStyle:*, яке додає тег (клас CSS) до приймача.

Щоб виконати задумане, потрібно оголосити таблицю стилів у екземплярі конфігурації у відповідному методі. Саму конфігурацію потрібно пов’язати з застосунком. Стилі додають до демонстраторів, тому для стилізованого напису створимо новий демонстратор і збережемо його в полі екземпляра *nameLabel* класу *ImdbFilmPresenter* (поле доведеться додати в оголошення класу). Цей демонстратор позначимо власним класом CSS за допомогою повідомлення *addStyle:*. Наш клас CSS називатиметься «*customLabel*».

Спочатку створимо спеціальну конфігурацію для застосунку.

```
SpMorphicConfiguration << #ImdbConfiguration
   package: 'CodeOfSpec20Book'
```

Тоді використаємо її в *ImdbApp*.

```
ImdbApp >> initialize
   super initialize.
   self useBackend: #Morphic
      with: ImdbConfiguration new
```

А тоді можна визначати власні стилі. Найпростіший спосіб – це створити стиль із рядка. Тут зазначено, що позначений тегом *customLabel* візуальний елемент матиме напис червоного кольору.

```
ImdbConfiguration >> customStyleSheet
 ^ '
.application [
   .customLabel [ Font { #color: #red } ] ]'
```

Припильнуйте, щоб не загубити крапку «.» перед *application* та *customLabel*.

Перевизначимо метод *configure:* так, щоб він містив настроюваний стиль:

```
ImdbConfiguration >> configure: anApplication
   super configure: anApplication.
   self addStyleSheetFromString: self customStyleSheet
```

Все готово, щоб використати стиль для напису. У попередній версії методу *initialize-Presenters* Spec автоматично створював демонстратори для написів, але розробник не мав доступу до них. Тому виникла потреба явно додати демонстратор напису, щоб можна було позначити його CSS-подібним класом за допомогою повідомлення *addStyle: 'customLabel'*, як показано нижче.

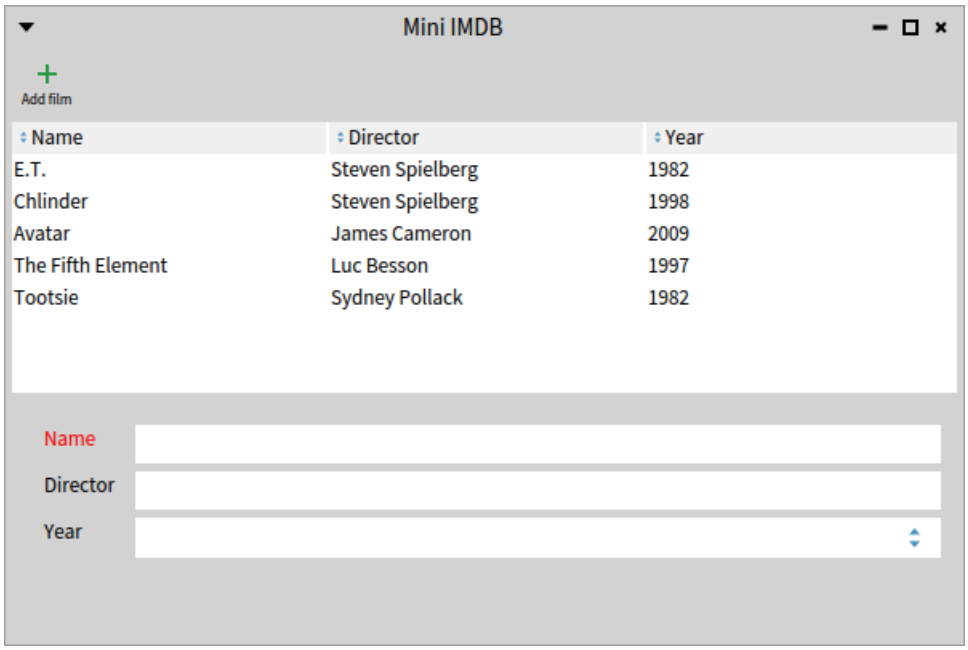


Рис. 3.12. Стилізований напис назви фільму

Нагадаємо, що змінну екземпляра *nameLabel* для зберігання демонстратора напису додали до оголошення *ImdbFilmPresenter*. Ініціалізуємо її в методі *initializePresenters* разом з іншими змінними.

```
ImdbFilmPresenter >> initializePresenters
   nameLabel := self newLabel
```

```
            label: 'Name';
            addStyle: 'customLabel';
            yourself.
         nameText := self newTextInput.
         directorText := self newTextInput.
         yearNumber := self newNumberInput
            rangeMinimum: 1900 maximum: Year current year;
            yourself
```

Зрештою, доведеться оновити визначення макета, щоб замість звичайного рядка використати стилізований демонстратор напису.

```
ImdbFilmPresenter >> defaultLayout
 ^ SpGridLayout build: [ :builder |
      Builder
         beColumnNotHomogeneous;
         column:2 withConstraints: #beExpand;
         add: nameLabel; add: nameText; nextRow;
         add: 'Director'; add: directorText; nextRow;
         add: 'Year'; add: yearNumber ]
```

Тепер видно, що напис *Name* демонстратора деталей фільму було стилізовано (див. рис. 3.12).

3.20. Підсумки розділу

Ми побачили, що у Spec розробник визначає, з яких візуальних елементів складається демонстратор. Клас демонстратора відповідає за опис взаємодії між вкладеними демонстраторами та з об'єктами моделі даних. Він також визначає тонкощі свого візуального відображення.

Від перекладача. Розділ містить певне повторення і значну частину нових понять. Як і в попередньому розділі, клас демонстратора наслідували від *SpPresenter* і визначили в ньому методи *initializePresenters*, *connectPresenters*, *defaultLayout*, щоб описати його будову та функціонування; використали методи *initializeWindow:*, *open*, щоб уточнити параметри вікна застосунку і відкрити його. Пригадаймо нові можливості. Функціонування демонстраторів доповнено методами *updatePresenter* (автоматично викликається після *initializePresenters* для отримання даних з моделі) та *setModel:* (для оновлення вмісту демонстратора відповідно до зміни в моделі даних). Описано відкривання демонстратора в модальному діалозі за допомогою *initializeDialogWindow:* та *openModal*. Продемонстровано можливість динамічного використання макетів. Взаємодію демонстраторів організовано за допомогою перенесення даних. Для керування всіма вікнами використано об'єкт застосунку. За допомогою застосунку налаштовано і використано таблицю стилів для відображення візуальних компонентів.

Частина II
Основи Spec

Розділ 4
Кількома словами про ядро Spec

Spec – це програмний каркас у Pharo для побудови графічних інтерфейсів користувача. Він надає будівельні блоки для інтерфейсів від простих вікон до складних інструментів, як браузери та налагоджувачі. За допомогою Spec розробники можуть втілити макет інтерфейсу користувача і взаємодію між візуальними елементами, з яких він складається. Наприклад, розробник може зазначити, що інструмент містить два компоненти: список, розташований ліворуч, і компонент, що відображає детальну інформацію – праворуч. Клацання на елементі списку відобразить детальну інформацію про вибраний елемент. Крім того, Spec підтримує повторне використання компонентів інтерфейсу користувача.

Spec слугує основою більшості інструментів Pharo, таких як Інспектор об'єктів, Навідник, Налагоджувач, Менеджер пакетів тощо. У цьому короткому розділі опишемо контекст функціонування ключових архітектурних елементів Spec.

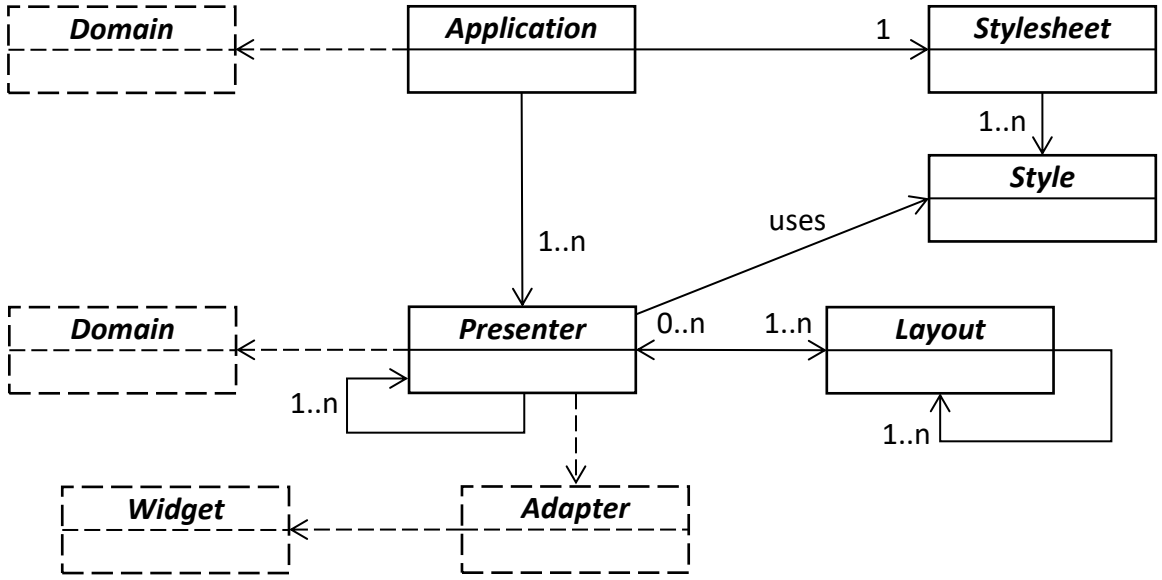


Рис. 4.1. Архітектура Spec

4.1. Огляд архітектури Spec

На рис. 4.1 зображена загальна архітектура Spec. Основа Spec складається з п'яти понять, про які йтиметься в наступних розділах. Найважливішими з них є *демонстратор* (presenter), *макет* (layout) і *застосунок* (application).

Демонстратор описує логіку елементів інтерфейсу користувача, забезпечує зв'язок з моделлю даних. *Застосунок* також перебуває у контакті з об'єктами домену, але загалом у його віданні перебувають специфічні для програми ресурси (піктограми, вікна тощо).

На основі демонстраторів і макетів Spec будує фактичний інтерфейс користувача. Всередині він використовує адаптери, особливі для кожного візуального компонента

та графічної бібліотеки. Тому демонстратори не залежать від графічних бібліотек і їх можна повторно використовувати в різних графічних середовищах.

4.2. Огляд архітектури ядра Spec

Ядро Spec охоплює такі елементи:

- **Застосунок.** Застосунок складається з кількох демонстраторів і таблиці стилів.
- **Демонстратори.** Демонстратор – модуль інтерактивної поведінки. Його поєднано з об'єктами домену та іншими демонстраторами. Візуальне компонування демонстратора визначає принаймні один макет.
- **Макети.** Макет описує розташування елементів і може бути рекурсивним.
- **Таблиця стилів і стилі.** Таблиця стилів містить стилі, які описують візуальні властивості: шрифти, кольори тощо. Стилі застосовують до візуальних компонентів демонстратора.

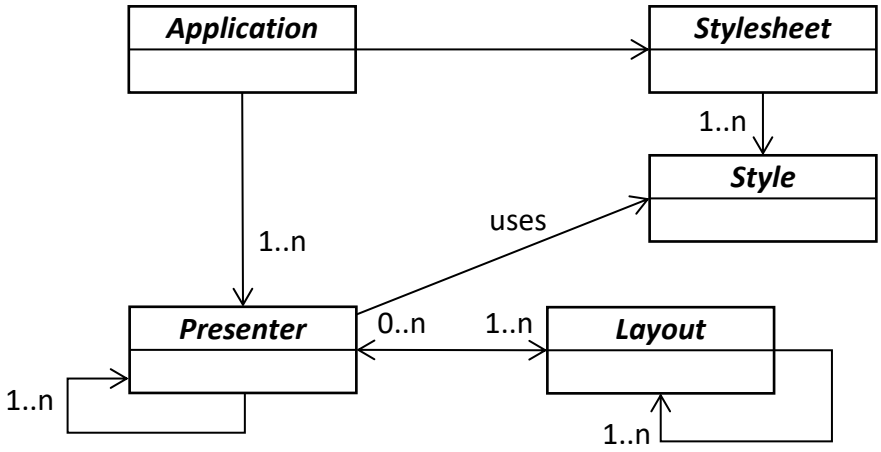


Рис. 4.2. Демонстратор, застосунок, макет і стиль Spec

Розглянемо детальніше кожен з головних елементів.

4.3. Демонстратори

Демонстратор (екземпляр підкласу *SpPresenter*) є надважливою частиною каркасу Spec. Він втілює логіку елемента інтерфейсу користувача. Він може визначати поведінку простого візуального компонента інтерфейсу, такого як кнопка, а може й складного компонента, складеного з багатьох інших демонстраторів (простих чи складних). Щоб створити свій інтерфейс користувача, оголошують демонстратори.

Spec постачається з попередньо визначеним набором основних демонстраторів (візуальних компонентів), готових до використання у ваших демонстраторах. Їх можна знайти в протоколі «*scripting – widgets*» класу *SpPresenter*. Тут є кнопки, написи, перемикачі, поля введення тексту, спадні списки, списки, меню, таблиці, дерева, панелі інструментів, панелі дій, а також складніші компоненти, такі як демонстратори відмінностей коду та блокноти. Можна легко створити новий демонстратор і відобразити його:

```
SpButtonPresenter new
   label: 'ok';
   open
```

Демонстратор також може містити модель, яка є об'єктом предметної області. З нею потрібно взаємодіяти, щоб відобразити або оновити дані. У цьому випадку клас демонстратора наслідують від *SpPresenterWithModel*, щоб екземпляр зберігав посилання на об'єкт домену й оновлював його, коли змінюється модель (див. розділ 6).

Демонстратор визначає макети. Один – обов'язковий. Щоб відобразити демонстратор зі стандартним макетом, використовують метод *open* або *openDialog*. Перший з них відкриє нове вікно, вміст якого становить демонстратор, а другий – модальне діалогове вікно. Можна також використовувати *openWithLayout:* або *openDialogWithLayout:*, щоб відкрити демонстратор із макетом, який передають як аргумент.

4.4. Застосунок

Застосунок Spec (екземпляр ієрархії класів *SpApplication*) відповідає за ініціалізацію, конфігурування та ресурси вашої програми. *SpApplication* не є демонстратором, бо не має графічного зображення. Екземпляр *SpApplication* визначає всю програму (зберігає посилання на графічну бібліотеку, тему, піктограми й інші графічні ресурси), зберігає відкриті вікна, які належать програмі, але сам не відображається.

Застосунок підтримує методи доступу до вікон або ресурсів, таких як піктограми, надає абстракції для взаємодії з користувачем (інформування, помилка, вибір файлу або каталогу).

Нарешті, застосунок підтримує таблицю стилів, яку Spec використовує для стилізації елементів інтерфейсу користувача. Стандартний стиль завжди доступний, але його можна видозмінити під власні потреби, як описано в розділі 15.

Також потрібно оголосити метод, щоб визначити, яке головне вікно чи демонстратор використовуватиметься для запуску програми. Нижче спеціалізовано метод *start*:

```
MyApplication >> start
   (MyMainPresenter newApplication: self) open
```

Тепер можна запустити свою програму за допомогою *MyApplication new run*. Такий код автоматично викличе оголошений метод *start*.

4.5. Налаштування застосунку

Під час ініціалізації застосунку можна зазначити, яку графічну бібліотеку використовувати: Morphic (як усталено) або GTK. У майбутньому Spec також підтримуватиме Toplo, нову бібліотеку компонентів, створену на основі Bloc. Вона замінить Morphic.

Використання Morphic

Продемонструємо налаштування на прикладі програми «Mini IMDB» з розділу 3. Конфігурацію визначено як підклас *SpMorphicConfiguration*.

```
SpMorphicConfiguration << #ImdbMorphicConfiguration
   package: 'CodeOfSpec20Book'
```

У класі конфігурації визначено метод *configure:*

```
ImdbMorphicConfiguration >> configure: anApplication
   super configure: anApplication.
```

```
        "There are ways to write/read this from strings or files,
         but this is how you do it programatically."
        self styleSheet
           addClass: 'header' with: [ :style |
              style
                 addPropertyFontWith: [ :font | font bold: true ];
                 addPropertyDrawWith: [ :draw | draw color: Color red ] ]
```

Зауважимо, що стиль можна задати рядком, як описано в розділі 15.

Нарешті, у класі застосунку за допомогою повідомлення *useBackend:with:* оголошено, що потрібно використати графічну бібліотеку *Morphic* з відповідною конфігурацією.

```
ImdbApp >> initialize
   super initialize.
   self useBackend: #Morphic with: ImdbMorphicConfiguration new
```

Використання теми і налаштувань GTK

Для GTK виконують схожі кроки. Визначають підклас *SpGTKConfiguration*.

```
SpGTKConfiguration << #ImdbGTKConfiguration
   package: 'CodeOfSpec20Book'
```

Потім налаштовують його, вибираючи та розширюючи CSS.

```
ImdbGTKConfiguration >> configure: anApplication
   super configure: anApplication.
   "This will choose the theme 'Sierra-dark' if it is available"
   self installTheme: 'Sierra-dark'.
   "This will add a 'provider' (a stylesheet)"
   self addCSSProviderFromString:
      '.header {color: red; font-weight: bold}'
```

А в ініціалізації застосунку оголошують, що конфігурація повинна використовуватися для GTK.

```
ImdbApp >> initialize
   super initialize.
   self useBackend: #GTK with: ImdbGTKConfiguration new
```

4.6. Макети

Для показу своїх елементів демонстратор використовує макет. Макет описує, як розташувати елементи на екрані. У Spec доступно кілька макетів, щоб з їхньою допомогою створювати гарні інтерфейси користувача.

- **GridLayout**. Макет *SpGridLayout* обирають, коли потрібно створити демонстратор із написами та полями введення, вирівняними у формі прямокутної сітки (стиль форми). Можна задати, в якій клітинці сітки потрібно розмістити візуальний елемент.
- **BoxLayout**. Макет *SpBoxLayout* розташовує демонстратори у прямокутнику послідовно у стовпець (зверху вниз), або в рядок (зліва направо).

- **PanedLayout**. *SpPanedLayout* – це макет із двома елементами, які називають «панелями», і роздільником між ними. Користувач може перетягувати роздільник, щоб змінити розмір панелей.
- **TabLayout**. *SpTabLayout* – «багатошаровий» макет, який демонструє всі свої елементи як вкладки. Користувач вибирає вкладку, щоб побачити її вміст.
- **MillerLayout**. Макет для реалізації стовпців Міллера, також відомих як каскадні списки (https://en.wikipedia.org/wiki/Miller_columns).

Будь-який макет у Spec динамічний і компонований, тобто його можна вибирати в момент відкривання демонстратора, його можна використовувати у складі інших демонстраторів. Загалом макет визначають методом екземпляра демонстратора, але його можна визначити і методом класу.

Визначити макет так само просто, як оголосити метод *defaultLayout*. Цей метод автоматично викликається, якщо макет не встановлено вручну.

Повернімося до методу *defaultLayout* з розділу 2.

```
CustomerSatisfactionPresenter >> defaultLayout
   ^ SpBoxLayout newTopToBottom
      add: (SpBoxLayout newLeftToRight
               add: buttonHappy;
               add: buttonNeutral;
               add: buttonBad;
               yourself);
      add: result;
      yourself
```

Цей метод визначає два макети *BoxLayout*:

- один, що містить три кнопки;
- інший, що містить перший і текст результату під ним.

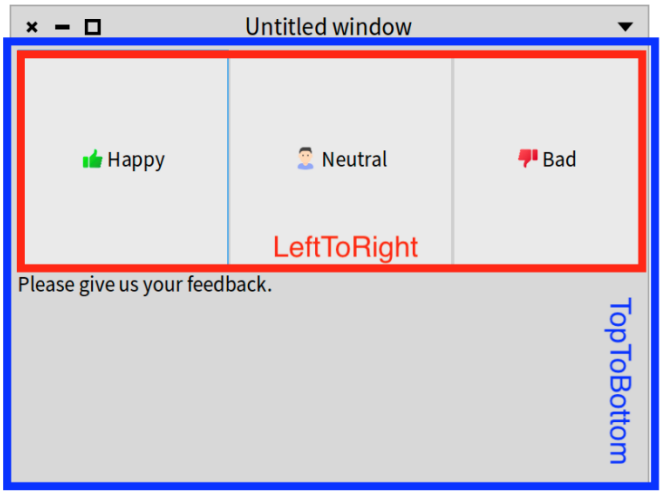


Рис. 4.3. Макет, що відповідає методу *defaultLayout*

Кожен із макетів демонстратора містить посилання на вкладені демонстратори (*buttonHappy*, *buttonNeutral*, *buttonBad*, *result*). На рис.4.3 зображено відповідний результат.

4.7. Стилі та таблиці стилів

Застосунок Spec завжди постачається з таблицею усталених стилів. Вона містить визначення стилів, які можна застосовувати до демонстраторів. Стилі детально описані в розділі 15.

Стиль – це контейнер властивостей для стилізації компонентів і визначає (певною мірою) їхню поведінку в різних макетах.

Ось приклад таблиці стилів для графічної бібліотеки Morphic:

```
'.application [
   .lightGreen [ Draw { #color: #B3E6B5 } ],
   .lightBlue [ Draw { #color: #lightBlue } ] ]'
```

Стилі у Spec схожі на CSS, але виражені в STON. Пам'ятаймо, що не можна забувати крапки на початку кожного оголошення.

Щоб застосувати стиль у застосунку Spec, застосункові надсилають повідомлення *styleSheet:*

```
myStyleSheet := SpStyleVariableSTONReader fromString:
   '.application [
      Font { #bold: true },
      .bgBlack [ Draw { #backgroundColor: #black } ],
      .blue [ Draw { #color: #blue } ]
   ]'
application styleSheet: SpStyle defaultStyleSheet, myStyleSheet.
```

Тоді можна стилізувати демонстратора за допомогою повідомлення *addStyle:* (подібно до тегу з класом у CSS) так:

```
presenter label: 'I am a label'.
presenter addStyle: 'blue'.
```

4.8. Поведінка демонстратора

Як тільки компоненти графічного інтерфейсу користувача визначені (тобто визначені демонстратори і макети Spec), потрібно задати поведінку інтерфейсу користувача: що трапиться, коли відкриють новий демонстратор?

Ймовірно, виникне потреба надати деякі дані (модель) демонстратору, щоб їх можна було відобразити у вікні. Це називають перенесенням: дані переносять від одного демонстратора до іншого. Перенесення визначають як реакції на події.

Досить легко визначити поведінку інтерфейсу користувача за допомогою попередньо оголошених подій візуальних компонентів. Їх можна знайти у протоколі «*api-events*» класів демонстраторів. Найчастіше використовують події *whenSelectionChangedDo:*, *whenModelChangedDo:*, *whenTextChangedDo:*. Ось кілька прикладів:

```
messageList
   whenSelectionChangedDo: [ :selection |
      messageDetail model: selection selectedItem ];
   whenModelChangedDo: [ self updateTitle ].
textModel whenSubmitDo: [ :text | self accept: text ].
```

```
    addButton action: [ self addDirectory ].
    filterInput whenTextChangedDo: [ :text | self refreshTable ].
```

4.9. Підсумки розділу

Клас *SpPresenter* – центральний клас, який має такі обов'язки:

- Ініціалізація складових демонстратора та його стану.
- Визначення макета застосунку.
- Поєднання елементів для підтримки потоку взаємодії.
- Оновлення компонентів інтерфейсу користувача.

Усі ці пункти будуть проілюстровані в наступних розділах.

Розділ 5
Тестування застосунків Spec

Розробники часто думають, що тестувати графічний інтерфейс користувача складно. Це правда, що вичерпна перевірка розташування і компонування візуальних компонентів може бути виснажливою. Однак тестування логіки програми, зокрема логіки взаємодії, цілком можливе. Це те, чому присвячений цей розділ. Ми з'ясуємо, що тестування застосунку Spec просте й ефективне.

5.1. Тестування демонстраторів

Тести є головним інструментом для перевірки того, що все працює правильно. Завдяки ним можна не боятися пошкодити код і не помітити цього. Тести підтримують процес поліпшення та доповнення програмного коду. Це загальновідомі твердження, які стосуються багатьох галузей розробки програм, так само істинні й щодо створення інтерфейсів користувача.

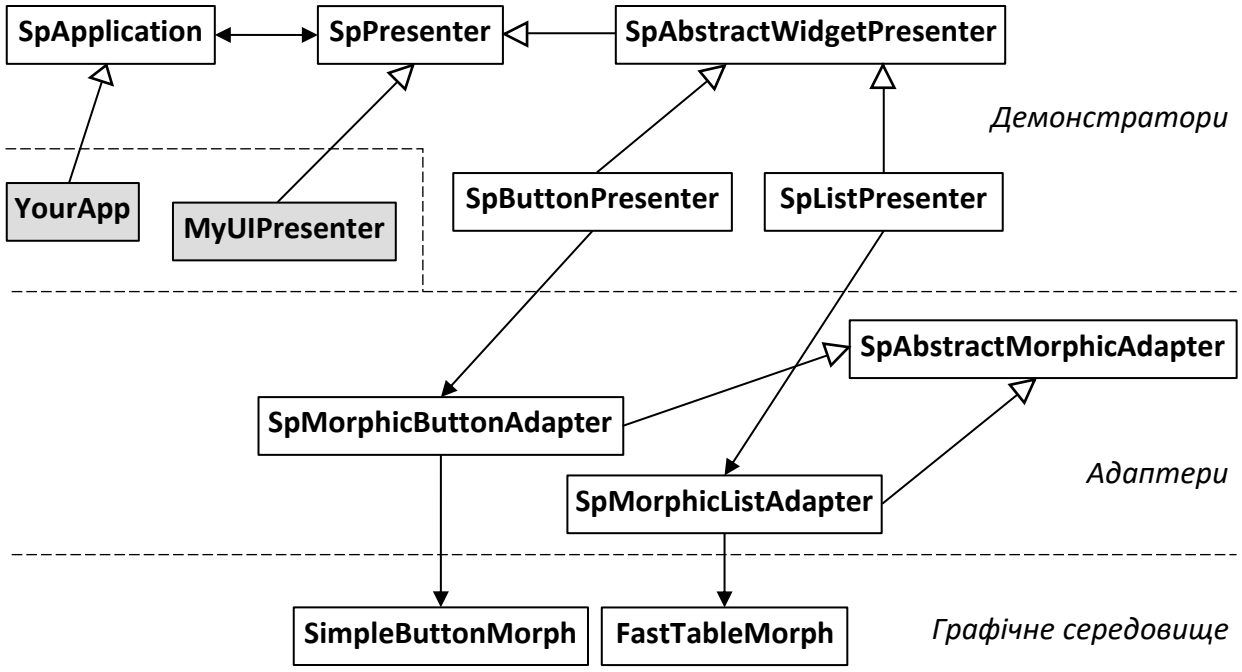


Рис. 5.1. Архітектура Spec складається з трьох шарів: *Демонстратори – Адаптери – Графічні бібліотеки*

Архітектура Spec

Spec засновано на архітектурі з трьома різними шарами, як зображено на рис. 5.1:

- **Демонстратори** визначають логіку взаємодії та керують об'єктами предметної області, доступаються до низькорівневих графічних компонентів, але через програмний інтерфейс, визначений адаптерами.
- **Адаптери** – це об'єкти, які надають доступ до низькорівневих графічних компонентів, слугують мостом між демонстраторами та компонентами.

- **Низькорівневі компоненти** – це звичайні графічні компоненти, які можна використовувати без Spec.

Три ролі розробника

Щоб допомогти зрозуміти різні можливості тестування, якими можна займатися, виокремимо такі ролі та пов'язані з ними клопоти.

- **Користувачі Spec** – це розробники, які будують новий застосунок. Вони визначають логіку програми, складають демонстратори й об'єкти предметної області. Ми віримо, що це та роль, яку ви будете виконувати більшу частину часу.
- **Розробники Spec** більше стурбовані розробкою нових демонстраторів Spec і пов'язуванням їх з адаптерами.
- **Розробники графічних компонентів** стурбовані логікою та роботою певного компонента для заданого графічного середовища.

Бачення користувача Spec

Зосередимося на висвітленні першої ролі. Для читача, зацікавленого у другій, хорошим місцем для початку є клас *SpAbstractBackendForTest*.

Користувач Spec може покладатися на те, що графічне середовище працює належно, його завдання полягає в перевірці логіки компонентів інтерфейсу користувача. Потрібно переконатися, що кожного разу, коли змінюється модель, компоненти інтерфейсу відображають ці зміни. І навпаки, коли змінюються компоненти інтерфейсу користувача, потрібно переконатися, що модель оновлено. Наведемо приклад.

Рис. 5.2. Застосунок Spec

5.2. Приклад для користувача Spec

Тестуватимемо простий застосунок Spec, зображений на рис. 5.2. Моделлю для нього є екземпляр класу *Color*. Застосунок показує список кольорів, з яких користувач може вибрати один. Застосунок показує вибраний колір у великому прямокутнику та виводить у текстовій панелі його *printString* і шістнадцятковий код. Застосунок має також дві кнопки, щоб зробити вибраний колір світлішим або темнішим.

Демонстратор визначається, як описано нижче. Клас має шість змінних екземпляра. Перші п'ять з них містять піддемонстратори, з яких складається вікно програми. Шоста змінна екземпляра містить колір – модель даних програми.

```
SpPresenter << #ColorChooser
   slots: { #colorList . #colorDetails . #colorBox . #lighterButton .
      #darkerButton . #currentColor };
   package: 'CodeOfSpec20Book'
```

Метод *initializePresenters* створює вкладені демонстратори: *colorList* містить демонстратор списку з кольорами, *colorBox* відображає вибраний колір у *SpRoassalPresenter*, *colorDetails* містить текстовий демонстратор, який показує інформацію про колір, *lighterButton* і *darkerButton* – це кнопки, які роблять поточний колір світлішим або темнішим.

```
ColorChooser >> initializePresenters
   colorList := self newList
      display: [ :color | '' ];
      displayBackgroundColor: [ :color | color ];
      yourself.
   colorBox := self instantiate: SpRoassalPresenter.
   lighterButton := self newButton
      label: 'Lighter';
      action: [ self lighter ];
      yourself.
   darkerButton := self newButton
      label: 'Darker';
      action: [ self darker ];
      yourself.
   colorDetails := self newText
```

Початкове значення змінної *currentColor* задано не в методі *initializePresenters*, а в *setModelBeforeInitialization:*, бо колір можна задати під час створення нового екземпляра *ColorChooser*.

```
ColorChooser >> setModelBeforeInitialization: aColor
   currentColor := aColor
```

Метод *defaultLayout* визначає макет з двох частин: лівої та правої. Ліворуч розташовано список кольорів. Права частина містить панель для кольору, дві кнопки і текстове поле для деталей кольору. Композиція горизонтальних і вертикальних макетів разом з 5-піксельними відступами між елементами створює вікно, зображене на рис. 5.2.

```
ColorChooser >> defaultLayout
   | colorBoxAndDetails buttons |
   buttons := SpBoxLayout newLeftToRight
      spacing: 5;
      add: lighterButton;
      add: darkerButton;
      yourself.
   colorBoxAndDetails := SpBoxLayout newTopToBottom
      spacing: 5;
      add: colorBox;
      add: buttons expand: false;
      add: colorDetails;
      yourself.
   ^ SpBoxLayout newLeftToRight
      spacing: 5;
      add: colorList expand: false;
      add: colorBoxAndDetails;      yourself
```

Метод *initializeWindow:* задає заголовок і початкові розміри вікна.

```
ColorChooser >> initializeWindow: aWindowPresenter
   aWindowPresenter
      title: 'Color Chooser';
      initialExtent: 400@294
```

Легко описати взаємодію вкладених демонстраторів. Коли користувач вибере один зі списку кольорів, потрібно оновити інформацію про нього.

```
ColorChooser >> connectPresenters
   colorList whenSelectionChangedDo: [ :selection |
      self updateColor: selection selectedItem ]
```

Дії щодо оновлення поля кольору та деталей кольору метод *connectPresenters* делегує методові *updateColor:*. Видно, що *updateColor:* піклується про можливе значення *nil* змінної *currentColor*.

```
ColorChooser >> updateColor: color
   | details |
   currentColor := color.
   colorBox canvas
      background: (currentColor ifNil: [ Color transparent ]);
      signalUpdate.
   details := currentColor
      ifNil: [ '' ]
      ifNotNil: [ self detailsFor: currentColor ].
   colorDetails text: details
```

Відповідальність за створення тексту з деталями кольору метод *updateColor:* делегує методові *detailsFor:*.

```
ColorChooser >> detailsFor: color
   ^ String streamContents: [ :stream |
      stream
         print: color; cr; cr; nextPut: $#;
         nextPutAll: color asHexString ]
```

Щоб налаштувати початковий стан демонстраторів, визначимо метод *updatePresenter*. Він заповнює список кольорів стандартними, визначеними в *defaultColors*, і задає початковий колір за допомогою *updateColor:*.

```
ColorChooser >> updatePresenter
   | initialColor |
   initialColor := currentColor.
   colorList items: self defaultColors.
   self updateColor: initialColor
```

Зауважимо, що збереження *initialColor := currentColor* початкового кольору необхідне, бо *colorList items: self defaultColors* скидає вибір у списку, що спричиняє виконання блока з *connectPresenters*. Цей блок надсилає *updateColor: nil*, оскільки в списку ще нічого не вибрано. Тому метод так і влаштовано: він зберігає початковий колір і застосовує його за допомогою *self updateColor: initialColor*.

Для простоти метод *defaultColors* повертає колекцію небагатьох уживаних кольорів. Його можна легко змінити так, щоб він повертав іншу колекцію кольорів. Наприклад, можна спробувати додати в неї *Color red wheel: 20*.

```
ColorChooser >> defaultColors
   ^ {
      Color red .
      Color orange .
      Color yellow .
      Color green .
      Color magenta .
      Color cyan .
      Color blue .
      Color purple .
      Color pink .
      Color brown .
      Color white .
      Color gray .
      Color black }
```

У наведеному вище коді бракує лише двох методів для завершення реалізації класу. Метод *initializePresenters* встановлює дії для кнопок, які викликають такі два методи. Вони делегують виконання важкої роботи методу *updateColor:*.[4]

```
ColorChooser >> lighter
   self updateColor: currentColor lighter

ColorChooser >> darker
   self updateColor: currentColor darker
```

Після додавання наведеного вище коду можна запустити застосунок. Почнемо зі стандартного відкривання за допомогою *ColorChooser new open*.

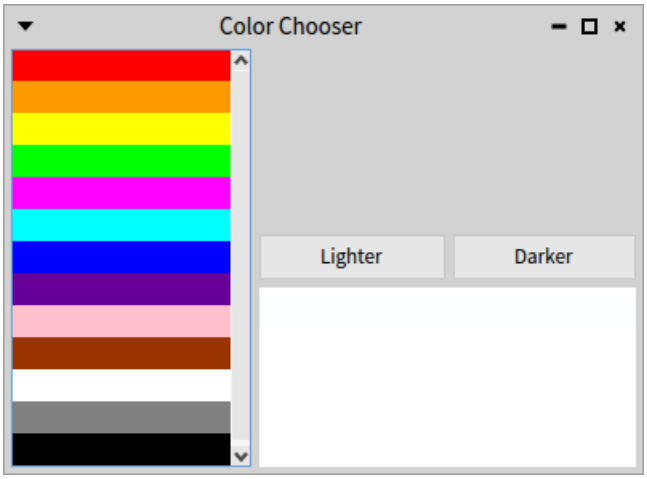


Рис. 5.3. Стандартний вигляд застосунку *ColorChooser*

У цьому випадку початковий колір не задано, тому вікно виглядає, як на рис. 5.3. Панель кольору не відображає нічого, а тексту з деталями кольору немає.

[4] Не тільки методові *updateColor:*, а й методам *lighter, darker* класу *Color* (прим. – *Ярошко С.*).

Рис. 5.4. Застосунок *ColorChooser* відкрито з початковим жовтим кольором

Подивімося, що трапиться, якщо задати колір перед відкриванням.

```
(ColorChooser on: Color yellow) open
```

У цьому випадку початковим кольором задано жовтий, він має відобразитися, коли відкриється вікно. Зверніть увагу, що метод *on:* не було визначено в класі *ColorChooser*. Його успадковано від надкласу *SpAbstractPresenter*. Результат зображено на рис. 5.4.

5.3. Тести

Увесь код застосунку зібрали, настав час написати кілька тестів. Розпочнемо з визначення класу тестів.

```
TestCase << #ColorChooserTest
   slots: { #chooser };
   package: 'CodeOfSpec20Book'
```

Кожен тест відкриватиме новий екземпляр застосунку *ColorChooser*. Його зберігатиме змінна екземпляра *chooser*. Щоб забезпечити очищення пам'яті від тестованого застосунку, визначимо метод *tearDown*. У ньому враховано, що тест може перерватися до того, як *chooser* буде прив'язано до екземпляра *ColorChooser*.

```
ColorChooserTest >> tearDown
   chooser ifNotNil: [ chooser delete ].
   super tearDown
```

Підготувавши таку інфраструктуру, можна писати тести.

Стандартне відкривання

Перший тест описує стан застосунку після стандартного відкривання.

```
ColorChooserTest >> testDefault
   "Коли ColorChooser відкривають без заданого кольору,
    панель кольору прозора, а деталі кольору відсутні."

   chooser := ColorChooser new.
   chooser open.
```

```
        self assert: chooser boxColor equals: Color transparent.
        self assert: chooser detailsText equals: ''
```

Щоб це запрацювало, потрібно додати кілька так званих методів підтримки тестів. Вони належать до протоколу «*testing support*» класу *ColorChooser*, бо призначені лише для використання в тестах.

```
ColorChooser >> boxColor
   ^ colorBox canvas color

ColorChooser >> detailsText
   ^ colorDetails text
```

Правильна ініціалізація

Другий тест описує стан застосунку після відкриття з заданим кольором.

```
ColorChooserTest >> testInitialization
   "Коли ColorChooser відкривають на певному кольорі,
    панель кольору відображає цей колір, а текстове вікно
    відображає його printString і шістнадцятковий код."

   chooser := ColorChooser on: Color palePeach.
   chooser open.

   self assert: chooser boxColor equals: Color palePeach.
   self assert: chooser detailsText
      equals: 'Color palePeach\\#FFEDD5' withCRs
```

Вибір кольору

Третій тест описує, що відбувається, коли користувач вибирає колір.

Спочатку тест вибирає перший колір у списку та перевіряє стан вкладених демонстраторів. Потім він вибирає сьомий колір у списку та перевіряє очікувані зміни їхнього стану.

```
ColorChooserTest >> testChooseColor
   "Коли користувач вибирає колір зі списку,
    панель кольору відображає цей колір, а текстове вікно
    відображає його printString і шістнадцятковий код."

   chooser := ColorChooser new.
   chooser open.

   chooser clickColorAtIndex: 1.
   self assert: chooser boxColor equals: Color red.
   self assert: chooser detailsText equals: 'Color red\\#FF0000' withCRs.

   chooser clickColorAtIndex: 7.
   self assert: chooser boxColor equals: Color blue.
   self assert: chooser detailsText equals: 'Color blue\\#0000FF' withCRs
```

У цьому тесті використано додатковий «метод підтримки тесту», що імітує клацання на елементі списку.

```
ColorChooser >> clickColorAtIndex: index
   colorList clickAtIndex: index
```

Перетворення поточного кольору на світліший

Настав час описати поведінку програми після натискання кнопки ***Lighter***.

Тест складається з чотирьох частин. Спершу імітуємо клацання на першому у списку кольорі. Це призводить до оновлення панелі кольору та текстової інформації. Після «натискання» кнопки ***Lighter*** тест перевіряє змінений стан панелі та тексту. Потім він «натискає» її вдруге, щоб описати, що поточний колір можна робити світлішим знову і знову. Нарешті, тест вибирає сьомий колір у списку та перевіряє очікувані зміни стану піддемонстраторів.

```
ColorChooserTest >> testLighter
   "Коли користувач натискає кнопку 'Lighter',
    панель кольору відображає світліший колір,
    змінюються також текстове зображення і шістнадцятковий код."

   chooser := ColorChooser new.
   chooser open.

   chooser clickColorAtIndex: 1.
   chooser clickLighterButton.
   self
      assert: chooser boxColor
      equals: (Color r: 1.0 g: 0.030303030303030304 b:
         0.030303030303030304 alpha: 1.0).
   self
      assert: chooser detailsText
      equals: '(Color r: 1.0 g: 0.030303030303030304 b:
0.030303030303030304 alpha: 1.0)\\#FF0707' withCRs.

   chooser clickLighterButton.
   self
      assert: chooser boxColor
      equals: (Color r: 1.0 g: 0.06060606060606061 b:
         0.06060606060606061 alpha: 1.0).
   self
      assert: chooser detailsText
      equals: '(Color r: 1.0 g: 0.06060606060606061 b:
0.06060606060606061 alpha: 1.0)\\#FF0F0F' withCRs.

   chooser clickColorAtIndex: 7.
   chooser clickLighterButton.
   self
      assert: chooser boxColor
      equals: (Color r: 0.030303030303030304 g:
         0.030303030303030304 b: 1.0 alpha: 1.0).
   self
      assert: chooser detailsText
      equals: '(Color r: 0.030303030303030304 g: 0.030303030303030304
b: 1.0 alpha: 1.0)\\#0707FF' withCRs
```

Цей тест, як і інші, потребує додаткового методу підтримки.

```
ColorChooser >> clickLighterButton
   lighterButton click
```

Перетворення поточного кольору на темніший

Тест дуже схожий на попередній, тільки замість кнопки ***Lighter*** він натискає ***Darker***.

```
ColorChooserTest >> testDarker
   "Коли користувач натискає кнопку 'Darker',
    панель кольору відображає темніший колір,
    змінюються також текстове зображення і шістнадцятковий код."

   chooser := ColorChooser new.
   chooser open.

   chooser clickColorAtIndex: 1.
   chooser clickDarkerButton.
   self
      assert: chooser boxColor
      equals: (Color r: 0.9198435972629521 g: 0.0 b: 0.0 alpha: 1.0).
   self
      assert: chooser detailsText
      equals: '(Color r: 0.9198435972629521 g: 0.0 b: 0.0 alpha:
1.0)\\#EB0000' withCRs.

   chooser clickDarkerButton.
   self
      assert: chooser boxColor
      equals: (Color r: 0.8396871945259042 g: 0.0 b: 0.0 alpha: 1.0).
   self
      assert: chooser detailsText
      equals: '(Color r: 0.8396871945259042 g: 0.0 b: 0.0 alpha:
1.0)\\#D60000' withCRs.

   chooser clickColorAtIndex: 7.
   chooser clickDarkerButton.
   self
      assert: chooser boxColor
      equals: (Color r: 0.0 g: 0.0 b: 0.9198435972629521 alpha: 1.0).
   self
      assert: chooser detailsText
      equals: '(Color r: 0.0 g: 0.0 b: 0.9198435972629521 alpha:
1.0)\\#0000EB' withCRs
```

Знову ж таки, цей тест потребує додаткового методу підтримки. Зауважимо, що замість нього можна було б оголосити метод-селектор кнопки і доступатися до неї з тесту, але віддамо перевагу допоміжному методу, бо він акцентує на логіці використання.

```
ColorChooser >> clickDarkerButton
   darkerButton click
```

Перевірка властивостей вікна

Тепер варто перевірити, чи правильно побудоване вікно застосунку. Перевіримо його заголовок і початковий розмір.

```
ColorChooserTest >> testInitializeWindow
   | window |
   chooser := ColorChooser new.
   window := chooser open.
   self assert: window isBuilt.
   self assert: window title equals: 'Color Chooser'.
   self assert: window initialExtent equals: 400@294
```

5.4. Тестування застосунку

За запуск програми та відстеження її вікон у Spec відповідає екземпляр підкласу *SpApplication*. Йому надсилають повідомлення *run*, щоб запустити застосунок на виконання, наприклад, так: *ColorChooserApplication new run*. У тілі *run* буде викликано метод *start* застосунку. Метод *start* – це метод-зачіпка, який потрібно перевантажити, щоб програма стартувала так, як хоче програміст.

Важливо пам'ятати, що в методі *start* потрібно налаштувати демонстратор перед відкриттям так, щоб він знав свій застосунок. Це важливо, бо тоді застосунок знатиме вікна, які відкриває.

Працюватимемо в стилі TDD[5] і визначимо спочатку клас тестів і його методи.

```
TestCase << #ColorChooserApplicationTest
   slots: { #application };
   package: 'CodeOfSpec20Book'

ColorChooserApplicationTest >> setUp
   super setUp.
   application := ColorChooserApplication new

ColorChooserApplicationTest >> tearDown
   application ifNotNil: [ application closeAllWindows ].
   super tearDown

ColorChooserApplicationTest >> testWindowRegistration
   self assert: application windows size equals: 0.
   application start.
   self assert: application windows size equals: 1.
   application start.
   self assert: application windows size equals: 2
```

Від перекладача. У тілі методу *setUp* використано невідомий поки що клас *ColorChooserApplication*, тому оглядач класів запропонує вирішити, що означатиме це глобальне ім'я: чи глобальну змінну, чи поле класу, чи новий клас, чи ще щось. Ви можете одразу оголосити клас, як у фрагменті нижче, або вибрати *Cancel*, і отже, відкласти оголошення класу застосунку на потім.

Метод *testWindowRegistration* описує очікувану поведінку програми. Якщо відкриті вікна правильно зареєстровані, то застосунок повинен мати доступ до всіх них. Тест послідовно відкриває два вікна та перевіряє, чи збільшується кількість вікон, відомих застосункові.

[5] *TDD* – скорочення від Test Driven Development, розробка, керована тестами (прим. – *Ярошко С.*).

Тест не проходить, бо *ColorChooserApplication* ще не існує. Визначимо його.

```
SpApplication << #ColorChooserApplication
   slots: {};
   package: 'CodeOfSpec20Book'
```

Тест все одно провалюється. Помилку виявляє друга перевірка: після запуску колекція вікон залишається порожньою, бо застосунок не реєструє відкриті вікна. Помилку треба виправити. Реалізуймо метод *start* так, щоб він реєстрував вікна.

```
ColorChooserApplication >> start
   ColorChooser new
      application: self;
      open
```

Та-да! Тест проходить.

5.5. Відомі обмеження та підсумки розділу

У розділі з'ясовано, що ви можете скористатися перевагами Spec для визначення тестів, які допоможуть розвивати візуальну частину програми. Це справді ключ до розробки сучасного програмного забезпечення та для зменшення стресу в майбутньому. Тож отримайте вигоду від гнучкої розробки.

Наразі Spec не пропонує способів створення сценаріїв і керування вікнами, що вирінають. Неможливо створити сценарій кнопки, яка відкриває діалогове вікно для введення значення. Майбутні версії Spec повинні охопити цю функцію, якої ще немає.

Від перекладача. Крім тестування, в розділі можна знайти багато нових цікавих можливостей щодо використання демонстраторів. Зокрема, в методі *initializePresenters* демонстратор списку налаштовано так, щоб його елементи відображали різні кольори; вперше згадано метод *setModelBeforeInitialization:*, який дає змогу оновити дані програми ще перед відкриванням вікна, а для створення екземпляра демонстратора з заданими даними використано метод *on:*; у методах підтримки тестів можна знайти приклади програмної взаємодії з візуальними компонентами, які імітують дії користувача – методи *clickAtIndex:*, *click*; у методі *testInitializeWindow* отримали доступ до вікна застосунку; в класі застосунку визначили метод *start*. Сподіваємося, що читачі зауважили й інші цікавинки.

Розроблений у розділі застосунок має недоліки щодо функціонування кнопок $Lighter$, $Darker$: поки користувач не вибрав колір зі списку, натискання кнопок спричиняють помилку, а після того, як користувач зробив обраний колір світлішим або темнішим (кнопками), демонстратор списку все ще зберігає зроблений перед тим вибір, і повернутися до початкового кольору простим клацанням не вийде, адже список «думає» що потрібний елемент вже виділений і бажана подія не настає. Можна клацнути на елементі списку двічі. А можна видозмінити методи опрацювання подій кнопок. Наприклад, метод *lighter* виглядатиме так.

```
lighter
    | color |
    currentColor ifNil: [ ^ self ].
    color := currentColor lighter.
    colorList unselectAll.
    self updateColor: color
```

Розділ 6

Подвійне призначення демонстраторів: модель даних і взаємодія

Демонстратор відіграє двояку роль у Spec. З одного боку, він діє як сполучна ланка між об'єктами предметної області та візуальними компонентами, а з іншого – він реалізує логіку поведінки інтерфейсу користувача, поєднуючи піддемонстратори між собою. Ці два аспекти становлять основу демонстратора, і це те, про що йдеться в розділі.

Розпочнемо з висвітлення важливого аспекту функціонування демонстраторів: комунікації з об'єктами домену, який надалі називатимемо моделлю даних.

У цьому розділі розглянемо ключові аспекти Spec і відзначимо важливі етапи налаштування в процесі побудови застосунку, що стане в пригоді в майбутньому.

6.1. Про демонстратор з моделлю даних

Часто виникає потреба відкрити демонстратор з певними даними, наприклад, зі списком запланованих справ. У такому випадку було б доречно, щоб піддемонстратори (список, редактор тексту тощо) отримали початкові дані з того об'єкта, який передали демонстраторові. Наприклад, перелік товарів з кошика покупця.

Проте звичайне створення екземпляра демонстратора за допомогою повідомлення *new* і наступне пересилання йому об'єкта даних не працюватиме, як хотілося б, бо такі повідомлення, як *initializePresenters*, на той момент будуть уже опрацьовані.

У Spec є два шляхи вирішення цієї проблеми, і, зокрема, Spec пропонує спеціальний клас демонстратора, що називається *SpPresenterWithModel*. Пояснимо, як ним скористатися.

Побудуємо простий приклад, щоб показати, як це зробити обома способами. Реалізуємо демонстратор, який відображає перелік сигнатур методів класу. Спочатку використаємо демонстратор, успадкований від усталеного надкласу (*SpPresenter*), а потім – демонстратор, призначений для роботи з моделлю (підклас *SpPresenterWithModel*).

6.2. Приклад з *SpPresenter*

Якщо немає потреби реагувати на зміни моделі, то можна просто наслідувати *SpPresenter*, перевантажити метод *setModelBeforeInitialization:*, щоб задати свій об'єкт домену, і створити екземпляр демонстратора *YourPresenter on: yourDomainObject*.

Це саме те, що робитимемо далі.

Спочатку оголосимо новий клас демонстратора.

```
SpPresenter << #MethodLister
   slots: { #sourceClass . #list };
   package: 'CodeOfSpec20Book'
```

Тоді створимо вкладений демонстратор списку і заповнимо його даними.

```
MethodLister >> initializePresenters
   list := self newList.
   list items: sourceClass selectors sorted
```

Перевантажимо метод *setModelBeforeInitialization:* так, щоб зберегти його аргумент, що надходить із повідомлення *on:*, у змінній екземпляра *sourceClass* для використання у майбутньому.

```
MethodLister >> setModelBeforeInitialization: aModel
   sourceClass := aModel
```

Визначимо також базовий макет демонстратора.

```
MethodLister >> defaultLayout
   ^ SpBoxLayout newTopToBottom
        add: #list;
        yourself
```

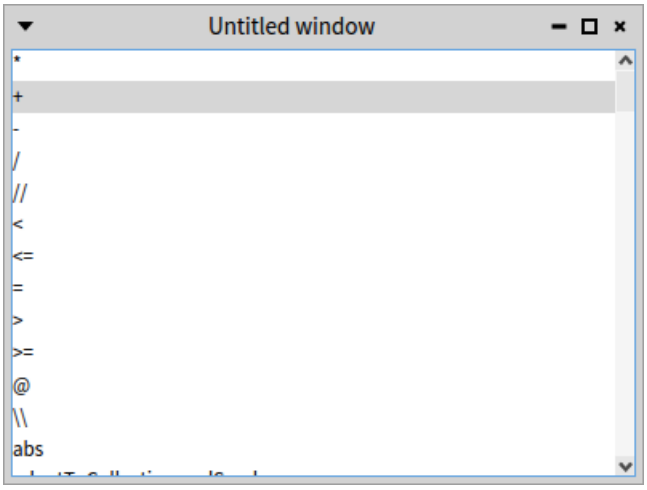


Рис. 6.1. Впорядкований список селекторів методів класу *Point*

Фрагмент коду *(MethodLister on: Point) open* відкриває вікно зі списком методів класу *Point*, як зображено на рис. 6.1.

6.3. *SpPresenter* VS *SpPresenterWithModel*

Головна відмінність між використанням *SpPresenter* і *SpPresenterWithModel* виявляється у здатності реагувати на зміни моделі. Йдеться ось про яку ситуацію: поки відкрите вікно з демонстратором, може трапитися подія, яка змінить об'єкт, використаний для його наповнення. У прикладі з попереднього параграфа заміна переданого в застосунок класу мала б призвести до зміни відображеного переліку методів. Щоб забезпечити таку поведінку, потрібно використати *SpPresenterWithModel*.

Виконання зображеного нижче фрагмента коду засвідчує, що зміна моделі ніяк не впливає на написаний раніше застосунок у тому сенсі, що після зміни класу на *Rectangle* список не оновлюється і продовжує демонструвати перелік методів класу *Point*.

```
| lister |
lister := MethodLister on: Point. lister open.
lister setModel: Rectangle
```

Від перекладача. Тут автори книги забули сказати, що *MethodLister* наслідує порожній метод *setModel:*, тому для чистоти експерименту треба було б оголосити

```
MethodLister >> setModel: aModel
   sourceClass := aModel
```

Звичайно, це оголошення не змінить результату виконання згаданого фрагмента коду.

6.4. Приклад з *SpPresenterWithModel*

Окрім візуальних компонентів, демонстратор може містити модель, об'єкт предметної області, з яким взаємодіють, коли потрібно відобразити або оновити дані. У такому випадку клас демонстратора наслідують від *SpPresenterWithModel*, щоб демонстратор зберігав посилання на об'єкт домену та відстежував його зміни. Щоб змінити модель такого демонстратора, використовують повідомлення *model:*.

Метод *model:* успадкований від надкласу. Він реалізує таку поведінку:

- Якщо об'єкт домену є екземпляром класу *Model*, то він без змін зберігається в демонстраторі.
- Інакше створюється об'єкт-обгортка для зберігання об'єкта домену, щоби надходили сповіщення про всі зміни об'єкта домену, використаного у демонстраторі.

Тепер не потрібно визначати метод *setModelBeforeInitialization:*, як було раніше.

Повернімося до попереднього прикладу. Спершу наслідуємо від *SpPresenterWithModel*.

```
SpPresenterWithModel << #MethodListerWithModel
   slots: { #list };
   package: 'CodeOfSpec20Book'
```

Далі визначимо *initializePresenters*.

```
MethodListerWithModel >> initializePresenters
   list := self newList
```

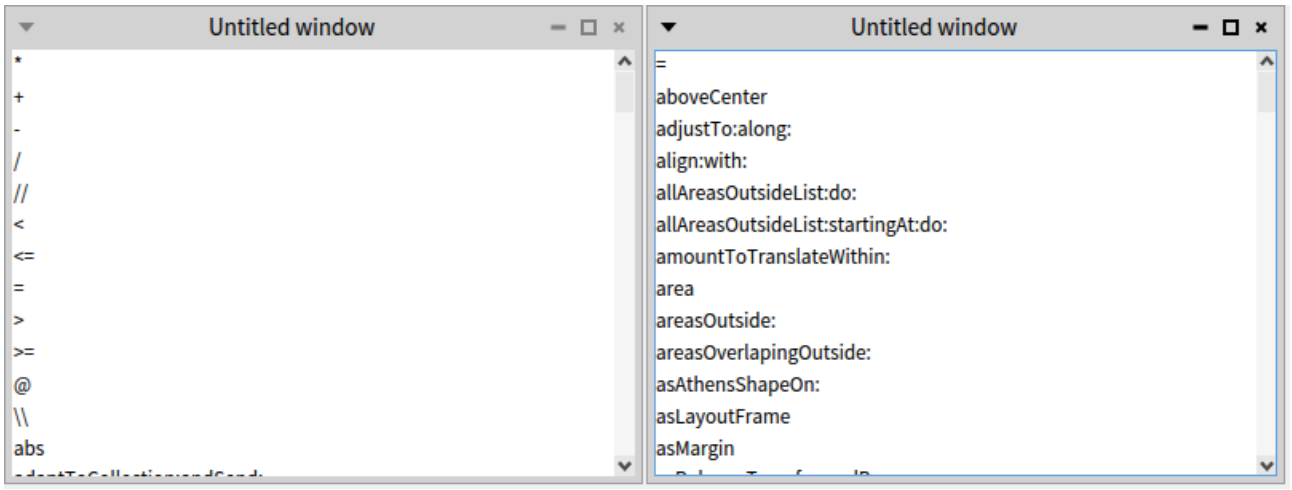


Рис. 6.2. Впорядкований список селекторів змінився відповідно до зміни моделі

Щоб реагувати на зміни моделі та належно оновлювати графічний інтерфейс, визначимо метод *modelChanged*.

```
MethodListerWithModel >> modelChanged
   list items: self model selectors sorted
```

Визначення макета залишимо без змін.

```
MethodListerWithModel >> defaultLayout
   ^ SpBoxLayout newTopToBottom
        add: #list;
        yourself
```

Тепер можна відкрити демонстратор. Виконання зазначеного нижче коду засвідчить, що такий демонстратор належно реагує на зміну моделі (див. рис. 6.2).

```
| lister |
lister := MethodListerWithModel on: Point.
lister open.
lister model: Rectangle
```

Нагадаємо, аби правильно створити демонстратор, використовують метод *newApplication: anApplication*. Це гарантує, що застосунок знатиме всі складові демонстратора.

Тому записаний раніше код мав би бути таким.

```
| lister app |
app := SpApplication new
lister := MethodListerWithModel newApplication: app.
```

Тоді виникає інша проблема, адже потрібно задати модель також. Правильним та ідіоматичним способом є використання методу *newApplication:model:*. Тоді остаточна версія коду буде такою.

```
| lister app |
app := SpApplication new.
lister := MethodListerWithModel newApplication: app model: Point.
lister open.
lister model: Rectangle
```

Ви побачили, що можна легко створити інтерфейс користувача застосунку, заповнений даними моделі та сприйнятливий до змін моделі.

Тепер зосередимося на моделюванні логіки інтерфейсу користувача.

6.5. Побудова інтерфейсу користувача: модель подання

Ключовим аспектом Spec є те, що всі інтерфейси користувача створюють за допомогою повторного використання та композиції наявних інтерфейсів користувача. Щоби таке стало можливим, оголошення інтерфейсу користувача визначає *проєкт* (або *модель*) інтерфейсу, а *не* створює самі візуальні компоненти, які будуть показані на екрані. Ці компоненти створює Spec, опираючись на обране графічне середовище.

Остаточно, саме модель презентації (композиція моделей) та візуальні компоненти становлять кінцевий інтерфейс користувача. Ця композиція моделей подана як об'єкт *SpPresenter*. Визначений у Spec демонстратор відповідає Демонстратору в тріаді MVP, як зображено на рис. 6.3.

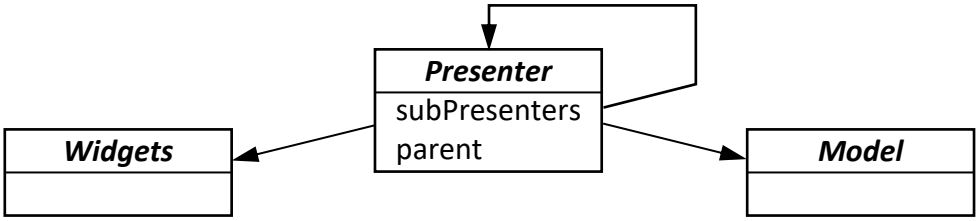


Рис. 6.3. Демонстратор – це модель презентації.
Він пов'язаний з візуальними компонентами та своєю моделлю домену, об'єднує вкладені демонстратори так, що формується дерево демонстраторів

Щоб визначити новий інтерфейс користувача, розробник повинен оголосити підклас *SpPresenter*.

Головно він збудований навколо вирішення трьох завдань, які втілюються у вигляді таких його методів:

- *initializePresenters* зосереджений на створенні вкладених демонстраторів;
- *connectPresenters* турбується про взаємодію піддемонстраторів;
- *defaultLayout* визначає макет розташування піддемонстраторів.

Як наслідок, такі методи зазвичай містить кожна модель інтерфейсу користувача. Щоб отримати уявлення про методи, про які йтиметься далі, можна прочитати код маленького інтерфейсу, описаного в розділі 2.

Тут ми описуємо тонкощі кожного методу та те, як ці три методи працюють разом для створення загального інтерфейсу користувача.

6.6. Метод *initializePresenters*

Метод *initializePresenters* створює, зберігає у змінних екземпляра та частково налаштовує різні візуальні компоненти, які будуть частиною інтерфейсу користувача.

Втілення проєкту побудови демонстратора спричиняє створення екземплярів та ініціалізацію різних низькорівневих візуальних компонентів, які всі разом утворюють видимий графічний інтерфейс користувача. Перша частина конфігурації кожного компонента визначена в *initializePresenters*.

Завдання цього методу – визначити те, як виглядатимуть компоненти та якою є їхня автономна поведінка. Тут може бути описана реакція на натискання кнопок, наприклад, як у застосунку *ColorChooser*. До відповідальності методу *не* належить визначення взаємодії *між* компонентами.

У загальному випадку метод *initializePresenters* повинен відповідати шаблону:

- створення екземпляра візуального компонента;
- налаштування параметрів компонента;
- задання послідовності переходів фокуса введення.

Останній крок не обов'язковий, бо порядок переходів фокуса стандартно визначається порядком оголошення вкладених демонстраторів.

Застереження. Оголошувати метод *initializePresenters* потрібно обов'язково, бо без нього інтерфейс користувача не міститиме візуальних компонентів.

Створення вкладених демонстраторів

Створити екземпляр підпрезентатора (тобто моделі візуального компонента, що входить до складу інтерфейсу користувача) можна двома способами: за допомогою спеціального методу створення або за допомогою методу *instantiate:*.

- У світлі першого з них програмний каркас підтримує унарні повідомлення для створення всіх базових компонентів. Ці повідомлення мають вигляд *new[Widget]*, наприклад, *newButton* створює компонент кнопки, а *newList* – компонент списку. Повний список доступних методів створення компонентів можна знайти в класі *SpPresenter* у протоколі «*scripting – widgets*».
- Другий спосіб загальніший. Для повторного використання підкласу *SpPresenter*'а (крім тих, що обробляються першим способом), екземпляр компонента створюють за допомогою методу *instantiate:*. Наприклад, щоб включити *MessageBrowser* до складу демонстратора, потрібно виконати код *self instantiate: MessageBrowser*. Метод *instantiate:* відповідає за створення внутрішнього дерева вкладених демонстраторів, збудованого за відношенням батьківства.

6.7. Метод *connectPresenters*

Метод *connectPresenters* визначає взаємодію між різними візуальними компонентами. Він поєднує поведінку окремих компонентів, і тим самим описує поведінку цілого вікна, тобто те, як увесь графічний інтерфейс реагує на дії користувача. Зазвичай цей метод складається зі специфікацій дій, які потрібно виконати, коли компонент отримує повідомлення про певну подію. Весь потік взаємодії інтерфейсу користувача виникає з поширення таких подій.

Примітка. Метод *connectPresenters* – необов'язковий для побудови інтерфейсу користувача у Spec, але рекомендовано завжди ясно відокремлювати цю поведінку.

Від перекладача. Щоб задати текст напису на кнопці, екземплярові *SpButtonPresenter* надсилають повідомлення *label:*, а щоб задати реакцію на натискання – повідомлення *action:*. Це синтаксично однакові повідомлення, ключові з одним аргументом. Надсилати їх можна в довільному порядку, аби кнопка вже існувала. Тому обидва їх можна надіслати в методі *initializePresenters*, як в застосунку *ColorChooser* з розділу 5. Але перше зі згаданих повідомлень налаштовує вигляд кнопки, а друге – поведінку, тому *label:* варто використати в методі *initializePresenters*, а *action:* – в *connectPresenters*, як у застосунку *CustomerSatisfaction* з розділу 2.

У Spec різні моделі інтерфейсу користувача містяться у носіях значень, а механізм подій покладається на сповіщення від цих носіїв для керування взаємодією між компонентами.

Носії значень надають метод *whenChangedDo:*, який використовують для реєстрації блока, який буде виконано у випадку настання змін, і метод *whenChangedSend: aSelector to: aReceiver* для надсилання повідомлення певному об'єкту. Додатково до цих примітивних методів базові компоненти надають конкретизовані методи-зачіпки, наприклад, реакцію на вибір елемента у списку задають за допомогою *whenSelectionChangedDo:*.

6.8. Метод *defaultLayout*

Макет демонстратора визначають за допомогою методів, які описують, як різні компоненти розташовані в інтерфейсі користувача. Крім того, він також визначає, як демонстратор реагує на зміну розміру вікна. Як побачимо згодом, такі методи можуть мати різні назви.

Метод *defaultLayout* є методом екземпляра, але його також можна визначити на рівні класу. Іншими словами, усі екземпляри одного інтерфейсу користувача мають здебільшого однаковий макет, але макет може бути особливим для одного екземпляра та бути динамічним.

Примітка. Задати макет потрібно обов'язково, бо без нього інтерфейс користувача не відображатиме ніяких візуальних компонентів.

Використання модифікатора *layout:*

Рекомендовано чітко розділити ініціалізацію демонстратора на *initializePresenters* і *defaultLayout*. Проте можна також використовувати повідомлення *layout:*, щоб задати макет на етапі ініціалізації демонстратора.

Кілька макетів для вікна

Для того самого інтерфейсу користувача можна описати кілька макетів, а коли інтерфейс збудовано, можна обумовити використання конкретного макета. Для цього замість виклику *open* (як ми робили досі) надішліть повідомлення *openWithLayout:* з бажаним макетом як аргумент.

6.9. Підсумки розділу

У цьому розділі детальніше описано, як кожен з трьох основних методів Spec: *initialize-Presenters*, *defaultLayout* і *connectPresenters* відповідає за різні аспекти процесу створення інтерфейсу користувача.

Повторне використання основоположне у Spec, але про нього не йшлося явно в цьому розділі. Додаткову інформацію щодо цього питання можна отримати в наступному розділі.

Розділ 7

Повторне використання і композиція в ділі

Головна мета влаштування Spec полягає в тому, щоб уможливити безперебійне повторне використання візуальних компонентів. Зроблено так тому, що це призводить до значного підвищення продуктивності під час створення інтерфейсів користувача.

Увага до повторного використання вже була помітна в попередніх розділах, де видно, що базові візуальні компоненти можна використовувати так, ніби вони є закінченими інтерфейсами користувача. У цьому розділі зосередимося на повторному використанні та композиції демонстраторів і продемонструємо, що такий підхід зазвичай не потребує додаткових затрат. Єдина вимога під час побудови інтерфейсу користувача – розглянути, як він мав би бути параметризований для повторного використання.

Іншими словами, у цьому розділі ви дізнаєтеся, як створити новий інтерфейс користувача через повторне використання вже визначених елементів.

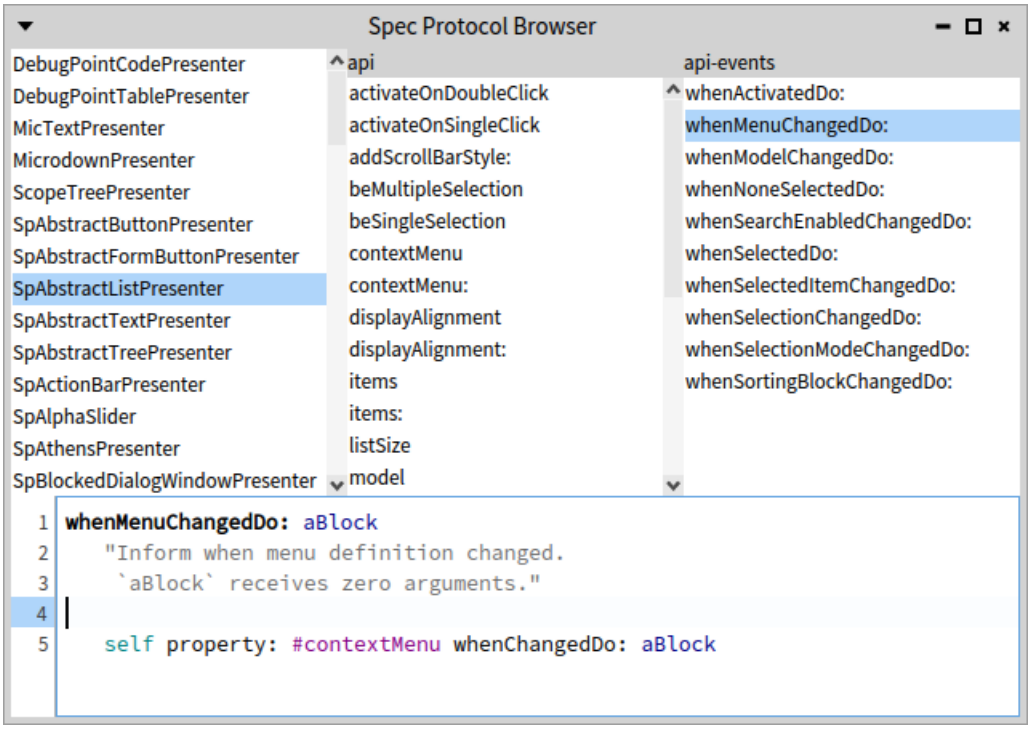


Рис. 7.1. *ProtocolCodeBrowser*: перегляд доступних API візуальних компонентів

7.1. Початкові вимоги

Щоб продемонструвати, як Spec робить можливими композицію та повторне використання інтерфейсів користувача, створимо у цьому розділі застосунок, чий інтерфейс, зображений на рис. 7.1, скомпоновано з чотирьох частин.

1. **WidgetClassListPresenter**: цей демонстратор містить *SpListPresenter* спеціально для відображення підкласів *SpAbstractWidgetPresenter*.

2. **ProtocolMethodListPresenter**: цей демонстратор складено з *SpListPresenter* і *SpLabelPresenter*, щоб відображати селектори методів з окремого протоколу.
3. **ProtocolViewerPresenter**: демонстратор складено з одного *WidgetClassListPresenter* і двох *ProtocolMethodListPresenter*, щоб переглядати методи всіх підкласів *SpAbstractWidgetPresenter*.
4. **ProtocolCodeBrowserPresenter**: демонстратор повторно використовує *ProtocolViewerPresenter*, змінює його макет і додає *SpTextPresenter*, щоб показати програмний код методів.

7.2. Створення інтерфейсу користувача для використання як компонента

Перший інтерфейс користувача, який буде створено, відображає список усіх підкласів класу *SpAbstractWidgetPresenter*. Пізніше його буде повторно використано як компонент для повнішого інтерфейсу. Програмний код виглядає так.

Спочатку створюємо підклас *SpPresenter* з однією змінною екземпляра *list*, яка міститиме екземпляр *SpListPresenter*.

```
SpPresenter << #WidgetClassListPresenter
   slots: { #list };
   package: 'CodeOfSpec20Book'
```

У методі *initializePresenters* створимо список і заповнимо його необхідними класами в алфавітному порядку.

```
WidgetClassListPresenter >> initializePresenters
   list := self newList.
   list items: (SpAbstractWidgetPresenter
      allSubclasses sorted: [:a :b | a name < b name ]).
   self focusOrder add: list
```

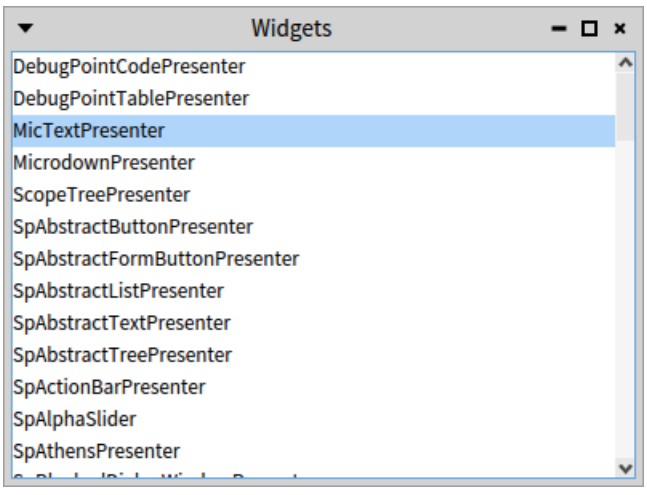


Рис. 7.2. *WidgetClassListPresenter* у окремому вікні

Також задамо заголовок вікна.

```
WidgetClassListPresenter >> initializeWindow: aWindowPresenter
   aWindowPresenter title: 'Widgets'
```

Макет містить тільки список.

```
WidgetClassListPresenter >> defaultLayout
   ^ SpBoxLayout newLeftToRight
        add: #list; yourself
```

Виконання фрагмента *WidgetClassListPresenter new open* мало б відкрити вікно, як на рис. 7.2.

7.3. Підтримка повторного використання

Оскільки цей демонстратор пізніше використовуватимуть разом з іншими для побудови більш довершеного інтерфейсу користувача, потрібно надати можливість налаштовувати реакцію на клацання на елементі списку. Неможливо дізнатися заздалегідь повний перелік дій, які відбуватимуться там, де його буде використано повторно. Тому найліпше рішення – покласти цю відповідальність на користувача компонента. Щоразу, коли цей інтерфейс користувача використають як компонент, програміст повинен буде налаштувати його під власні потреби. Для цього додамо конфігураційний метод *whenSelectionChangedDo:* такого вигляду.

```
WidgetClassListPresenter >> whenSelectionChangedDo: aBlock
   list whenSelectionChangedDo: aBlock
```

Тепер кожен, хто повторно використовує цей компонент, може параметризувати його блоком, який виконуватиметься щоразу, коли вибір буде змінено.

7.4. Компонування двох базових демонстраторів у новий компонент

Далі створимо інтерфейс користувача, який поєднає два компоненти, список і напис, і демонструватиме перелік усіх методів заданого протоколу. З огляду на повторне використання, він не має відмінностей від попереднього інтерфейсу користувача. Так є тому, що на повторне використання інтерфейсу користувача як компонента *ніяк не впливає* ні кількість компонентів, які він містить, ані їхня позиція. Великі та складні інтерфейси користувача повторно використовують так само, як і прості візуальні компоненти.

```
SpPresenter << #ProtocolMethodListPresenter
   slots: { #label . #methods };
   package: 'CodeOfSpec20Book'
```

Метод *initializePresenters* для цього інтерфейсу користувача зовсім простий. Задамо стандартний текст напису «Protocol». Його буде змінено під час повторного використання компонента.

```
ProtocolMethodListPresenter >> initializePresenters
   methods := self newList.
   methods display: [ :m | m selector ].
   label := self newLabel.
   label label: 'Protocol'.
   self focusOrder add: methods
```

Щоб надати вікну, в якому відкриватиметься демонстратор, змістовний заголовок, визначимо метод *initializeWindow:*.

```
ProtocolMethodListPresenter >> initializeWindow: aWindowPresenter
   aWindowPresenter title: 'Protocol widget'
```

Код макета формує стовпець з написом над списком методів.

```
ProtocolMethodListPresenter >> defaultLayout
   ^ SpBoxLayout newTopToBottom
         add: #label;
         add: #methods;
         yourself
```

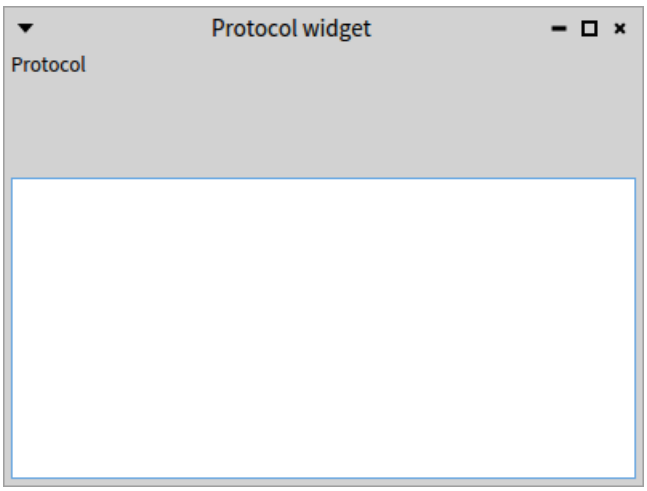


Рис. 7.3. *ProtocolMethodListPresenter* з неякісним макетом

Цей інтерфейс користувача можна побачити, виконавши *ProtocolMethodListPresenter new open*. Як зображено на рис. 7.3, список порожній, і результат не надто гарний. Це не дивно, бо не задано вміст списку. Також потрібно краще розмістити компоненти.

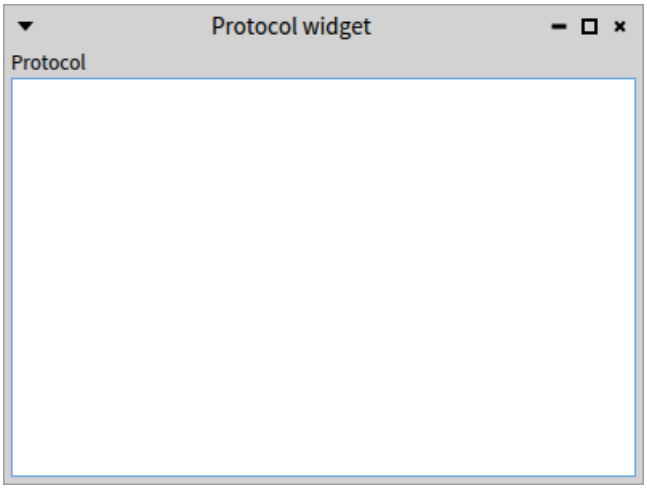


Рис. 7.4. *ProtocolMethodListPresenter* з виправленим макетом

```
ProtocolMethodListPresenter >> defaultLayout
   ^ SpBoxLayout newTopToBottom
         add: #label expand: false;
         add: #methods;    yourself
```

Тепер інтерфейс користувача мав би виглядати краще як на рис. 7.4.

Наш компонент для відображення списку методів заданого протоколу потрібно налаштовувати перед використанням: треба заповнити список методів і зазначити назву протоколу. Задля підтримки налаштування додамо деякі конфігураційні методи.

```
ProtocolMethodListPresenter >> items: aCollection
   methods items: aCollection

ProtocolMethodListPresenter >> label: aText
   label label: aText

ProtocolMethodListPresenter >> resetSelection
   methods selection unselectAll

ProtocolMethodListPresenter >> whenSelectionChangedDo: aBlock
   methods whenSelectionChangedDo: aBlock
```

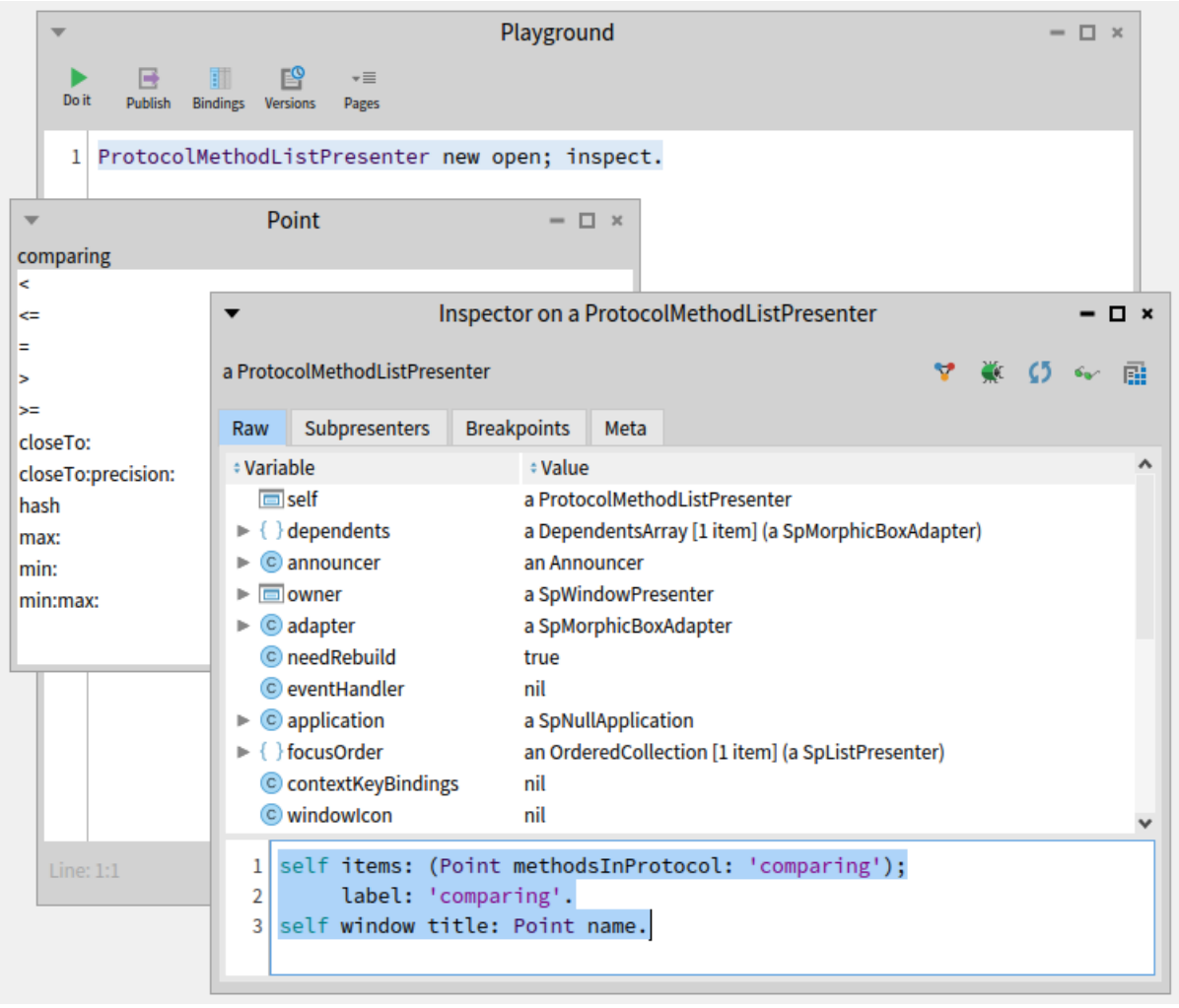


Рис. 7.5. Безпосередня взаємодія з програмованим компонентом

7.5. Інспектування «живих» компонентів

Тепер можна вручну перевірити, чи працює компонент, виконавши:

```
ProtocolMethodListPresenter new open; inspect
```

Тоді в інспекторі використаємо щойно створений демонстратор для перевірки конфігурування. Перегляньте результат на рис. 7.5.

```
self items: (Point methodsInProtocol: 'comparing');
     label: 'comparing'.
self window title: Point name.
```

Можна продовжити експерименти і, наприклад, впорядкувати елементи списку так:

```
self items: (Point methods sort: #selector ascending)
```

7.6. Написання тестів

Коли виникає бажання вручну перевірити написане, то це означає, що пора написати тест. Легко писати прості тести для компонентів, коли не йдеться про виринаючі вікна. Тож скористаймося цим.

Додамо селектор для доступу до списку методів і оголосимо клас тестів.

```
ProtocolMethodListPresenter >> methods
   ^ methods

TestCase << #ProtocolMethodListPresenterTest
   slots: {};
   package: 'CodeOfSpec20Book'

ProtocolMethodListPresenterTest >> testItems
   | proto methods |
   methods := Point methods sort: #selector ascending.
   proto := ProtocolMethodListPresenter new.
   proto items: methods.
   self assert: proto methods items first class
      equals: CompiledMethod.
   self assert: proto methods items first selector
      equals: methods first selector
```

Сподіваємося, що ми переконали вас, що тестувати інтерфейс користувача легко зі Spec. Не пропустіть цю можливість опанувати складність свого програмного забезпечення.

7.7. Управління трьома компонентами та їхньою взаємодією

Третій інтерфейс користувача, який ми створюємо, є композицією двох попередніх. Буде видно, що немає різниці між налаштуванням власного демонстратора та налаштуванням системних компонентів: обидва типи компонентів налаштовують викликами методів протоколу «*api*».

Новий демонстратор складається з одного *WidgetClassListPresenter* і двох *ProtocolMethodListPresenter*. Його поведінка полягає у тому, що коли користувач вибере клас у *WidgetClassListPresenter*, методи цього класу з протоколів «*api*» та «*api-events*» будуть зображері у двох компонентах *ProtocolMethodListPresenter*.

```
SpPresenter << #ProtocolViewerPresenter
   slots: { #models . #api . #events };
   package: 'CodeOfSpec20Book'
```

Метод *initializePresenters* демонструє використання *instantiate:* для створення екземплярів нестандартних компонентів і деякі з різних методів конфігурування класу *ProtocolMethodListPresenter*.

```
ProtocolViewerPresenter >> initializePresenters
   models := self instantiate: WidgetClassListPresenter.
   api := self instantiate: ProtocolMethodListPresenter.
   events := self instantiate: ProtocolMethodListPresenter.
   api label: 'api'.
   events label: 'api-events'.
   self focusOrder
      add: models;
      add: api;
      add: events

ProtocolViewerPresenter >> initializeWindow: aWindowPresenter
   aWindowPresenter title: 'Protocol viewer'
```

Щоб описати взаємодію між різними компонентами, оголосимо метод *connectPresenters*. У ньому визначено, що як тільки вибрано клас, скидається вибір у обидвох списках методів, а самі вони заповнюються новими значеннями. Крім того, якщо вибрано метод в одному списку методів, то вибір в іншому скидається.

```
ProtocolViewerPresenter >> connectPresenters
   models whenSelectionChangedDo: [ :selection |
      | class |
      api resetSelection.
      events resetSelection.
      class := selection selectedItem.
      class
         ifNil: [
            api items: #().
            events items: #() ]
         ifNotNil: [
            api items: (self methodsIn: class for: 'api').
            events items: (self methodsIn: class for: 'api - events') ]
      ].
   api whenSelectionChangedDo: [ :selection |
      selection selectedItem ifNotNil: [ events resetSelection ]
      ].
   events whenSelectionChangedDo: [ :selection |
      selection selectedItem ifNotNil: [ api resetSelection ]
      ]

ProtocolViewerPresenter >> methodsIn: class for: protocol
   ^ (class methodsInProtocol: protocol)
         sorted: [ :a :b | a selector < b selector ]
```

Нарешті, макет поміщає піддемонстратори в один стовпець, причому всі вони займають однакові по висоті частини вікна.

```
ProtocolViewerPresenter >> defaultLayout
   ^ SpBoxLayout newTopToBottom
      add: #models; add: #api; add: #events;
      yourself
```

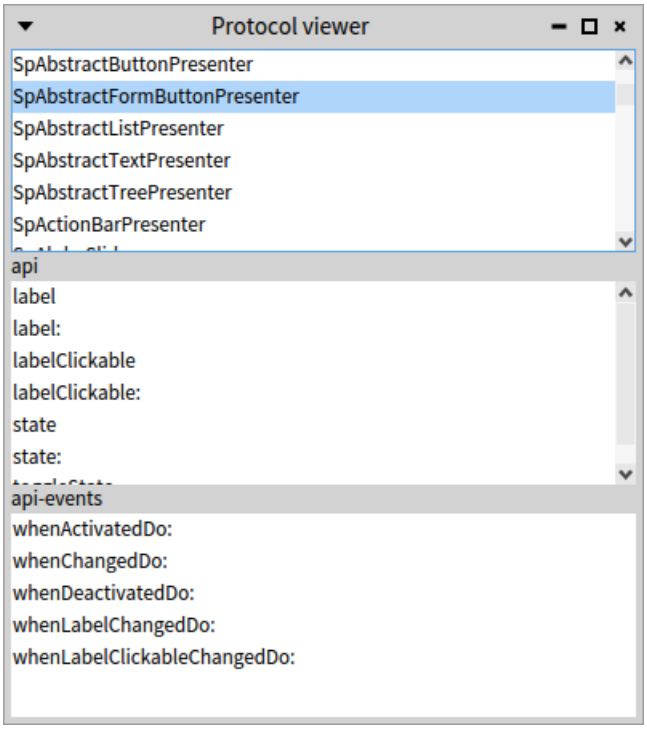


Рис. 7.6. *ProtocolViewerPresenter* у вертикальному режимі

Як і раніше, готовий демонстратор можна побачити у вікні, виконавши фрагмент коду *ProtocolViewerPresenter new open*. Результат зображено на рис. 7.6.

Цей демонстратор цілком функціональний. Клацнувши на класі, ви побачите методи протоколів «*api*» та «*api-events*» цього класу.

Від перекладача. Як усталено, вікно застосунку має початковий розмір 400 на 300 пікселів, чого замало, щоб продемонструвати всі три списки. Вам доведеться розтягти його донизу, або оголосити метод *initializeWindow:* у такій редакції:

```
ProtocolViewerPresenter >> initializeWindow: aWindowPresenter
    aWindowPresenter
       title: 'Protocol viewer';
       initialExtent: 400@450
```

7.8. Використання різних макетів

Зауважимо, що можна змінити макет так, щоб розташувати всі компоненти в рядок, як зображено на рис. 7.7. Згодом покажемо, що демонстратор може мати кілька макетів, і що програміст вирішує, який з них використати.

Можемо зробити ще ліпше. Визначимо два такі методи.

```
ProtocolViewerPresenter >> horizontalLayout
   ^ SpBoxLayout newLeftToRight
      add: #models;
      add: #api;
      add: #events;
      yourself
```

```
ProtocolViewerPresenter >> verticalLayout
   ^ SpBoxLayout newTopToBottom
      add: #models;
      add: #api;
      add: #events;
      yourself
```

І змінимо оголошення стандартного макета.

```
ProtocolViewerPresenter >> defaultLayout
   ^ self verticalLayout
```

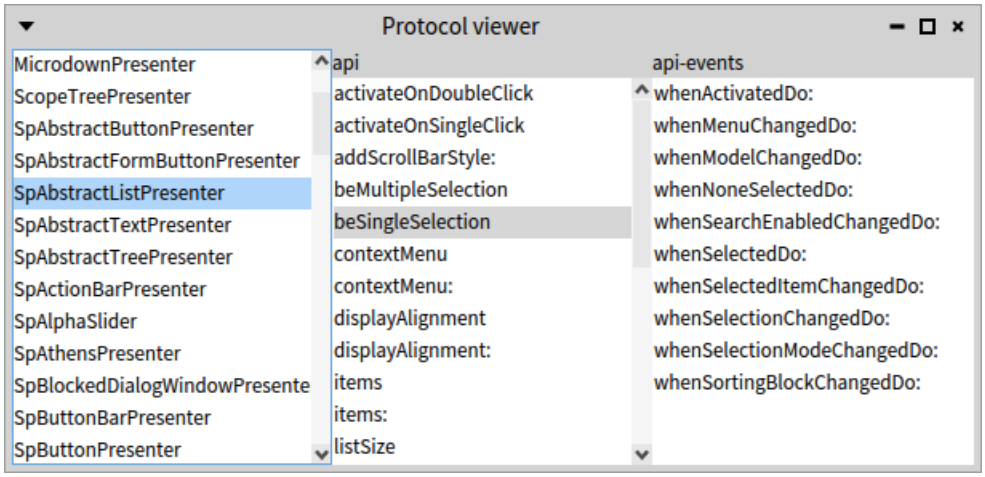


Рис. 7.7. *ProtocolViewerPresenter* у горизонтальному режимі

Тепер можна вирішувати, з яким макетом відкрити вікно. Щоб задати макет, використовують повідомлення *openWithLayout:*, як у прикладі нижче. Вікно мало б виглядати як на рис. 7.7.

```
ProtocolViewerPresenter class >> exampleHorizontal
    | inst |
    instance := self new.
    instance openWithLayout: instance horizontalLayout
```

7.9. Розширення API

Коли *ProtocolViewerPresenter* використовуватиметься повторно, його, ймовірно, потрібно буде налаштовувати, подібно як це було з *ProtocolMethodListPresenter*. Відповідне конфігурування тут полягає в тому, щоб задати, що робити, коли змінюється вибір у будь-якому з трьох списків. Тому додамо такі три методи до протоколу «*api*».

```
ProtocolViewerPresenter >> whenSelectionInClassChanged: aBlock
   models whenSelectionChangedDo: aBlock

ProtocolViewerPresenter >> whenSelectionInAPIChanged: aBlock
   api whenSelectionChangedDo: aBlock

ProtocolViewerPresenter >> whenSelectionInEventChanged: aBlock
   events whenSelectionChangedDo: aBlock
```

Зауваження. Ці методи додають семантичну інформацію до API конфігурації. Вони повідомляють, що налаштовують реакцію на зміну вибору в списку класів, «*api*» або «*api-events*». Безперечно, такі методи чіткіше пояснюють можливості налаштування, ніж просто наявність доступу до вкладених демонстраторів.

7.10. Зміна макета використаного компонента

Іноді, коли ви хочете повторно використати наявний інтерфейс користувача як компонент, його макет не відповідає вашим потребам. Не біда, Spec дає змогу використати і такий інтерфейс, надаючи засоби для перевантаження макета. Далі покажемо як це зробити.

Четвертий, останній у цьому розділі інтерфейс користувача повторно використовує *ProtocolViewerPresenter*, змінює його макет і додає текстове поле для відображення програмного коду вибраного методу.

```
SpPresenter << #ProtocolCodeBrowserPresenter
   slots: { #text . #viewer };
   package: 'CodeOfSpec20Book'
```

```
ProtocolCodeBrowserPresenter >> initializePresenters
   text := self instantiate: SpCodePresenter.
   viewer := self instantiate: ProtocolViewerPresenter.
   text syntaxHighlight: true.
   self focusOrder
      add: viewer;
      add: text

ProtocolCodeBrowserPresenter >> defaultLayout
   ^ SpBoxLayout newTopToBottom
         add: (SpBoxLayout newLeftToRight add: #viewer; yourself);
         add: #text;
         yourself

ProtocolCodeBrowserPresenter >> initializeWindow: aWindowPresenter
   aWindowPresenter title: 'Spec Protocol Browser'
```

Метод *connectPresenters* задає реакцію текстового поля на вибір у списках. Коли метод вибрано, текстове поле оновлює свій вміст, щоб показати його програмний код.

```
ProtocolCodeBrowserPresenter >> connectPresenters
   viewer whenSelectionInClassChanged: [ :selection |
      text behavior: selection selectedItem
   ].
   viewer whenSelectionInAPIChanged: [ :selection |
      selection selectedItem ifNotNil: [ :item |
         text beForMethod: item;
            text: item sourceCode ]
      ].
   viewer whenSelectionInEventChanged: [ :selection |
      selection selectedItem ifNotNil: [ :item |
         text beForMethod: item;
            text: item sourceCode ] ]
```

Оглядач коду методів майже готовий. Відкрити його вікно можна за допомогою коду *ProtocolCodeBrowserPresenter new open*. За наявної реалізації *initializePresenters* екземпляр *ProtocolViewerPresenter*, який зберігається в змінній екземпляра *viewer*, розташує свої вкладені демонстратори у стовпець, бо його стандартний макет вертикальний. Нам би пасувало використати інший макет, щоб розташувати демонстратори в ряд. Цього можна досягти, надіславши до *viewer* повідомлення *layout:* з потрібним макетом. Отож адаптуймо *initializePresenters* так.

```
initializePresenters
   text := self instantiate: SpCodePresenter.
   viewer := self instantiate: ProtocolViewerPresenter.
   viewer layout: viewer horizontalLayout.
   text syntaxHighlight: true.
   self focusOrder
      add: viewer;
      add: text
```

Тепер відкриється вікно як на рис. 7.1.

Від перекладача. Легко переконатися, що вибір нового класу скидає виокремлення зі списків методів, але не очищує поле з текстом відображеного раніше методу. Що виправити цей недолік, варто точніше налаштувати опрацювання події *whenSelectionInClassChanged:* в методі *connectPresenters*:

```
ProtocolCodeBrowserPresenter >> connectPresenters
   viewer whenSelectionInClassChanged: [ :selection |
      text behavior: selection selectedItem;
         text: ''
   ].   . . .
```

7.11. Заміна макета

Існують різні способи, як можна задати макет демонстратора. Продемонструймо їх на прикладі *ProtocolViewerPresenter*. Перший варіант полягає у використанні для відкривання вікна методу *openWithLayout:*.

```
presenter := ProtocolViewerPresenter new.
presenter openWithLayout: (SpBoxLayout newLeftToRight
   add: #models;
   add: #api;
   add: #events;
   yourself)
```

Або можна надіслати демонстратору повідомлення *layout:*, щоб задати макет, а вікно відкрити згодом.

```
presenter := ProtocolViewerPresenter new.
presenter layout: (SpBoxLayout newLeftToRight
   add: #models;
   add: #api;
   add: #events;
   yourself).
presenter open
```

Альтернативний спосіб полягає у використанні макета, наданого демонстратором, як було в попередньому параграфі.

```
presenter := ProtocolViewerPresenter new.
presenter layout: presenter horizontalLayout.
presenter open
```

7.12. Міркування щодо загальнодоступного API конфігурування

У цьому розділі було описано кілька визначень методів у загальнодоступному API конфігурування проєктованого демонстратора. Реалізація тих методів полягала в делегуванні внутрішнім компонентам, але, залежно від внутрішньої логіки інтерфейсу користувача, конфігурування може бути куди складнішим, ніж описане.

Щодо методів, які тільки делегують внутрішнім компонентам, виникає питання, чи є сенс взагалі визначати їх як методи в протоколах «*api*». По суті, це проєктне рішення, яке приймає програміст. Відсутність таких методів робить реалізацію демонстратора легшою, але тоді потрібні методи-селектори, а розплатою буде менша виразність коду та порушення інкапсуляції.

Для оцінки попередніх затрат на розробку можна розглянути приклад оглядача методів класу з параграфа 7.7. Наявність у нього трьох визначених API методів повідомляє користувачеві, що ми турбуємося про те, як реагувати на зміну вибору елемента зі списку класів, чи списку «*api*», чи «*api-events*». Загалом те саме стосується й інших прикладів у цьому розділі: кожен API метод повідомляє користувачеві демонстратора посил – очікується, що саме так цей демонстратор буде налаштований. Без оголошення таких методів програмістові буде не дуже зрозуміло, що можна зробити для ефективного налаштування демонстратора.

Стосовно майбутніх затрат. Якщо розраховувати на те, що програміст, який використовує наш компонент, має безпосередньо надсилати повідомлення до внутрішніх об'єктів (наприклад, змінних екземпляра), то це означає порушення інкапсуляції. Як наслідок, ми більше не маємо права змінювати внутрішні елементи компонента, наприклад, давати змінним екземпляра доречніші імена або змінювати тип використаного компонента. Такі зміни можуть порушити повторне використання компонента, і отже, суттєво обмежити можливості розвитку цього компонента в майбутньому. Безпечніше визначити загальнодоступний API і переконатися, що в майбутніх версіях демонстратора функціональні можливості цього API залишаться незмінними.

Тому, зрештою, важливо брати до уваги потреби майбутніх користувачів вашого компонента та майбутню еволюцію інтерфейсу користувача. Потрібно знайти компроміс між написанням додаткових методів і, можливо, ускладненням повторного використання компонента, а також, можливо, ускладненням майбутнього розвитку програмного продукту.

7.13. Нові шаблони проти старих

У Spec 1.0 демонстратори списків надавали інший API, а саме *whenSelectedItemChanged:*, як у такому прикладі.

```
initializePresenters
   models := self instantiate: WidgetClassListPresenter.
```

```
        api := self instantiate: ProtocolMethodListPresenter.
        events := self instantiate: ProtocolMethodListPresenter.
        api label: 'api'.
        events label: 'api-events'

    connectPresenters
        api whenSelectedItemChanged: [ :method |
            method ifNotNil: [ events resetSelection ] ].
        events whenSelectedItemChanged: [ :method |
            method ifNotNil: [ api resetSelection ] ]
```

У Spec 2.0 демонстратори списків і їм подібні надають спеціальний об'єкт, який представляє вибір зі списку. Зроблено так тому, що виділення є складним об'єктом (один вибір, множинний вибір). Отже, маємо:

```
    connectPresenters
        api whenSelectionChangedDo: [ :selection |
            selection selectedItem ifNotNil: [ events resetSelection ] ].
        events whenSelectionChangedDo: [ :selection |
            selection selectedItem ifNotNil: [ api resetSelection ] ]
```

Питання стосовно ваших компонентів полягає в тому, який API потрібно надавати користувачам. Якщо вам подобається спосіб Spec 1.0, то це все ще можливо, як показано нижче.

```
    whenSelectedItemChangedDo: aBlock
        methods whenSelectionChangedDo: [ :selection |
            selection selectedItem ifNotNil: [ :item | aBlock value: item ] ]
```

Але рекомендовано використовувати спосіб Spec 2.0, бо так ви узгодите свої компоненти з базовими компонентами Spec, і легше буде налагодити їхню взаємодію.

7.14. Підсумки розділу

У цьому розділі йшлося про ключовий момент Spec: можливість безперешкодного повторного використання наявних інтерфейсів користувача як компонентів. Автору нового інтерфейсу таке використання доступне без значних затрат. Єдине, що потрібно взяти до уваги, це те, як можна (чи потрібно) налаштувати інтерфейс користувача.

Повторне використання складних компонентів без значних витрат було ключовою метою розробки Spec, оскільки воно забезпечує суттєве підвищення продуктивності в процесі написання інтерфейсів користувача. Підвищення зумовлене, по-перше, доступністю наявних нетривіальних компонентів для повторного використання, по-друге, можливістю структурувати свій інтерфейс користувача на узгоджені та легше керовані частини зі зрозумілими інтерфейсами. Тому радимо вам думати про свій інтерфейс користувача як про композицію таких частин і будувати його модульно, щоб збільшити швидкість написання та полегшити супровід.

Розділ 8
Списки, таблиці та дерева

Важливою частиною інтерфейсу користувача є відображення списків даних. Вони можуть бути структуровані як таблиці, звичайні списки, або дерева, які підтримують вкладення даних.

Spec надає три основні демонстратори: *SpListPresenter*, *SpTreePresenter* і *SpTablePresenter*. Крім того, він пропонує *SpComponentListPresenter*, який дає змогу вставляти будь-який демонстратор у список. У розділі описано деякі функції цих демонстраторів.

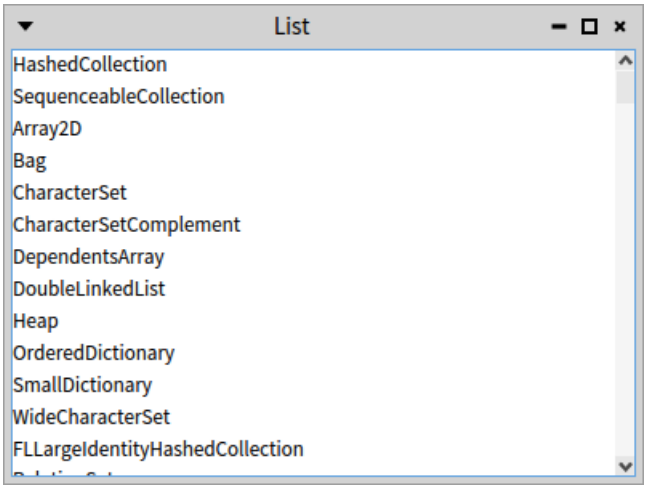


Рис. 8.1. Простий список демонструє імена класів

8.1. Списки

Створити демонстратор списку так само просто, як створити екземпляр *SpListPresenter* і задати список елементів, які він має відображати. Приклад такого коду подано нижче, а результат зображено на рис. 8.1.

```
SpListPresenter new
   items: Collection withAllSubclasses;
   open
```

Над елементами списку можна відобразити заголовок. Для цього використовують повідомлення *headerTitle:* з аргументом – непорожнім рядком. Прибрати заголовок можна за допомогою повідомлення *hideHeaderTitle*.

8.2. Налаштування способу відображення елементів

Усталений спосіб відображення елементів списку надсилає повідомлення *asStringOrText* кожному елементу і використовує отримані результати. Щоб власноруч керувати способом відображення, спискові надсилають повідомлення *display:* і передають у ньому блок, який за заданим елементом списку будує рядок. Цей блок буде застосовано до кожного елемента списку. Показаний нижче скрипт налаштовує презентатор списку

для відображення назв методів класу *Point* (див. рис. 8.2 праворуч) замість показу результатів *asStringOrText* (на рис. 8.2 ліворуч).

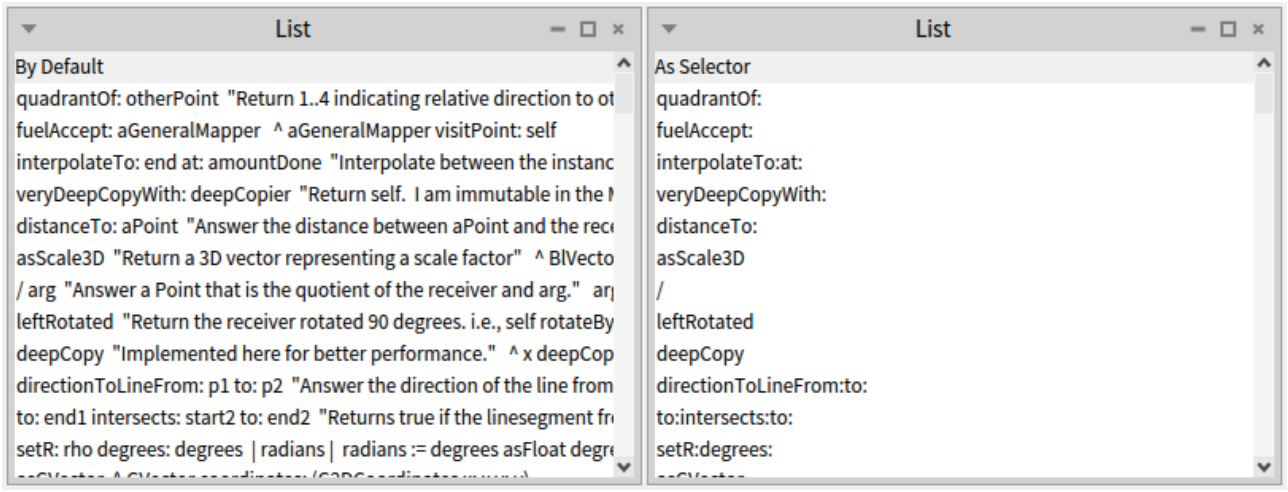


Рис. 8.2. Два способи відображення методів класу в списку: ліворуч – усталений спосіб виводить тексти методів, праворуч – налаштований спосіб виводить селектори методів

```
SpListPresenter new
   items: Point methods;
   headerTitle: 'As Selector';
   display: [ :item | item selector ];
   open
```

Елементи списку можна впорядкувати за допомогою повідомлення *sortingBlock:*.

```
SpListPresenter new
   items: Point methods;
   display: [ :item | item selector ];
   sortingBlock: [ :a :b | a selector < b selector ];
   open
```

8.3. Оздоблення елементів списку

Спосіб відображення елементів можна налаштувати детальніше. Такі можливості ілюструє приклад нижче. Налаштувати можна такі параметри: піктограму, пов'язану з елементом, задають повідомленням *displayIcon:*, колір елемента – повідомленням *displayColor:*, формат шрифту (товстий, курсив, підкреслений) – за допомогою відповідних повідомлень *displayItalic:*, *displayBold:* і *displayUnderline:* (див. рис. 8.3).

```
presenter := SpListPresenter new.
presenter
   items: Collection withAllSubclasses;
   displayIcon: [ :aClass | self iconNamed: aClass systemIconName ];
   displayColor: [ :aClass |
      (aClass name endsWith: 'Set')
         ifTrue: [ Color green ]
         ifFalse: [ presenter theme textColor ] ];
   displayItalic: [ :aClass | aClass isAbstract ];
   displayBold: [ :aClass | aClass hasSubclasses ];
   displayUnderline: [ :aClass | aClass numberOfMethods > 10 ];
   open
```

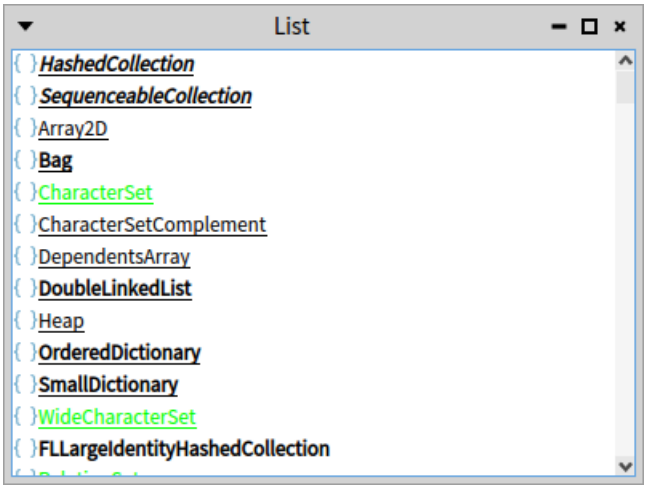


Рис. 8.3. Оздоблений список: додано піктограми, стилі шрифтів, колір тексту

8.4. Про одиничний або множинний вибір

Списки підтримують множинний вибір. Повідомлення *beMultipleSelection* вмикає цю можливість.

```
SpListPresenter new
   items: Collection withAllSubclasses;
   beMultipleSelection;
   open
```

Протокол реагування на зміну вибору відображає те, що виокремлення може містити кілька елементів. Справді, списки, фільтровані списки, дерева і таблиці пропонують API *whenSelectionChangedDo:*, а не *whenSelectedItemDo:*. Аргументом цього повідомлення є блок, чиїм параметром є екземпляр *SpSingleSelectionMode*, *SpMultipleSelectionMode*, *SpTreeMultipleSelectionMode* або *SpTreeSingleSelectionMode*.

Ось типовий випадок використання методу *whenSelectionChangedDo:*.

```
connectPresenters
   changesTree whenSelectionChangedDo: [ :selection | | diff |
      diff := selection selectedItem
         ifNil: [ '' ]
         ifNotNil: [ :item | self buildDiffFor: item ].
      textArea text: diff ]
```

8.5. Перетягування і скидання

Списки та інші контейнери підтримують перетягування. Зображений нижче скрипт показує, як налаштувати два списки для підтримки перетягування з одного та скидання на інший.

```
| list1 list2 |
list1 := SpListPresenter new.
list1
   items: #( 'abc' 'def' 'xyz' );
   dragEnabled: true.
```

```
list2 := SpListPresenter new.
list2 dropEnabled: true;
   wantsDrop: [ :transfer | transfer passenger
      allSatisfy: [:each | each isString ] ];
   acceptDrop: [ :transfer |
      list2 items: list2 items , transfer passenger ].

SpPresenter new
   layout: (SpBoxLayout newLeftToRight
      add: list1;
      add: list2;
      yourself);
   open
```

Пояснимо використані методи API.

- *dragEnabled:* налаштовує джерело даних, щоб дозволити перетягування своїх елементів.
- *dropEnabled:* налаштовує приймач даних, щоб приймати скинуті елементи.
- *wantsDrop: [:transfer | transfer passenger allSatisfy: [:each | each isString]]*. Аргумент повідомлення *wantsDrop:* – блок-предикат. Він описує умову, яку повинні задовольняти скинуті елементи.
- *acceptDrop: [:transfer | list2 items: list2 items , transfer passenger]*. Аргумент повідомлення *acceptDrop:* визначає, як обробити прийняті скинуті елементи.

8.6. Активування клацанням

Елемент списку може бути *активовано*, тобто він ініціюватиме подію для виконання дії над ним. Зауважте, що активація відрізняється від вибору: можна *вибрати* елемент, не активуючи його. Повідомлення *activateOnDoubleClick* змушує список реагувати на подвійне клацання, а його антиподом є *activateOnSingleClick*.

Від перекладача. Повідомлення *activateOnDoubleClick* і *activateOnSingleClick* керують станом змінної *activateOnSingleClick* екземпляра демонстратора списку. Її стандартне значення – *false*. У такому стані звичайне клацання на елементі списку спричинятиме зміну вибору, а подвійне – активацію. Щоб задати опрацювання активації, використовують повідомлення *whenActivatedDo: [:selection | ...]*.

8.7. Списки з фільтром

Списки можуть фільтрувати свої елементи, як зображено на рис. 8.4. Фрагмент коду нижче демонструє використання *SpFilteringListPresenter*.

```
SpFilteringListPresenter new
   items: Collection withAllSubclasses;
   open;
   withWindowDo: [ :window |
      window title: 'SpFilteringListPresenter example' ]
```

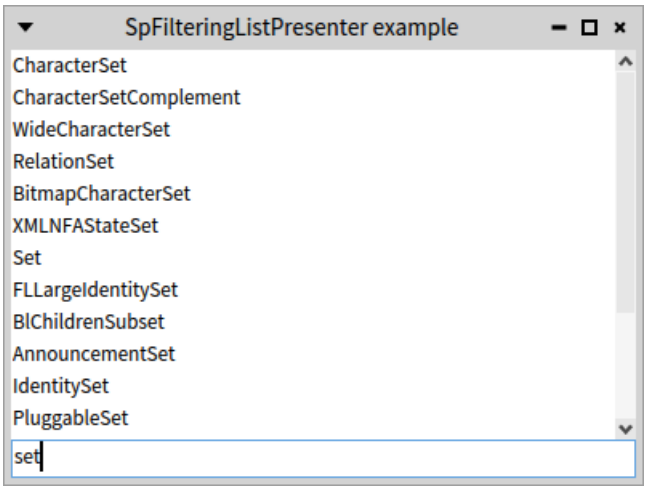


Рис. 8.4. Фільтрований список з рядком фільтра внизу

Ще один приклад демонструє, що фільтр можна розмістити вгорі.

```
SpFilteringListPresenter new
   items: Collection withAllSubclasses;
   openWithLayout: SpFilteringListPresenter topLayout;
   withWindowDo: [ :window |
      window title: 'SpFilteringListPresenter example' ]
```

Зауважимо, що фільтр можна застосувати заздалегідь за допомогою повідомлення *applyFilter:*.

```
SpFilteringListPresenter new
   items: Collection withAllSubclasses;
   openWithLayout: SpFilteringListPresenter topLayout;
   applyFilter: 'set';
   withWindowDo: [ :window |
      window title: 'SpFilteringListPresenter prefiltered example' ]
```

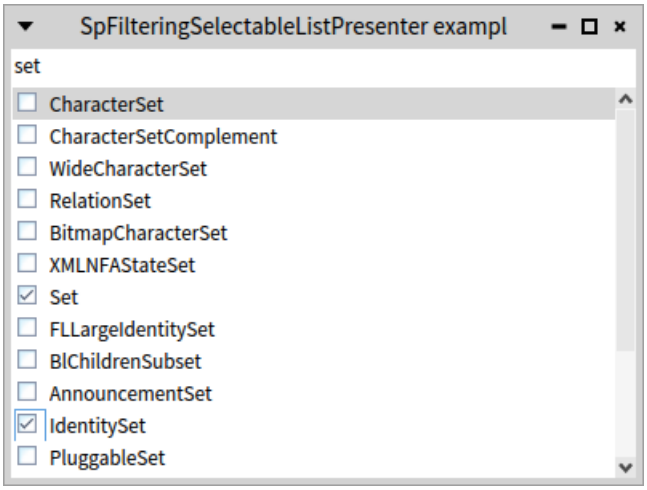


Рис. 8.5. Список з можливістю позначення та фільтром угорі

8.8. Фільтровані списки з можливістю позначення

Списки часто використовують для вибору елементів. Таку можливість пропонує клас *SpFilteringSelectableListPresenter*. На додаток до можливості фільтрувати елементи він дає змогу користувачеві відзначати елементи галочками, як зображено на рис. 8.5.

Такий програмний код створює список з фільтром і можливістю позначення елементів.

```
(SpFilteringSelectableListPresenter new
   items: Collection withAllSubclasses;
   layout: SpFilteringListPresenter topLayout;
   applyFilter: 'set';
   asWindow)
      title: 'SpFilteringSelectableListPresenter example';
      open
```

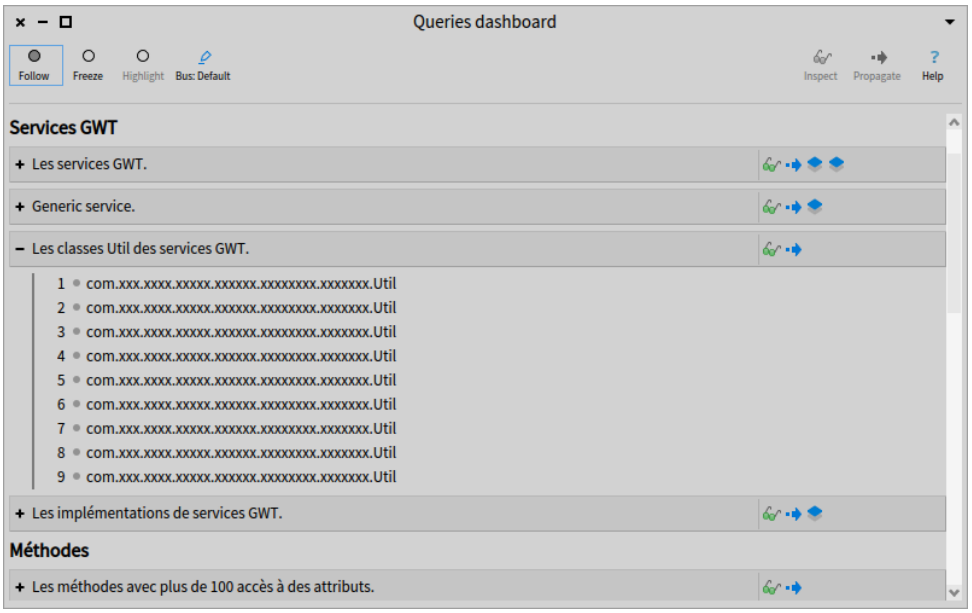


Рис. 8.6. Приклад списку компонентів з платформи ModMoose

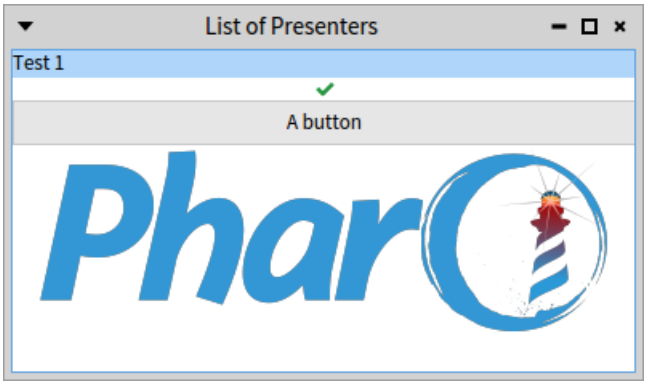


Рис. 8.7. Список компонентів з кількома різними демонстраторами: напис, піктограма, кнопка, зображення

8.9. Списки компонентів

Усі списки, які ми бачили досі, були однорідними в тому сенсі, що всі вони відображали рядки. Проте Spec надає можливість відображати список демонстраторів. Це означає, що елементи демонстратора списку можуть мати неоднаковий розмір і містити інші демонстратори.

Це допомагає розробникам створювати розвинені інтерфейси користувача, як інтерфейс конструктора звітів із набору інструментів ModMoose, показаний на рис. 8.6.

Такий код демонструє, як визначити *SpComponentListPresenter*, зображений на рис. 8.7.

```
| list |
list := {
   (SpLabelPresenter new
      label: 'Test 1';
      yourself).
   (SpImagePresenter new
      image: (self iconNamed: #smallOk);
      yourself).
   (SpButtonPresenter new
      label: 'A button';
      yourself).
   (SpImagePresenter new
      image: PolymorphSystemSettings pharoLogo asForm;
      yourself) }.

SpComponentListPresenter new
   presenters: list;
   open;
   withWindowDo: [ :window |
      window title: 'List of Presenters' ]
```

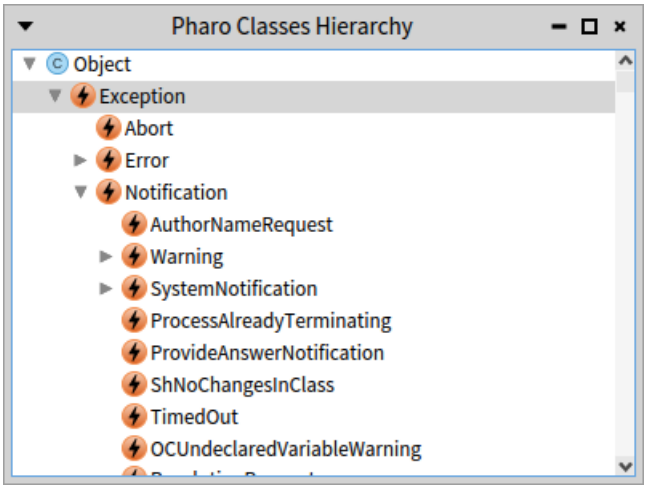


Рис. 8.8. Демонстратор дерева демонструє підкласи класу *Exception*

8.10. Дерева

Також Spec уміє будувати дерева. Наведений нижче код демонструє, як відобразити дерево всіх класів Pharo, організоване за відношенням наслідування (див. рис. 8.8).

```
(SpTreePresenter new
   roots: { Object };
   children: [ :aClass | aClass subclasses ];
   displayIcon: [ :aClass | self iconNamed: aClass systemIconName ];
   display: [ :aClass | aClass name ];
   expandPath: #( 1 1 3 ); asWindow)
      title: 'Pharo Classes Hierarchy'; open
```

Повідомлення *expandPath:* задає шлях по дереву, вузли якого відобразяться розгорнутими. Перша одиниця в масиві означає корінь дерева, а наступні елементи – номери вузлів на відповідному рівні дерева.

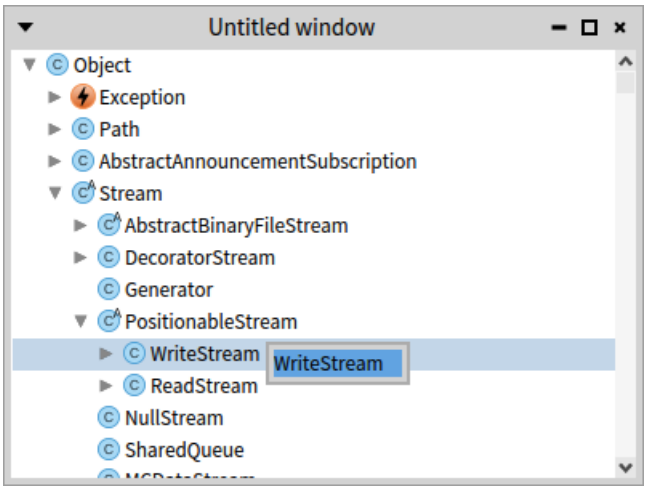


Рис. 8.9. Дерево з контекстним меню

Презентатор дерева можна оснастити динамічним контекстним меню. Такий фрагмент демонструє, як це зробити. Меню динамічне, бо його вміст конструюється залежно від обраного вузла дерева. Спосіб конструювання виражає блок. На рис. 8.9 показано результат.

```
| tree |
tree := SpTreePresenter new.
tree roots: { Object };
   children: [ :aClass | aClass subclasses ];
   displayIcon: [ :aClass | self iconNamed: aClass systemIconName ];
   display: [ :aClass | aClass name ];
   contextMenu: [
      SpMenuPresenter new
         addGroup: [ :group | group addItem: [ :item |
               item name: tree selectedItem asString ] ] ];
   open
```

Вузол дерева можна вибрати програмно. Для цього використовують повідомлення *selectPathByItems: scrollToSelection:*. У першому аргументі передають колекцію, шлях від кореня до вибраного вузла, а в другому – ознаку того, чи потрібно прокручувати демонстратор до місця вибору. Нижче наведено приклад коду, а на рис. 8.10 – результат.

```
| pathToSpPresenter |

pathToSpPresenter :=
   SpTreePresenter withAllSuperclasses reversed allButFirst.
```

```
SpTreePresenter new
   roots: { Object };
   children: [ :aClass | aClass subclasses ];
   displayIcon: [ :aClass | self iconNamed: aClass systemIconName ];
   display: [ :aClass | aClass name ];
   open;
   selectPathByItems: pathToSpPresenter scrollToSelection: true
```

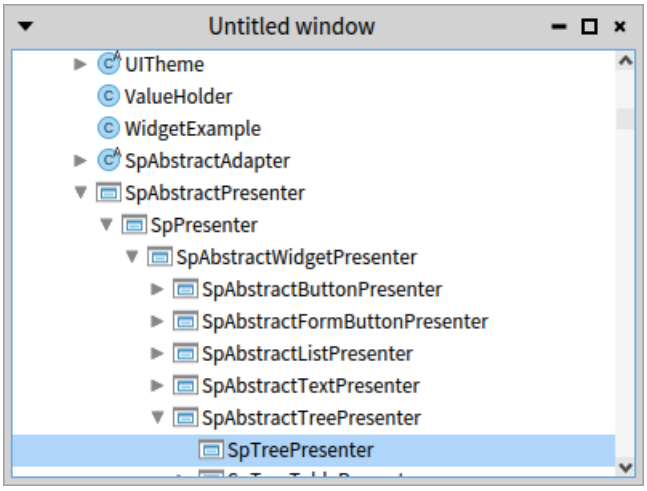


Рис. 8.10. Зображення дерева, прокручене до програмно вибраного вузла

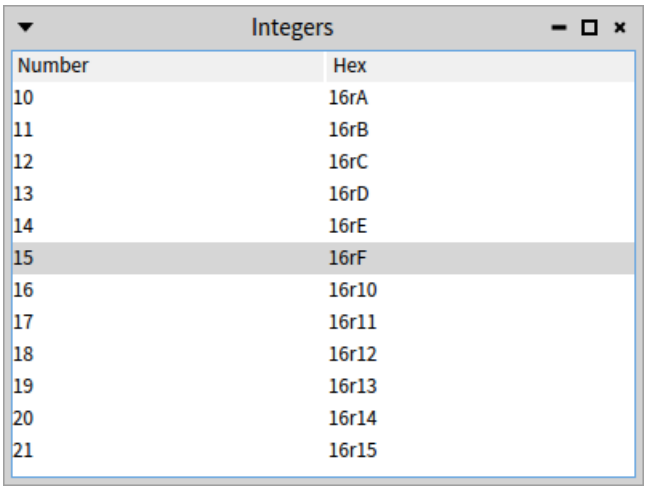


Рис. 8.11. Проста таблиця з двома стовпцями

8.11. Таблиці

Spec пропонує демонстратори таблиць. Таблиця може мати кілька стовпців, а стовпець може складатися з елементарних комірок. Стовпці додають до таблиці під час її побудови, причому вони можуть бути різних типів:

- *SpStringTableColumn* пропонує комірки – рядки.
- *SpCheckBoxTableColumn* надає комірки з прапорцем.
- *SpIndexTableColumn* відображає індекс поточного елемента.
- *SpDropListTableColumn* дає змогу розмістити в комірці розкривний список.
- *SpImageTableColumn* містить комірки з графічними елементами (піктограмами, зображеннями тощо).

- *SpCompositeTableColumn* пропонує можливість об'єднати стовпці різних типів у один. Наприклад, це дає змогу поєднати значок (*SpImageTableColumn*) з назвою (*SpStringTableColumn*).

8.12. Перша таблиця

Фрагмент коду показує, як визначити просту таблицю з двома стовпцями, як на рис. 8.11. Повідомлення *showColumnHeaders* велить демонстратору відображати заголовки стовпців.

```
SpTablePresenter new
   addColumn: (SpStringTableColumn title: 'Number' evaluated: #yourself);
   addColumn: (SpStringTableColumn title: 'Hex' evaluated: #hex);
   showColumnHeaders;
   items: (10 to: 21);
   open;
   withWindowDo: [ :window | window title: 'Integers' ]
```

Додайте до попередньої таблиці *SpIndexTableColumn title: 'My index'*, щоб побачити стовпець індексів у дії.

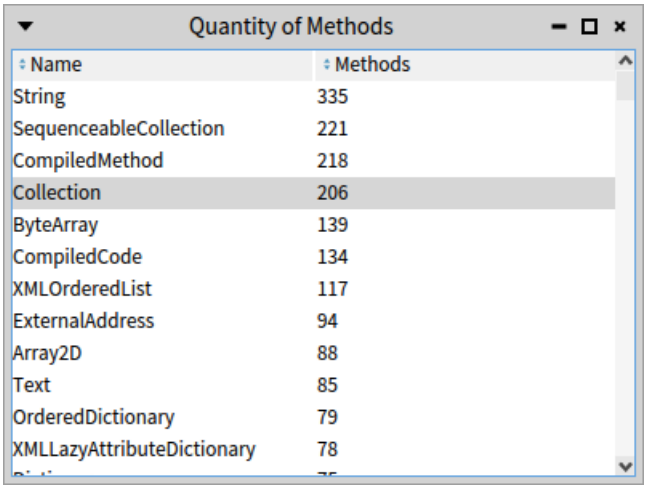


Рис. 8.12. Таблиця, рядки якої можна сортувати за кожним з двох стовпців

8.13. Заголовки, здатні сортувати

Приклад коду демонструє, як визначити таблицю з двома стовпцями, за якими можна сортувати вміст таблиці. На рис. 8.12 зображено результат після сортування другого стовпця за спаданням.

```
| classNameCompare methodCountSorter |

classNameCompare := [ :c1 :c2 | c1 name < c2 name ].
methodCountSorter := [ :c1 :c2 |
   c1 methodDictionary size threeWayCompareTo: c2 methodDictionary size
   ].
SpTablePresenter new
   addColumn: ((SpStringTableColumn title: 'Name'
         evaluated: #name)
         compareFunction: classNameCompare);
```

```
    addColumn: ((SpStringTableColumn title: 'Methods'
          evaluated: [ :c | c methodDictionary size ])
          sortFunction: methodCountSorter);
    items: Collection withAllSubclasses;
    open;
    withWindowDo: [ :window | window title: 'Quantity of Methods' ]
```

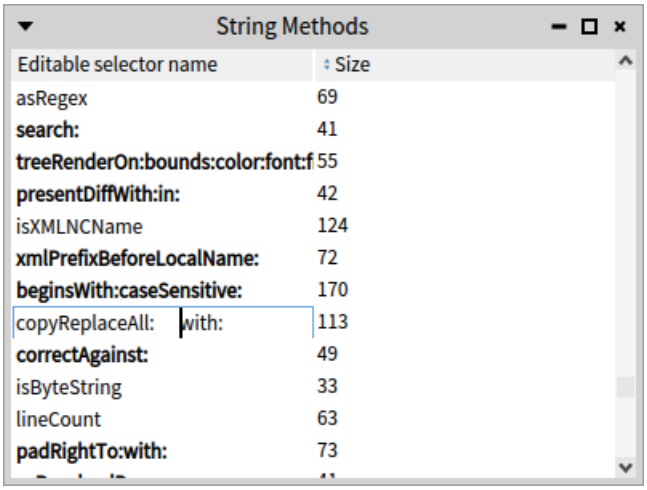


Рис. 8.13. Таблиця з редагованим стовпцем

8.14. Редаговані таблиці

Комірки таблиці можна редагувати, якщо надіслати стовпцю повідомлення *beEditable* й *onAcceptEdition:*[6]. Наведений нижче код будує таблицю, в якій можна редагувати вміст комірок першого стовпця. Отриману таблицю зображено на рис. 8.13.

```
| items |
items := String methods.
SpTablePresenter new
   addColumn:
      (SpStringTableColumn new
         title: 'Editable selector name';
         evaluated: [ :m | m selector ];
         displayBold: [ :m | m selector isKeyword ];
         beEditable;
         onAcceptEdition: [ :m :t |
            Transcript nextPutAll: t;
               cr; endEntry ];
         yourself);
   addColumn:
      (SpStringTableColumn title: 'Size' evaluated: #size)
         beSortable;
         showColumnHeaders;
         items: items;
   open;
   withWindowDo: [ :window | window title: 'String Methods' ]
```

[6] Щоб підтвердити внесені в комірку зміни, потрібно натиснути клавішу [*Enter*] на основній клавіатурі (прим. – *Ярошко С.*).

8.15. Ієрархічні таблиці

Spec пропонує спосіб створення дерева з додатковими стовпцями. Клас *SpTreeTable-Presenter* інкапсулює цю поведінку. Зверніть увагу на те, що перший стовпець у такій таблиці трактується як дерево.

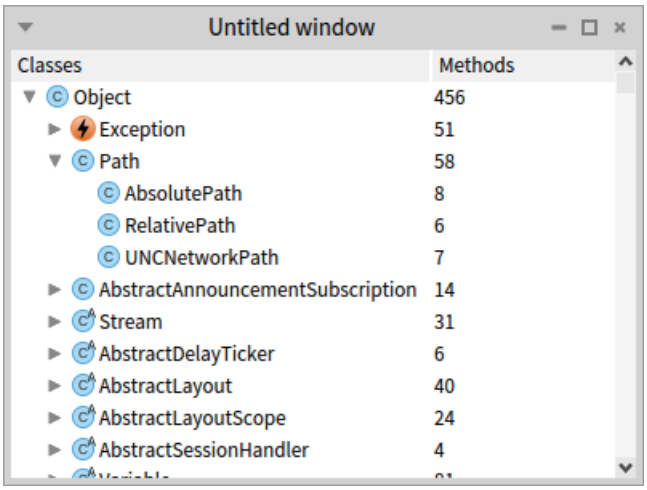


Рис. 8.14. Ієрархічна таблиця з двома стовпцями: перший складено з піктограми та рядка

У фрагменті коду нижче збудовано ієрархічну таблицю. Її перший стовпець *SpCom-positeTableColumn* – дерево, вузли якого складаються з піктограми та імені. Можна помітити, що стовпці до ієрархічної таблиці додають, як до звичайної, але джерело даних задають як для дерева. На рис. 8.14 зображено вікно після розгортання вузлів *Object* і *Path*. Перший стовпець розширили за допомогою перетягування мишкою.

```
SpTreeTablePresenter new
   beResizable;
   addColumn:
      (SpCompositeTableColumn new
         title: 'Classes';
         addColumn:
            (SpImageTableColumn evaluated: [ :aClass |
               self iconNamed: aClass systemIconName ]);
         addColumn:
            (SpStringTableColumn evaluated: [ :each | each name ] );
         yourself);
   addColumn:
      (SpStringTableColumn new
         title: 'Methods';
         evaluated: [ :class | class methodDictionary size asString ]);
   roots: { Object };
   children: [ :aClass | aClass subclasses ];
   open
```

Після отримання таблицею повідомлення *beResizable* межі між її стовпцями стають чутливими до курсора мишки, і користувач може змінювати ширину стовпців на власний розсуд. Додамо також, що за допомогою повідомлення *width:* можна задати початкову ширину будь-якого стовпця таблиці.

Ви можете спробувати не дуже мудрий приклад, у якому для побудови дерева використано невідповідні дані. Результат зображено на рис. 8.15.

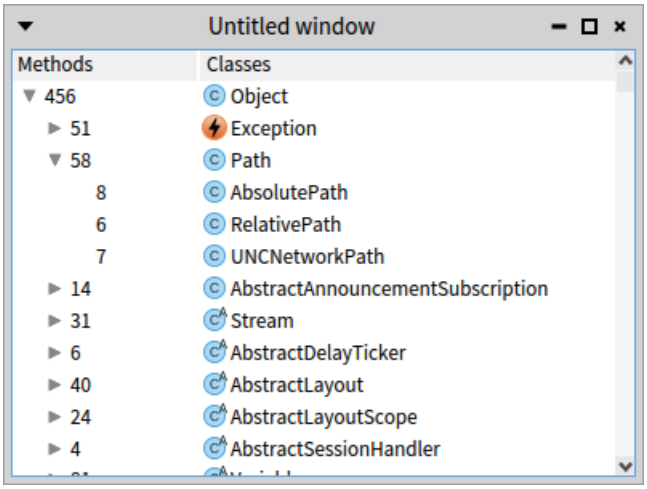


Рис. 8.15. Ієрархічна таблиця з дещо дивним деревом

```
| compositeColumn |
compositeColumn := SpCompositeTableColumn new title: 'Classes';
   addColumn: (SpImageTableColumn evaluated: [ :aClass |
         self iconNamed: aClass systemIconName ]);
   addColumn: (SpStringTableColumn evaluated: #name);
   width: 250; beExpandable;
   yourself.
SpTreeTablePresenter new
   beResizable;
   addColumn: (SpStringTableColumn new
         title: 'Methods'; width: 100;
         evaluated: [ :class | class methodDictionary size asString ]);
   addColumn: compositeColumn;
   roots: { Object };
   children: [ :aClass | aClass subclasses ];   open
```

8.16. Підсумки розділу

У цьому розділі представлено важливі контейнери: демонстратори списків, дерев і таблиць.

Від перекладача. Доповнимо скромний підсумок авторів книги, адже розділ містить багато цікавого та корисного матеріалу. Список відображає текстовий перелік колекції об'єктів, причому спосіб відображення можна налаштовувати. Він може фільтрувати, впорядковувати свої елементи, відображати їх різними шрифтами, додавати перемикачі вибору, виконувати задану дію у відповідь на подвійне клацання, підтримувати одиничний або множинний вибір. Джерело даних списку задають повідомленням *items:*. До списку можна додавати заголовок. Дерева добре підходять для відображення ієрархічних структур. Спосіб відображення вузла налаштовують за допомогою унарного блока. Вузли дерева можна оздоблювати значками, оснащувати контекстним меню. Доступні програмні методи прокручування та розгортання дерева. Таблиця складається з набору стовпців, можливо, різнотипних, кожен з яких є об'єктом з власним функціоналом. Стовпець має заголовок, може сортувати свої значення (а разом з тим, і рядки таблиці). Джерело даних таблиці, як і списку, задають повідомленням *items:*, тому таблиця підходить для відображення колекції структурованих значень.

Розділ 9
Керування вікнами

На цей момент уже описане повторне використання демонстраторів, підкласів *SpPresenter*, обговорені основи функціонування Spec і з'ясовано, як макетувати інтерфейс користувача. Але для повноцінного функціонування інтерфейсу користувача все ще бракує того, як відобразити всі ці компоненти всередині вікна. Дотепер у прикладах було показано лише кілька можливостей Spec щодо керування вікнами, здебільшого йшлося про відкривання вікна.

Цей розділ пропонує повніший огляд засобів, які надає Spec для керування вікнами. Буде продемонстрано способи відкривання та закривання вікон, вбудовані можливості створення панелей діалогу, встановлення розмірів і всі види оздоблення вікон.

9.1. Робочий приклад

Щоб проілюструвати доступні параметри конфігурації вікна, використаємо простий клас *WindowExamplePresenter*, який містить дві розташовані поруч кнопки. Вони ще не викликають ніяких дій. Їх буде додано трохи згодом у цьому розділі.

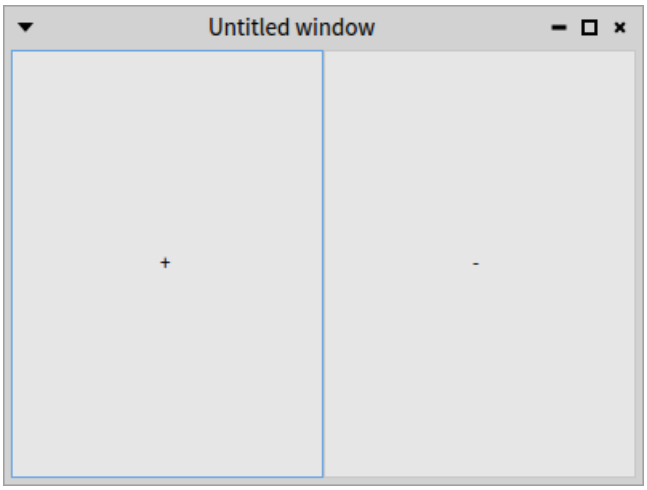


Рис. 9.1. Дуже просте вікно на основі *WindowExamplePresenter*

```
SpPresenter << #WindowExamplePresenter
   slots: { #minusButton . #plusButton };
   package: 'CodeOfSpec20Book'

WindowExamplePresenter >> initializePresenters
   plusButton := self newButton.
   minusButton := self newButton.
   plusButton label: '+'.
   minusButton label: '-'

WindowExamplePresenter >> defaultLayout
   ^ SpBoxLayout newLeftToRight
      add: #plusButton; add: #minusButton;
      yourself
```

9.2. Відкривання вікна або панелі діалогу

Інтерфейс користувача можна відкрити як звичайне, або як діалогове вікно, тобто без кнопок керування вікном, але з кнопками ***Ok*** і ***Cancel***. Продемонструємо, як це робиться, включно з параметрами налаштування, характерними для діалогових вікон. Для отримання додаткової інформації про оформлення вікон перегляньте параграф 9.5.

Відкривання вікна

Як було з'ясовано в попередніх розділах, щоб відкрити інтерфейс користувача, потрібно створити екземпляр його класу демонстратора та надіслати йому повідомлення *open*. Як наслідок буде створено екземпляр *SpWindowPresenter*, який вказуватиме на вікно з цим інтерфейсом і відобразить його на екрані.

Раніше також використовували повідомлення *openWithLayout:*, аргументом якого є макет, екземпляр підкласу *SpLayout*. Відкритий так інтерфейс користувача замість усталеного макета використовуватиме переданий у повідомленні.

Нижче показано два способи відкривання вікна для *WindowExamplePresenter*. Фрагмент коду відкриває два однакові вікна як те, що зображене на рис. 9.1.

```
| presenter |
presenter := WindowExamplePresenter new.
presenter open.
presenter openWithLayout: presenter defaultLayout
```

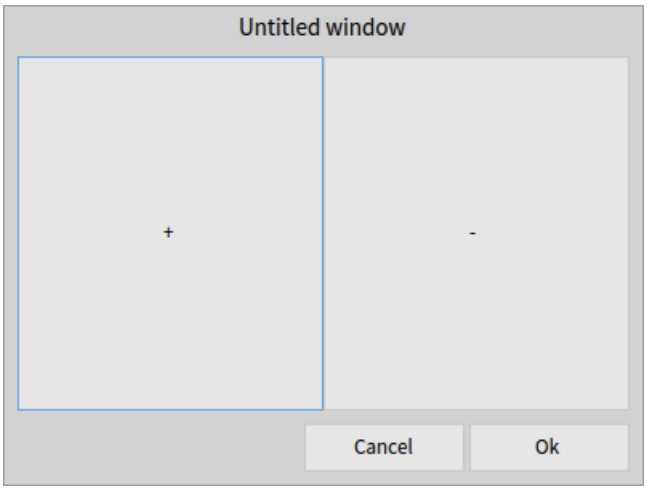


Рис. 9.2. Дуже простий діалог на основі *WindowExamplePresenter*

Відкривання панелі діалогу

Spec надає простий спосіб відкрити інтерфейс користувача у вигляді панелі діалогу з кнопками ***Ok*** і ***Cancel***. Діалогове вікно не має кнопок для зміни розміру, закривання та віконного меню. Щоб відкрити панель діалогу, надішліть повідомлення *openDialog*.

```
| presenter dialog |
presenter := WindowExamplePresenter new.
dialog := presenter openDialog
```

У результаті виконання методу *openDialog* буде отримано екземпляр класу *SpDialog-WindowPresenter* (підкласу *SpWindowPresenter*), який стане значенням змінної *dialog*. На рис. 9.2 зображено отриману панель діалогу.

Екземпляр *SpDialogWindowPresenter* можна налаштувати різними способами. Щоб кнопки діалогу виконували роботу після натискання, надішліть йому повідомлення *okAction:* або *cancelAction:* з блоком без аргументів.

```
| presenter dialog |
presenter := WindowExamplePresenter new.
dialog := presenter openDialog
   okAction: [ 'okAction' crTrace ];
   cancelAction: [ 'cancelAction' crTrace ]
```
[7]

Щоб довідатися, як було завершено діалог, надішліть екземплярові діалогу повідомлення *canceled*. Воно поверне значення *true*, якщо панель діалогу закрили кнопкою ***Cancel***.

9.3. Запобігання закриттю вікна

Spec дає змогу перевірити, чи справді можна закрити вікно, коли користувач натискає на кнопку закривання. Метод *SpWindowPresenter>>whenWillCloseDo:* приймає блок, який вирішує, чи можна закрити вікно. Можна було б так змінити *WindowExamplePresenter*:

```
WindowExamplePresenter >> initializeWindow: aWindowPresenter

   aWindowPresenter whenWillCloseDo: [ :announcement |
      announcement denyClose ]
```

Блок у прикладі має аргумент *announcement*. Він міститиме посилання на екземпляр *SpWindowWillClose*. Цей клас має два цікаві методи: *allowClose* і *denyClose*. Наведений вище фрагмент коду надсилає *denyClose* до *announcement*. Зробивши так, ми насправді створили вікно, яке неможливо закрити!

Щоб мати можливість все ж закрити вікно, потрібно змінити реалізацію зазначеного методу. За звичайних умов вікно мало б закриватися, тому блок має надсилати *denyClose* лише у тому випадку, коли вікно справді не можна закривати. Адаптуймо блок так, щоб перепитувати в користувача, чи хоче він закрити вікно.

```
WindowExamplePresenter >> initializeWindow: aWindowPresenter

   aWindowPresenter whenWillCloseDo: [ :announcement |
      (self confirm: 'Are you sure that you want to close the window?')
         ifFalse: [ announcement denyClose ] ]
```

Звичайно, наведений вище приклад методу максимально спрощений і не надто корисний[8]. Що саме перевіряти в такому методі, на практиці мало б залежати від логіки поведінки застосунку.

---

[7] Повідомлення *crTrace* виводить рядок, отримувач повідомлення, у вікно Transcript (прим. – *Ярошко С.*).
[8] З незрозумілих причин у Pharo 12, встановленому у Windows 10, цей метод перепитує користувача двічі і закриває вікно тільки після отримання другої відповіді, причому на закривання вікна впливає тільки перша (прим. – *Ярошко С.*).

9.4. Дії під час закривання вікна

Можливо також задати дію, яку буде виконано щоразу, коли вікно закривається: чи то звичайне, чи діалогове вікно.

Налаштування вікна

Якщо ви хочете отримати сповіщення про те, що вікно закрито, то треба так перевизначити метод *initializeWindow:* у класі демонстратора:

```
WindowExamplePresenter >> initializeWindow: aWindowPresenter
   aWindowPresenter whenClosedDo: [ self inform: 'When closed' ]
```

Такий фрагмент коду програмно відкриває та закриває вікно, і ви мали б побачити сповіщення, яке запускається в момент закриття.

```
| presenter window |
presenter := WindowExamplePresenter new.
window := presenter open.
window close
```

Налаштування панелі діалогу

Якщо потрібна така ж поведінка панелі діалогу, то можна або використати описаний раніше механізм (тобто визначити бажану під час закривання вікна поведінку в методі *initializeWindow:*), або налаштувати демонстратор діалогу, отриманий у результаті виконання повідомлення *openDialog*.

```
| presenter dialog |
presenter := WindowExamplePresenter new.
dialog := presenter openDialog.
dialog
   okAction: [ 'okAction' crTrace ];
   cancelAction: [ 'cancelAction' crTrace ];
   whenClosedDo: [ self inform: 'Bye bye!' ]
```

Дії з вікном

Повідомлення *withWindowDo:* гарантує, що відкритий у вікні демонстратор все ще існує, або перебуває в розумному стані.

```
presenter withWindowDo: [ :window | window title: 'MyTitle' ]
```

9.5. Розмір і оформлення вікна

Тепер зосередимося на визначенні розміру вікна до та після його відкривання, а потім опишемо видалення різних елементів керування, які належать до оформлення вікна.

Встановлення початкового розміру та його зміна

Для того, щоб відкрити вікно з певним розміром, надішліть повідомлення *initialExtent:* до відповідного *SpWindowPresenter* перед відкриванням, наприклад, так:

```
| windowPresenter |
 windowPresenter := WindowExamplePresenter new asWindow.
```

```
 windowPresenter initialExtent: 300@80.
 windowPresenter open
```

Проте загальним способом визначення початкового розміру вікна є використання повідомлення *initialExtent:* у методі *initializeWindow:*.

```
WindowExamplePresenter >> initializeWindow: aWindowPresenter
  aWindowPresenter initialExtent: 80@100
```

Зауважимо, що за допомогою повідомлення *initialPosition:* можна встановити початкове розташування вікна.

Розмір відкритого вікна також можна змінити. Для цього екземплярові вікна надсилають повідомлення *resize:*. Наприклад, можна змінити метод *initializePresenters* нашого прикладу так, щоб вікно змінювало розмір залежно від того, яку кнопку натиснуто.

```
WindowExamplePresenter >> initializePresenters
   plusButton := self newButton.
   minusButton := self newButton.
   plusButton label: '+'.
   minusButton label: '-'.
   plusButton action: [ self window resize: 500@200].
   minusButton action: [ self window resize: 200@100]
```

Задати розташування вікна застосунку стосовно світу Pharo чи інших вікон можна за допомогою повідомлень *centered*, *centeredRelativeTo:* і *centeredRelativeToTopWindow*.

Фіксований розмір

Зафіксувати розмір вікна так, щоб користувач не міг його змінити, перетягуючи сторони або кути, можна так:

```
| presenter |
presenter := WindowExamplePresenter new open.
presenter window beUnresizeable
```

Зауважимо, що те, що заборонено користувачеві, дозволено програмі: повідомлення *resize:* діятиме так само.

Вилучення оформлення вікна

Іноді має сенс мати вікно без оформлення, тобто без кнопок керування у заголовку. Наразі таке налаштування неможливо зробити з *SpWindowPresenter* вікна, але базова графічна бібліотека компонентів може це дозволити. Покажемо, як отримати *SpWindow* нашого прикладу та дати йому вказівку видалити різні кнопки керування.

```
| presenter |
presenter := WindowExamplePresenter new open.
presenter window
   removeCollapseBox;   removeExpandBox;
   removeCloseBox;      removeMenuBox
```

Зауваження. Це вікно все ще можна закрити за допомогою меню-ореола або повідомленням *close* екземплярові *SpWindowPresenter* (*presenter* у прикладі вище).

Налаштування назви у заголовку вікна

Усталена назва нового вікна «Untitled window». Звичайно, її можна змінити. Загальний спосіб полягає в тому, щоб перевизначити метод *initializeWindow:* і надіслати в його тілі повідомлення *title:* до *windowPresenter* як у прикладі.

```
WindowExamplePresenter >> initializeWindow: aWindowPresenter
   aWindowPresenter title: 'Click to grow or shrink.'
```

Крім того, можна задати заголовок будь-якого вікна після його відкриття, надіславши екземплярові вікна повідомлення *title:* з новою назвою як аргумент. Наприклад,

```
| presenter |
presenter := WindowExamplePresenter new.
presenter open.
presenter window title: 'I am different!'
```

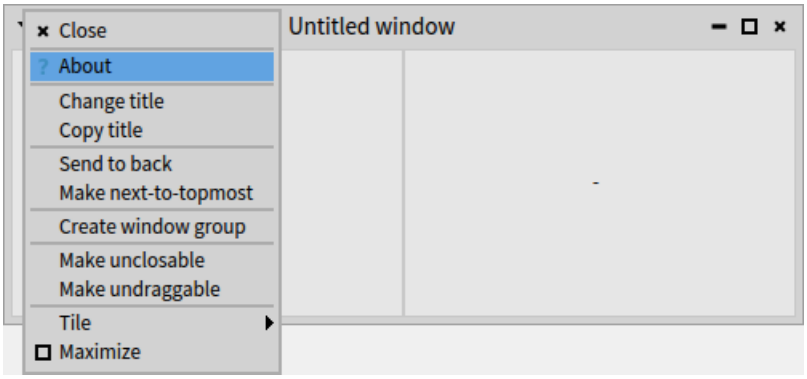


Рис. 9.3. Відкривання вікна «Про програму...»

Налаштування тексту «Про програму...»

Кожне вікно у Pharo має спадне меню вікна, заховане під трикутною стрілкою у заголовку вікна (див. рис. 9.3). Залежно від версії Pharo вона розташована в лівому або правому верхньому куті вікна. Один з пунктів цього меню, ***About***, відкриває вікно «Про програму...», який розробники застосунку використовують, щоб додати його опис і перелічити авторів.

Щоб задати текст, який відображатиметься у вікні «About», або перевизначають метод *aboutText* відповідного підкласу *SpPresenter* так, щоб він повертав бажаний текст, або надсилають повідомлення *aboutText:* екземпляру вікна перед відкриттям, наприклад, як показано нижче.

```
| windowPresenter |
 windowPresenter := WindowExamplePresenter new asWindow.
 windowPresenter aboutText: 'Click + to grow, - to shrink.'.
 windowPresenter open
```

Такий код відкриє вікно застосунку *WindowExamplePresenter*. Якщо вибрати пункт ***About*** у його меню, то відкриється вікно «Про програму...» з налаштованим текстом, як зображено на рис. 9.4.

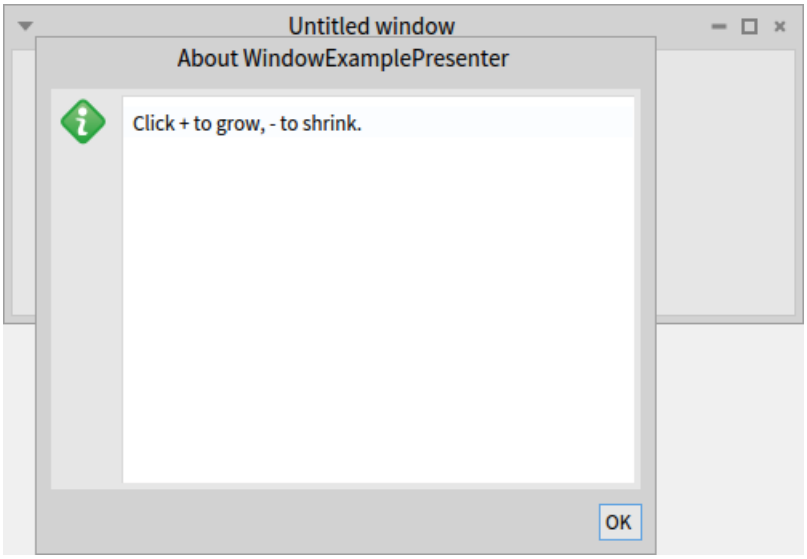


Рис. 9.4. Вікно «Про програму...»

9.6. Отримання значень з діалогу

Виконаний демонстратором метод *openDialog* повертає панель діалогу, щоб можна було легко надіслати їй повідомлення *isOk*. Коли *isOk* відповідає *true*, панель діалогу готова надати введені користувачем дані.

Розглянемо приклад: відкриємо діалог для вибору кольорів.

Налаштування інтерфейсу користувача становить більшу частину коду нижче, а цікава частина розташована наприкінці. Стандартний стан діалогу – «Скасовано», тому треба повідомити діалогове вікно, коли воно переходить у стан «Схвалено». Зробимо це в блоці повідомлення *okAction:*, надіславши діалоговому вікну повідомлення *beOk*.

Потім у блоці *whenClosedDo:* надішлемо вікну повідомлення *isOk*. Якщо воно поверне *true*, то має сенс опрацювати вибрані кольори. Для спрощення прикладу просто відкриємо множину вибраних кольорів в інспекторі.

```
| selectedColors presenter colorTable dialogPresenter |

selectedColors := Set new.
presenter := SpPresenter new.
colorTable := presenter newTable
   items: (Color red wheel: 10);
   addColumn: (SpCheckBoxTableColumn new
      evaluated: [ :color | selectedColors includes: color ];
      onActivation: [ :color | selectedColors add: color];
      onDeactivation: [ :color | selectedColors remove: color];
      width: 20;
      yourself);
   addColumn: (SpStringTableColumn new
      evaluated: [ :color | '' ];
      displayBackgroundColor: [ :color | color ];
      yourself);
   hideColumnHeaders;
   yourself.
```

```
presenter layout: (SpBoxLayout newTopToBottom
   add: colorTable;
   yourself).
dialogPresenter := presenter openDialog.
dialogPresenter
   title: 'Select colors';
   okAction: [ :dialog | dialog beOk ];
   whenClosedDo: [ dialogPresenter isOk
      ifTrue: [ selectedColors inspect ] ]
```

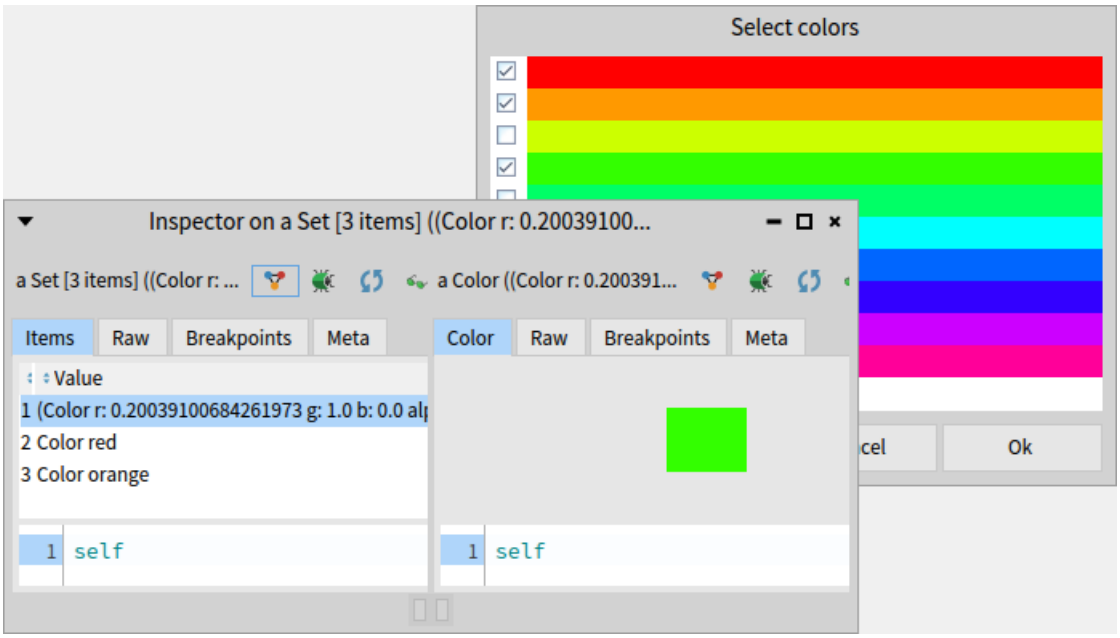


Рис. 9.5. Діалог вибору та вибрані кольори в інспекторі

На рис. 9.5 зображено результат виконання коду[9], колаж з двох вікон: діалогу й інспектора. Зрозуміло, що інспектор з'явиться після того, як користувач завершить діалог натисканням кнопки ***Ok***. Видно, що користувач обрав три кольори.

9.7. Стандартні демонстратори модальних діалогів

Модальне діалогове вікно забирає контроль над усім інтерфейсом користувача Pharo. Поки воно відкрите, неможливо вибрати інше вікно.

Spec постачає кілька невеликих попередньо визначених діалогів для інформування користувачів, або отримання інформації від них. Більшість із них успадковано від *SpDialogPresenter*. Для свого налаштування вони надають API будівельника.

Найпростішим діалогом є сповіщення.

```
SpAlertDialog new
   title: 'Inform example';
   label: 'You are seeing an inform dialog!';
   acceptLabel: 'Close this!';
   openModal
```

[9] Рисунок додав перекладач книги (прим. – *Ярошко С.*).

Діалоги підтвердження створюють так:

```
SpConfirmDialog new
   title: 'Confirm example';
   label: 'Are you sure?';
   acceptLabel: 'Sure!';
   cancelLabel: 'No, forget it';
   onAccept: [ :dialog| dialog alert: 'Yes!' ];
   onCancel: [ :dialog| dialog alert: 'No!' ];
   openModal
```

Характерним для Pharo способом використання таких діалогів є доступ до них через застосунок класу презентатора (в тілі методу):

```
self application newAlert
   title: 'Inform example';
   label: 'You are seeing an inform dialog!';
   acceptLabel: 'Close this!';
   openModal
```

Клас *SpApplication* надає такі API методи: *newConfirm*, *newAlert*, *newJobList*, *newRequest*, *newSelect*, *newRequestText*.

9.8. Розміщення демонстратора у вікні діалогу

Будь-який демонстратор можна розмістити в діалоговому вікні, перевизначивши метод *SpAbstractPresenter>>initializeDialogWindow:*, подібно до такого прикладу.

```
WindowExamplePresenter >> initializeDialogWindow: aDialogWindowPresenter
"
Used to initialize the model in the case of the use into a dialog window.
Override this to set buttons other than the default (Ok, Cancel).       "

   aDialogWindowPresenter
      addButton: 'Cancel' do: [ :presenter |
         presenter triggerCancelAction.
         presenter close ];
      addDefaultButton: 'Ok' do: [ :presenter |
         presenter triggerOkAction.
         presenter close ]
```

Перевизначте цей метод, щоб задати поведінку демонстратора на той випадок, коли його відкриють у діалоговому вікні.

9.9. Налаштування фокусу введення клавіатури

Деякі візуальні компоненти можуть приймати фокус уведення клавіатури. Найперше спадають на гадку компоненти для редагування тексту, але списки також можуть приймати фокус клавіатури. Кнопки теж. Власне кажучи, якщо доповідач відповідає на події клавіатури, то він може прийняти фокус уведення клавіатури.

Ознакою того, що компонент має фокус клавіатури, здебільшого є світло-блакитна рамка, зображена навколо нього. На рис. 9.1 показано, що кнопка «Плюс» має фокус

клавіатури. Візуальний компонент отримує фокус клавіатури, коли користувач клацає його мишкою, або натискає клавішу табуляції.

Користувач переміщує фокус клавіатури натисканням клавіші табуляції від компонента до компонента відповідно до порядку переходів фокусу. Якщо натиснути комбінацію [*Shift*+*Tab*], то фокус переміститься назад відповідно до порядку переходів. Як усталено, порядок переходів такий самий, як порядок додавання компонентів до демонстратора. Іноді цей порядок не бажаний. Тоді його треба налаштувати явно. Можна зробити це в методі *initializePresenters* демонстратора: задати порядок за допомогою повідомлення *focusOrder:*, або за допомогою *add:* додати піддемонстратори до об'єкта, який повернув метод *focusOrder*. Спробуймо зробити так у *WindowExamplePresenter*.

```
WindowExamplePresenter >> initializePresenters
   plusButton := self newButton.
   minusButton := self newButton.
   plusButton label: '+'.
   minusButton label: '-'.
   self focusOrder
      add: minusButton;
      add: plusButton
```

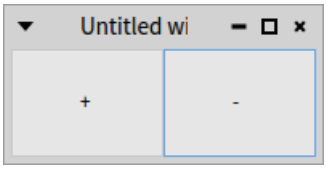


Рис. 9.6. Зворотний порядок переходів фокусу

На рис. 9.6 зображено результат після відкриття вікна. Кнопка «Мінус» має фокус клавіатури.

9.10. Дії під час відкривання вікна

Значення деяких параметрів демонстраторів або їхніх піддемонстраторів можна задати лише після відкриття вікна. Таке трапляється тоді, коли налаштування стану делеговано вкладеним компонентам. Вони доступні лише тоді, коли вікно відкрите. У розділі 13 описано, що призначити елементам меню гарячі клавіші можна тільки після відкриття вікна. Тут описано інший випадок, що відповідає попередньому параграфу.

Аби встановити порядок переходів фокуса клавіатури не потрібно, щоб вікно було відкрите, але встановити початковий фокус уведення на певному демонстраторі можна тільки у відкритому вікні. Налаштування початкового фокуса потрібне, якщо стандартний порядок переходів не підходить. Зазвичай так буває під час використання вкладених демонстраторів, порядок переходів фокуса для яких визначають явно чи неявно.

Щоб продемонструвати такий підхід, повернемо метод *initializePresenters* класу *WindowExamplePresenter* до попереднього стану та адаптуємо *initializeWindow:*.

```
WindowExamplePresenter >> initializePresenters
   plusButton := self newButton.
   minusButton := self newButton.
   plusButton label: '+'.
   minusButton label: '-'
```

Щоб встановити початковий фокус клавіатури на кнопці «Мінус», надсилаємо *takeKeyboardFocus* демонстратору кнопки в блоці, аргументі повідомлення *whenOpenedDo:*. Цей блок буде виконано відразу після відкриття вікна.

```
WindowExamplePresenter >> initializeWindow: aWindowPresenter

   super initializeWindow: aWindowPresenter.
   aWindowPresenter whenOpenedDo: [
      minusButton takeKeyboardFocus ]
```

Після відкриття вікна побачимо фокус клавіатури на кнопці «Мінус», як на рис. 9.6.

Можемо зробити крок далі. Під час відкриття екземпляра *WindowExamplePresenter* у діалоговому вікні за допомогою коду *WindowExamplePresenter new openDialog* кнопка «Плюс» отримує фокус уведення, бо вона перша в усталеному порядку переходів фокусу (див. рис. 9.2).

Початковий фокус клавіатури на кнопці «Плюс» у діалоговому вікні може бути небажаним. Ймовірно, розумніше було б розмістити фокус клавіатури на кнопці ***Ok*** панелі діалогу, щоб користувач міг одразу схвалити свій вибір, натиснувши клавішу [*Enter*] або [*Space*], якщо жодна інша взаємодія з діалогом не потрібна. Зробімо це. Замість того, щоб змінювати метод *initializeWindow:*, змінимо *initializeDialogWindow:*.

```
WindowExamplePresenter >> initializeDialogWindow: aDialogWindowPresenter

   super initializeDialogWindow: aDialogWindowPresenter.
   aDialogWindowPresenter
      whenOpenedDo: [
         aDialogWindowPresenter defaultButton takeKeyboardFocus ]
```

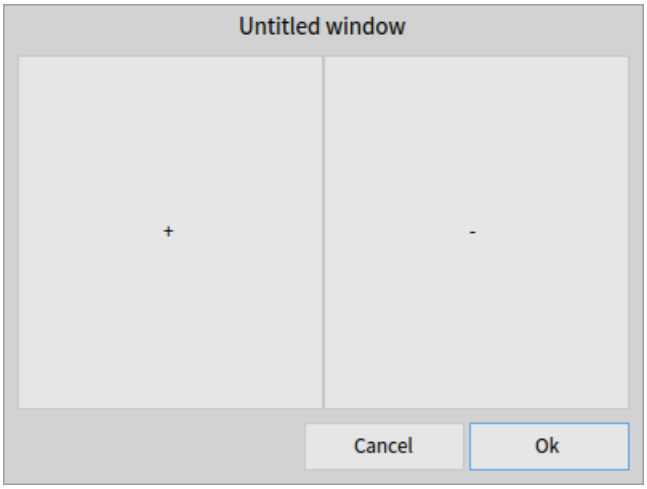


Рис. 9.7. Фокус уведення на кнопці ***Ok*** діалогу

Параметр *aDialogWindowPresenter* посилається на екземпляр *SpDialogWindowPresenter*, який у відповідь на повідомлення *defaultButton* повертає кнопку ***Ok***. Їй ми надсилаємо повідомлення *takeKeyboardFocus*. Після відкриття панелі діалогу за допомогою коду *WindowExamplePresenter new openDialog* побачимо вікно, як на рис. 9.7, з фокусом уведення на кнопці ***Ok***.

9.11. Оновлене API налаштування вікна[10]

У Pharo 13 реалізація та клієнти Spec отримають глибоко очищений API. Згадаємо тут найважливіші аспекти.

Методи налаштування вікна

Методи перехоплення *windowTitle* і *windowIcon* є публічним API для визначення назви та піктограми вікна. Метод *windowIcon* може делегувати керування значками застосункові. Справді, піктограми більше стосуються інтерфейсу користувача, ніж логіки програми.

```
MyPresenter >> windowIcon
   ^ self iconNamed: #testRunner
```

Метод *windowTitle* можна визначити зовсім просто.

```
MyPresenter >> windowTitle
   ^ 'My Presenter'
```

Якщо ваш демонстратор використовує модель, то *windowTitle* – це зазвичай те місце, де назву можна запитувати в моделі.

```
MyPresenter >> windowTitle
   ^ self model informationString
```

Використання цих двох методів зменшує складність *initializeWindow:*, часто на стільки, що він більше не потрібний.

9.12. Запам’ятовування зміненого розміру вікна

Починаючи з Pharo 13, Spec забезпечує кращий досвід користувача. Він дає змогу запам'ятати останній змінений розмір вікна. Щоб це працювало, користувачі Spec повинні дотримуватися певного шаблону.

- Найперше у визначенні методу *initializeWindow:* потрібно викликати метод надкласу. Це нове правило підтримки для розробників.

```
MyPresenter >> initializeWindow: aWindowPresenter
   super initializeWindow: aWindowPresenter.
   ...
```

- Потім розробник повинен визначити метод класу *defaultPreferredExtent*

```
MyPresenter class >> defaultPreferredExtent
   ^ 500@800
```

- Далі, метод *preferredExtent* не можна перевизначати.
- І врешті, не встановлюйте *initialExtent* демонстратора в методі *initializeWindow*, натомість використовуйте метод *defaultPreferredExtent*.

[10] Параграфи 9.11, 9.12 з нової версії книги додав перекладач (прим. – *Ярошко С.*).

9.13. Підсумки розділу

У цьому розділі описано засоби Spec для налаштування вікон. Спочатку розглянули відкривання та закривання вікон, а також те, як відкрити демонстратор у формі діалогу. Після цього було налаштовано розмір вікна та його оформлення. Після наголосу на таких невеликих, але важливих деталях вікна, як його назва і текст «Про програму», розділ завершився обробкою діалогових вікон і згадкою про Pharo 13.

Від перекладача. З попередніх розділів відомо, що налаштування вікна (наприклад, *title:*, *initialExtent:*) задають у методі *initializeWindow:*, а налаштування діалогу – в методі *initializeDialogWindow:*. Цей розділ поповнив наш арсенал новими засобами, зокрема щодо використання діалогів. Щоб налаштувати поведінку діалогу, використовують методи *okAction:* та *cancelAction:*. Закривання діалогу супроводжується подією, реакцію на яку можна задати повідомленням *whenClosedDo:*, наприклад, щоб отримати від діалогу відповідь користувача. Стан діалогу змінюють повідомленнями *beOk*, *beCanceled*, а перевіряють повідомленнями *canceled*, *isOk*. Для усталених цілей можна використовувати діалоги *SpAlertDialog*, *SpConfirmDialog*, *SpRequestDialog* тощо.

Відкривання та закривання вікна також супроводжується подіями. Налаштувати реакцію на них можна за допомогою *whenWillCloseDo:*, *whenClosedDo:*, *whenOpenedDo:*. Демонстратору відкритого вікна надсилають повідомлення *withWindowDo:*, *beUnresizeable*, *removeCollapseBox*, *removeExpandBox*, *removeCloseBox*, *removeMenuBox*. Усталений порядок переходу фокуса клавіатури можна змінити. Для цього використовують повідомлення *focusOrder* демонстраторові застосунку, або *takeKeyboardFocus* вкладеному демонстраторові. Рядок з довідкою про призначення застосунку повертає метод *aboutText*.

Розділ 10
Макети

У Spec макети представлені екземплярами спеціальних класів, підкласів *SpExecutable-Layout*. Класи макета кодують різні способи розташування елементів: послідовне, на панелі з розділювачем або в комірках сітки. У розділі представлено наявні макети, їхнє визначення й описано, як можна повторно використовувати макети, коли демонстратор повторно використовує інші демонстратори.

10.1. Нагадування про основний принцип

У Spec очікується, що з демонстратором пов'язані об'єкти макета, екземпляри класів макетів. Кожен демонстратор повинен описати розташування своїх вкладених демонстраторів.

На противагу Spec 1.0, де макети були визначені лише методами класу, у Spec 2.0, щоб визначити макет демонстратора, можна:

- визначити метод екземпляра *defaultLayout*;
- використати повідомлення *layout:* у методі *initializePresenters*, щоб задати екземпляр макета в поточному демонстраторі.

Повідомлення *defaultLayout* повертає екземпляр макета, а *layout:* – задає макет, наприклад, екземпляр *SpBoxLayout* або *SpPanedLayout*. Ці два методи є бажаним способом визначення макетів.

Зауважимо, що збережено можливість визначення методу-селектора класу, наприклад, *defaultLayout* для тих, хто віддає перевагу такому способу.

Таке нове влаштування макетів відображає їхню динамічну природу у Spec і той факт, що можна наповнювати їх одразу екземплярами демонстраторів, без потреби завчасу створювати піддемонстратори у змінних екземпляра, щоб потім використовувати їхні імена, як це було в Spec 1.0. Якщо ж трапиться випадок, коли буде потрібен «шаблон» макета, то, все одно, можна робити так, як колись.

10.2. Робочий приклад

Для того, щоб мати можливість експериментувати з макетами, описаними в цьому розділі, визначимо простий демонстратор і назвемо його *TwoButtons*.

```
SpPresenter << #TwoButtons
   slots: { #button1 . #button2 };
   package: 'CodeOfSpec20Book'
```

Визначимо простий метод *initializePresenters*.

```
TwoButtons >> initializePresenters
   button1 := self newButton.
   button2 := self newButton.
   button1 label: '1'.
   button2 label: '2'
```

10.3. Послідовний макет (*SpBoxLayout* і *SpBoxConstraints*)

Клас *SpBoxLayout* відображає демонстратори у послідовності блоків. Орієнтація макета може бути горизонтальною або вертикальною, а демонстратори розташовані відповідно зліва направо або зверху вниз[11]. Такий макет може складатися з інших макетів.

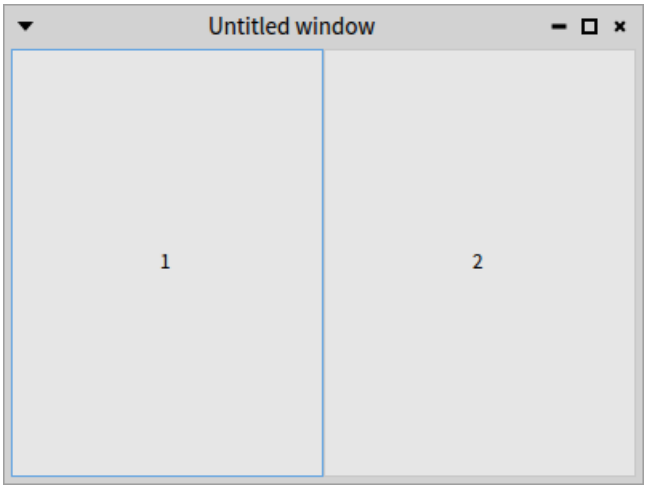


Рис. 10.1. Дві кнопки розташовані горизонтально зліва направо

Визначимо перший простий макет так, щоб збудувати вікно, як на рис. 10.1.

```
TwoButtons >> defaultLayout
   ^ SpBoxLayout newLeftToRight
      add: button1;
      add: button2;
      yourself
```

Бачимо, що за замовчуванням вкладений демонстратор розширює свій розмір так, щоб заповнити весь простір свого контейнера.

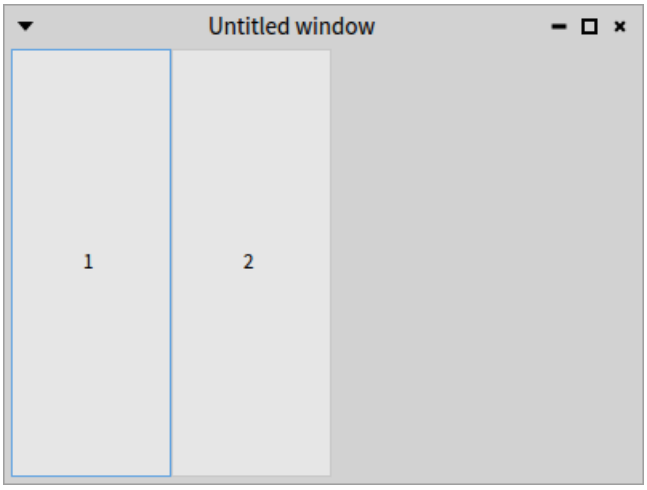


Рис. 10.2. Дві кнопки розташовані зліва направо, але не розтягнуті

[11] Схожий макет використовують у Java, аналогом у .Net MAUI є *StackLayout* (прим. – *Ярошко С.*).

У послідовному макеті з вертикальною орієнтацією елемент використовує весь доступний простір по ширині і заповнює простір по висоті відповідно до правил. У горизонтальному макеті навпаки: елемент заповнює всю висоту і частину ширини.

Можна поліпшити вигляд макета, якщо за допомогою повідомлення *add:expand:* повідомити вкладеним демонстраторам, що вони не повинні розгортатися на всю ширину свого контейнера. Результат показано на рис. 10.2.

```
TwoButtons >> defaultLayout
   ^ SpBoxLayout newLeftToRight
      add: button1 expand: false;
      add: button2 expand: false;
      yourself
```

Одночасно з додаванням демонстратора можна налаштувати й інші параметри. Повне повідомлення для цього – *add:expand:fill:padding:*.

- *expand:* – аргумент керує виділенням простору дочірньому елементу. Якщо його значення *true*, то вкладений компонент отримає в межах вікна більше місця, ніж його стандартний розмір. Додатковий простір рівномірно розподіляється між усіма дочірніми елементами, які використовують цю опцію.
- *fill:* – якщо значення аргумента *true*, то дочірній елемент так підлаштовує свої розміри, щоб зайняти весь виділений йому простір. У протилежному випадку вкладений компонент зберігає свій стандартний розмір. Цей параметр не діє, якщо *expand* має значення *false*.
- *padding:* – задає додатковий відступ у пікселях, який потрібно залишити між дочірнім елементом і його сусідами понад загальну ширину інтервалу, визначену властивістю *spacing*. Якщо компонент розташовано на одному з кінців макета, то відступ до нього роблять від краю макета. Додатковий відступ займає частину розміру візуального компонента: висоти, якщо макет вертикальний.

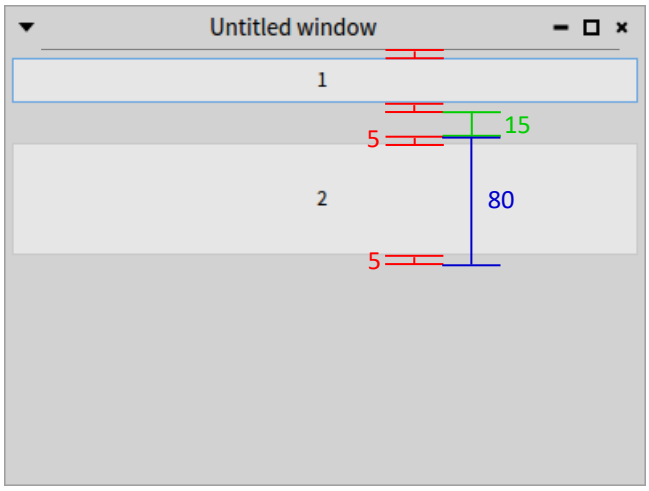


Рис. 10.3. Дві кнопки з налаштованими параметрами відступів і заповнення розташовані згори донизу

Щоб трохи проілюструвати використання цього API, змінимо метод *defaultLayout*, як написано нижче. Результат зображено на рис. 10.3. Проте хочемо наголосити, що краще не використовувати фіксовану висоту та відступи.

Позначення на рисунку зображають червоним кольором – відступи, висоту *button2* – синім і інтервал – зеленим. Зауважте, що відступи *button2* зараховано до її висоти.

```
TwoButtons >> defaultLayout
   ^ SpBoxLayout newTopToBottom
      spacing: 15;
      add: button1 expand: false fill: true padding: 5;
      add: button2 withConstraints: [ :constraints |
         constraints height: 80; padding: 5 ];
      yourself
```

У методі *defaultLayout* надіслано повідомлення *add:withConstraints:*. Воно дає змогу налаштувати параметри в тому випадку, коли часто використовувані повідомлення *add:*, *add:expand:* і *add:expand:fill:padding:* не в змозі задовольнити конкретні потреби. Аргумент *constraints* блока є екземпляром класу *SpBoxConstraints*.

10.4. Вирівнювання елементів послідовного макета

У макеті *BoxLayout* можна налаштувати вирівнювання дочірніх елементів по горизонталі та по вертикалі. Вирівнювання по горизонталі налаштовують переліченими повідомленнями, які надсилають екземплярові *SpBoxLayout*:

- *hAlignStart*
- *hAlignCenter*
- *hAlignEnd*

Вирівнювання по вертикалі задають повідомленнями

- *vAlignStart*
- *vAlignCenter*
- *vAlignEnd*

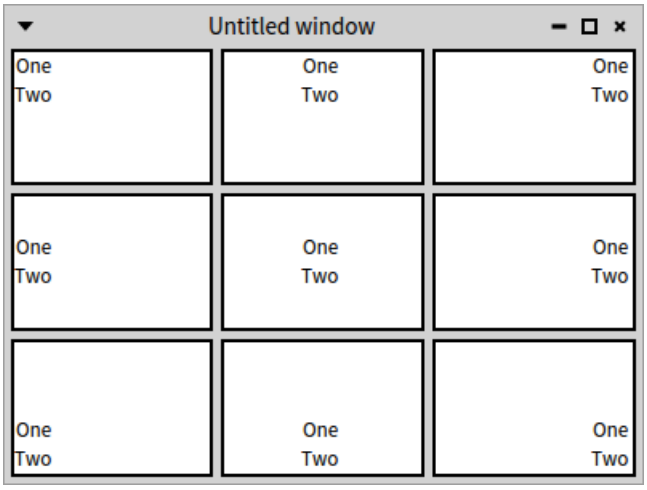


Рис. 10.4. Дев’ять плиток з різними способами вирівнювання

10.5. Приклад вирівнювання

Подивимося, як це працює, на невеликому прикладі, зображеному на рис. 10.4. Створимо демонстратор із дев’ятьма вкладеними демонстраторами, які назвемо «плитками». Розмістимо їх у три рядки по три в рядку. Кожен піддемонстратор відображає два демонстратори написів з назвами «One» і «Two». Клас презентатора визначає дев’ять змінних екземпляра. Назви засвідчують розташування вмісту всередині кожної плитки.

```
SpPresenter << #AlignmentExample
   slots: {
          #northWest . #north .  #northEast .
          #west .      #center . #east .
          #southWest . #south .  #southEast };
   package: 'CodeOfSpec20Book'
```

Як завжди, *initializePresenters* пов'язує змінні екземпляра з вкладеними демонстраторами. Для створення плиток він використовує допоміжний метод *newTile:*.

```
AlignmentExample >> initializePresenters
   northWest := self newTile: [ :tileLayout |
      tileLayout vAlignStart; hAlignStart ].
   north := self newTile: [ :tileLayout |
      tileLayout vAlignStart; hAlignCenter ].
   northEast := self newTile: [ :tileLayout |
      tileLayout vAlignStart; hAlignEnd ].
   west := self newTile: [ :tileLayout |
      tileLayout vAlignCenter; hAlignStart ].
   center := self newTile: [ :tileLayout |
      tileLayout vAlignCenter; hAlignCenter ].
   east := self newTile: [ :tileLayout |
      tileLayout vAlignCenter; hAlignEnd ].
   southWest := self newTile: [ :tileLayout |
      tileLayout vAlignEnd; hAlignStart ].
   south := self newTile: [ :tileLayout |
      tileLayout vAlignEnd; hAlignCenter ].
   southEast := self newTile: [ :tileLayout |
      tileLayout vAlignEnd; hAlignEnd ]
```

Зверніть увагу на те, що аргумент повідомлення *newTile:* – блок з параметром *titleLayout*, пов'язаним з екземпляром *SpBoxLayout*. Усередині дев'яти блоків надсилають згадані раніше повідомлення про вирівнювання, щоб налаштувати вирівнювання всередині плиток. Наприклад, до верхньої лівої плитки, названої «*northWest*», надсилають *vAlignStart* і *hAlignStart*, щоб вирівняти вміст по верхньому краю плитки та по лівому краю, відповідно.

```
AlignmentExample >> newTile: alignmentBlock
   | tileLayout |
   tileLayout := SpBoxLayout newTopToBottom
      add: self newLabelOne;
      add: self newLabelTwo;
      yourself.
   alignmentBlock value: tileLayout.
   ^ SpPresenter new
      layout: tileLayout;
      addStyle: 'tile';
      yourself
```

Метод *newTile:* використовує два інші допоміжні методи:

```
AlignmentExample >> newLabelOne
   ^ self newLabel
      label: 'One';
      yourself
```

```
AlignmentExample >> newLabelTwo
   ^ self newLabel
      label: 'Two';
      yourself
```

Макет вікна визначає такий метод.

```
AlignmentExample >> defaultLayout
   ^ SpBoxLayout newTopToBottom
      spacing: 5;
      add: (self rowWithAll: { northWest . north . northEast });
      add: (self rowWithAll: { west . center . east });
      add: (self rowWithAll: { southWest . south . southEast });
      yourself
```

Він повертає вертикальний послідовний макет з трьох рядків, заданий інтервал між якими становить 5 пікселів. У методі тричі надсилають повідомлення *rowWithAll:*, щоб створити горизонтальні послідовні макети з трьома вкладеними демонстраторами кожен. *rowWithAll:* застосовує однаковий інтервал 5 пікселів між плитками в рядку.

```
AlignmentExample >> rowWithAll: tiles
   | row |
   row := SpBoxLayout newLeftToRight
      spacing: 5;
      yourself.
   tiles do: [ :tile | row add: tile ].
   ^ row
```

Для наочності застосуємо таблицю стилів, щоб відобразити плитки з білим фоном і чорною рамкою.

```
AlignmentExample >> application
   ^ SpApplication new
      addStyleSheetFromString: '.application [
         .tile [
            Container { #borderWidth: 2, #borderColor: #black },
            Draw { #backgroundColor: #white } ]
      ]';
      yourself
```

Тепер є весь потрібний код, щоб відкрити вікно.

```
AlignmentExample new open
```

Результат показано на рис. 10.4. Кожна плитка відображає демонстратори написів у іншому місці. Демонстратори написів розташовані в стовпець.

10.6. Вирівнювання в горизонтальному макеті

Подивімося, що відбудеться, якщо помістити демонстратори написів у горизонтальний макет. Для цього достатньо змінити одне повідомлення до *SpBoxLayout* в методі *newTile:* з *newTopToBottom* на *newLeftToRight*.

```
AlignmentExample >> newTile: alignmentBlock
   | tileLayout |
   tileLayout := SpBoxLayout newLeftToRight
      add: self newLabelOne;
      add: self newLabelTwo;
      yourself.
   alignmentBlock value: tileLayout.
   ^ SpPresenter new
      layout: tileLayout;
      addStyle: 'tile';
      yourself
```

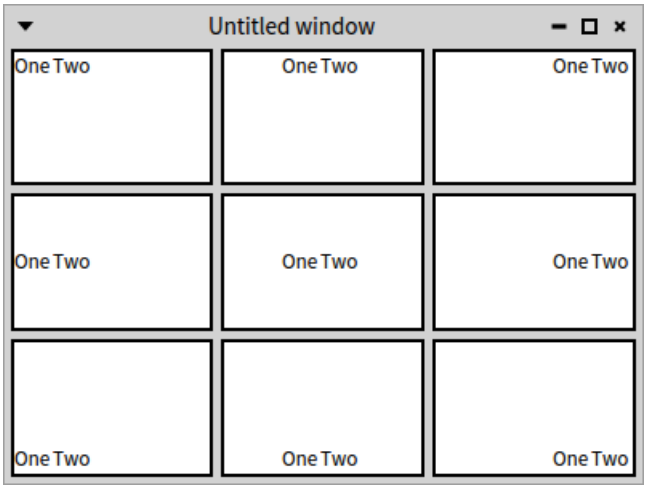


Рис. 10.5. Дев'ять плиток з написами в орієнтованих горизонтально макетах

Рис. 10.5 демонструє результат повторного відкривання вікна. Тепер написи розташовані в рядок.

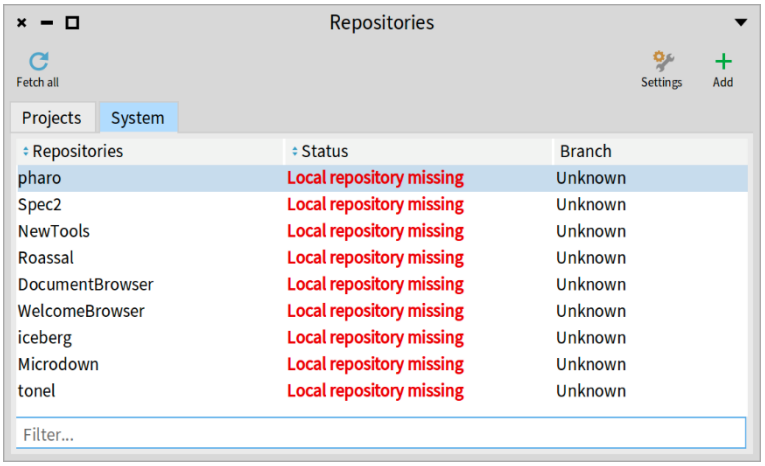


Рис. 10.6. Кнопки інструментів розташовані ліворуч і праворуч

10.7. Складніші макети

Тепер, коли відомо, як вирівнювати вкладені демонстратори, розглянемо складніший випадок. Припустимо, що потрібно розташувати три кнопки в ряд, дві з яких розташовані біля лівого краю вікна, а одна – біля правого. Таке налаштування зручне для

панелей інструментів з кнопками ліворуч і праворуч, наприклад, як в оглядачі репозиторіїв Iceberg (див. рис. 10.6). Панель має одну кнопку зліва та дві справа.

Створимо новий клас демонстратора *ButtonBar*:

```
SpPresenter << #ButtonBar
   slots: { #button1 . #button2 . #button3 };
   package: 'CodeOfSpec20Book'
```

Ініціалізуємо три кнопки:

```
ButtonBar >> initializePresenters
   button1 := self newButton.
   button2 := self newButton.
   button3 := self newButton.
   button1 label: '1'.
   button2 label: '2'.
   button3 label: '3'
```

Використаємо макет з двома вкладеними макетами: перший – для двох кнопок ліворуч, другий – для третьої кнопки праворуч. Застосуємо інтервал 15 пікселів між кнопками у першому макеті.

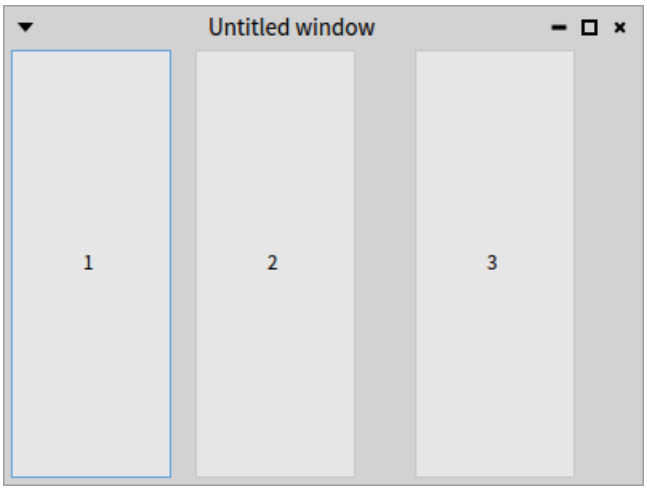


Рис. 10.7. Три кнопки, розділені на ліву та праву частини

```
ButtonBar >> defaultLayout
   | left right |
   left := SpBoxLayout newLeftToRight
      spacing: 15;
      add: button1 expand: false;
      add: button2 expand: false;
      yourself.
   right := SpBoxLayout newLeftToRight
      add: button3 expand: false;
      yourself.
   ^ SpBoxLayout newLeftToRight
      add: left;
      add: right;
      yourself
```

Якщо відкрити цей демонстратор за допомогою *ButtonBar new open*, то побачимо вікно, зображене на рис. 10.7.

Макет не зовсім такий, як хотілося. Третя кнопка розташована не біля правого краю вікна. Ось тут і з'являється вирівнювання з попереднього параграфа. Змінимо метод *defaultLayout* так, щоб вирівняти кнопку до правого краю макета правого блока. Додамо повідомлення *hAlignEnd*.

```
ButtonBar >> defaultLayout
   | left right |
   left := SpBoxLayout newLeftToRight
      spacing: 15;
      add: button1 expand: false;
      add: button2 expand: false;
      yourself.
   right := SpBoxLayout newLeftToRight
      hAlignEnd;
      add: button3 expand: false;
      yourself.
   ^ SpBoxLayout newLeftToRight
      add: left;
      add: right;
      yourself
```

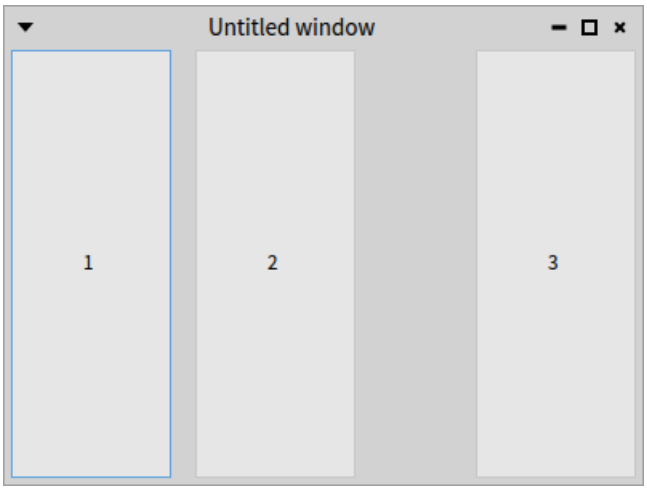


Рис. 10.8. Три кнопки, остання з яких вирівняна до правого краю

Коли знову відкриємо демонстратор, то побачимо вікно, як на рис. 10.8. Саме про такий макет і йшлося.

Цей приклад демонструє, що для досягнення бажаного результату щодо конструювання складнішого макета потрібні вкладені макети.

10.8. Приготування прикладу для повторного використання

Перш ніж представити деякі з інших макетів, розкриємо важливий аспект композиції демонстраторів у Spec: композит може оголосити, що він хоче використовувати вкладений демонстратор з певним макетом.

Розглянемо наш штучний приклад інтерфейсу користувача з двома кнопками. Використаємо у ньому два різні макети. Визначимо два методи класу, які їх повертатимуть.

Зауважте, що можна було б визначити такі методи на стороні екземпляра, але зробимо це на стороні класу, щоб мати змогу отримати макети без створення екземпляра.

```
TwoButtons class >> buttonRow
   ^ SpBoxLayout newLeftToRight
      add: #button1;
      add: #button2;
      yourself

TwoButtons class >> buttonColumn
   ^ SpBoxLayout newTopToBottom
      add: #button1;
      add: #button2;
      yourself
```

Зауважимо, що коли макет визначають методом класу, то для посилання на змінну екземпляра використовують відповідний символ. Наприклад, для посилання на демонстратор, який зберігається в змінній екземпляра *button2*, використовують символ *#button2*.

10.9. Відкривання з певним макетом

Повідомлення *openWithLayout:* дає змогу задати макет, який треба використати для відкривання демонстратора. Ось кілька прикладів:

- *TwoButtons new openWithLayout: TwoButtons buttonRow* розташовує кнопки в рядок;
- *TwoButtons new openWithLayout: TwoButtons buttonColumn* розташовує їх у стовпець.

Визначимо метод *defaultLayout* так, щоб він викликав один з визначених методів класу, і можна було відкрити демонстратор без задавання макета.

```
TwoButtons >> defaultLayout
   ^ self class buttonRow
```

10.10. Ліпша архітектура коду

Використання макетів можна організувати ліпше. Визначимо для цього два методи екземпляра: вони інкапсулюватимуть конфігурацію макета.

```
TwoButtons >> beColumn
   self layout: self class buttonColumn

TwoButtons >> beRow
   self layout: self class buttonRow
```

Тоді можна виконати такий скрипт:

```
TwoButtons new
   beColumn;
   open
```

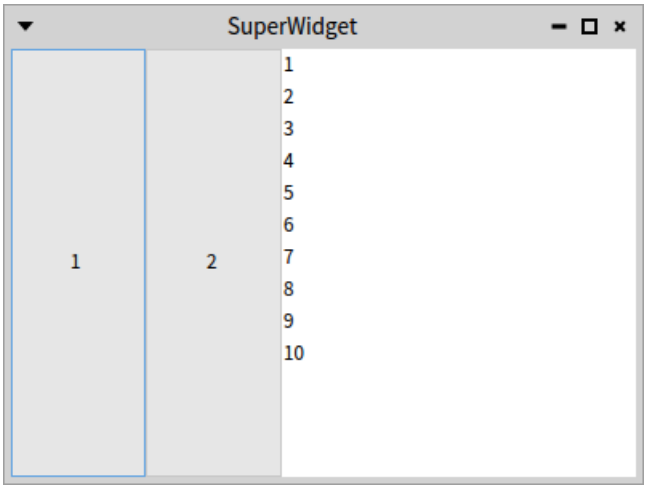


Рис. 10.9. У складеному демонстраторі кнопки розташовано в рядок

10.11. Вибір макета для демонстратора, який використовують повторно

Наявність кількох макетів для демонстратора означає, що існує спосіб вибрати макет, з яким його використати в складі іншого демонстратора. Це просто. На прикладі продемонструємо кілька способів. Створимо новий демонстратор *ButtonAndListH*.

```
SpPresenter << #ButtonAndListH
   slots: { #buttons . #list };
   package: 'CodeOfSpec20Book'

ButtonAndListH >> initializePresenters
   buttons := self instantiate: TwoButtons.
   list := self newList.
   list items: (1 to: 10)

ButtonAndListH >> initializeWindow: aWindowPresenter
   aWindowPresenter title: 'SuperWidget'

ButtonAndListH >> defaultLayout
   ^ SpBoxLayout newLeftToRight
      add: buttons;
      add: list;
      yourself
```

Клас *ButtonAndListH* створює вікно «SuperWidget» як на рис. 10.9. Він повторно використовує демонстратор *TwoButtons* і розташовує всі три компоненти в рядок, бо *TwoButtons* стандартно викликає метод макета *buttonRow*.

Тепер можна оголосити клас *ButtonAndListV*, підклас *ButtonAndListH*, і змінити лише метод *initializePresenters*, як показано нижче. Тут визначено, що вкладений компонент *buttons* використовуватиме метод макета *buttonColumn*, і отже, результатом буде вікно, як на рис. 10.10.

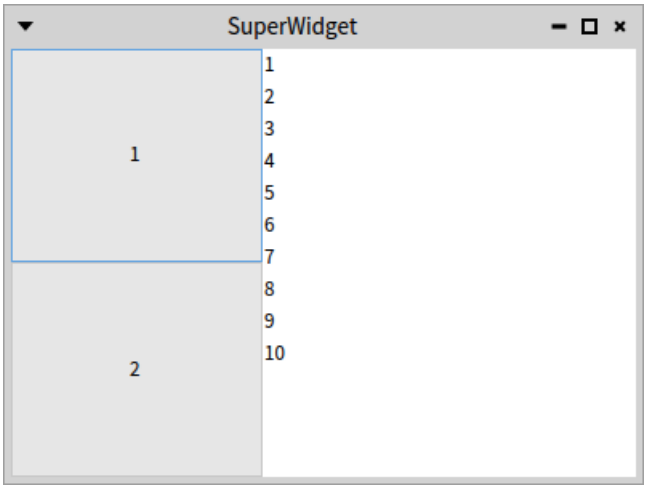


Рис. 10.10. У складеному демонстраторі кнопки розташовано в стовпець

```
ButtonAndListH << #ButtonAndListV
   slots: {};
   package: 'CodeOfSpec20Book'

ButtonAndListV >> initializePresenters
   super initializePresenters.
   buttons beColumn
```

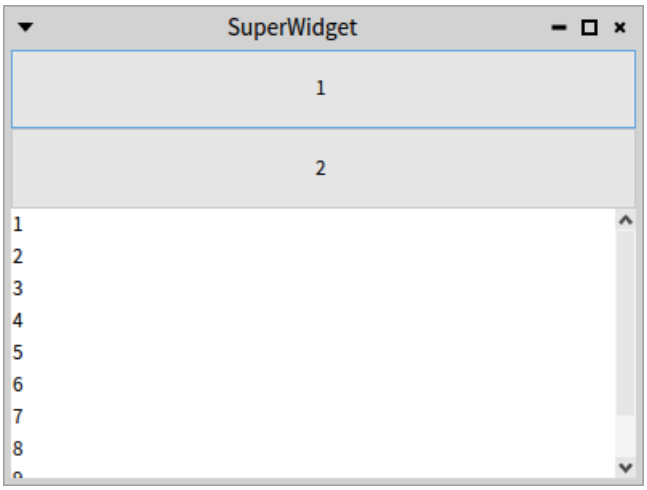


Рис. 10.11. І кнопки, і список розташовані в стовпець

10.12. Альтернативний спосіб вибору макета

Альтернативою є визначення нового методу *defaultLayout* і використання в ньому повідомлення *add:layout:*. Визначимо інший демонстратор.

```
ButtonAndListH << #ButtonAndListV2
   slots: {};
   package: 'CodeOfSpec20Book'
```

Новий метод *defaultLayout* визначимо так:

```
ButtonAndListV2 >> defaultLayout
   ^ SpBoxLayout newTopToBottom
```

```
        add: buttons layout: #buttonColumn;
        add: list;
        yourself
```

Зверніть увагу на використання повідомлення *add:layout:* із символом, селектором методу, який повертає конфігурацію макета: *#buttonColumn*. Так має бути, бо в методі *defaultLayout* ще не можна отримати доступ до стану підкомпонента. Відкриймо вікно.

```
    ButtonAndListV2 new open
```

Код відкриє вікно як на рис. 10.11.

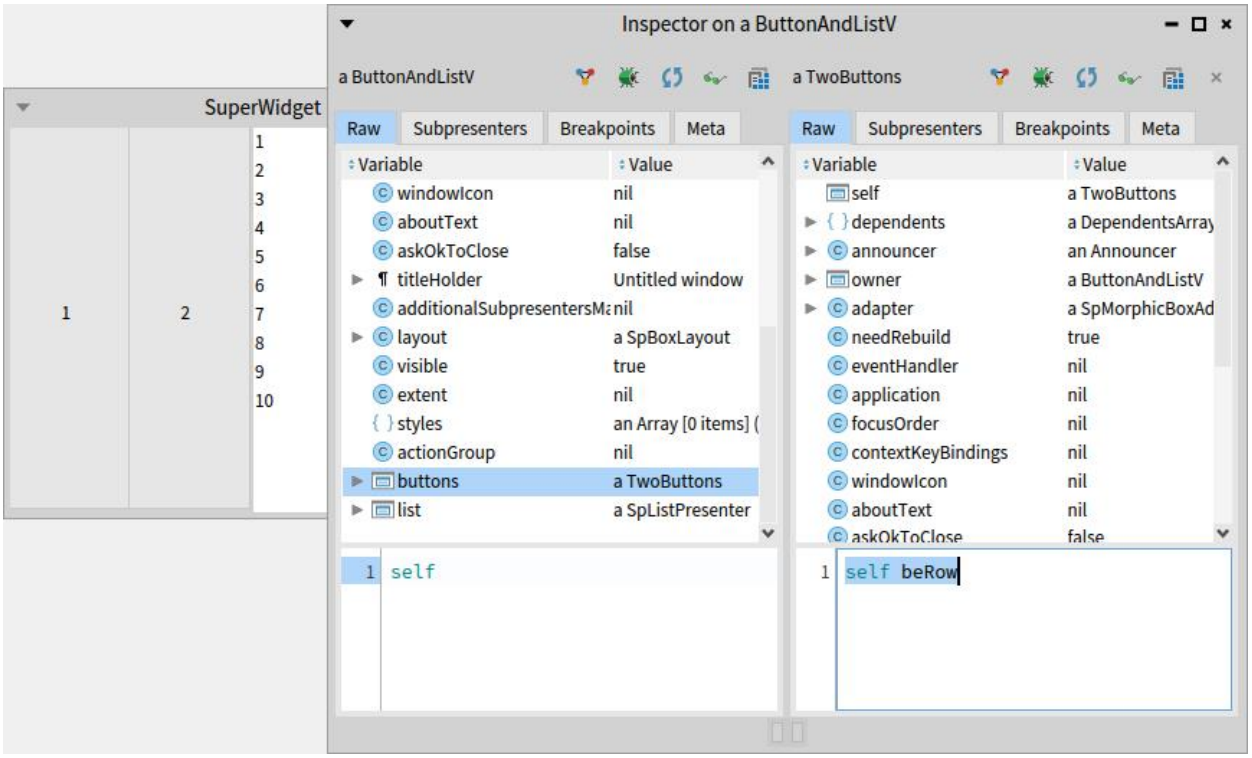


Рис. 10.12. Інтерактивна зміна макета через інспектор об’єктів

10.13. Динамічна зміна макета

Вигляд демонстратора можна змінювати динамічно, наприклад, з інспектора. Відкрийте презентатор за допомогою *ButtonAndListV new inspect open*.

Такий код відкриває інспектор на демонстраторі та вікно з кнопками, розташованими в стовпець, як зображено на рис. 10.10.

Виберіть в інспекторі змінну екземпляра *buttons* і виконайте *self beRow* у панелі редагування праворуч. Результат показано на рис. 10.12.

10.14. Прямокутна сітка (*SpGridLayout*)

Клас *SpGridLayout* розташовує демонстратори в комірках сітки відповідно до певних властивостей макета, як от:

- позиція, яку задають обов’язково (*columnNumber@rowNumber*), і
- розмір, який можна додати за бажанням (*columnExtension@rowExtension*).

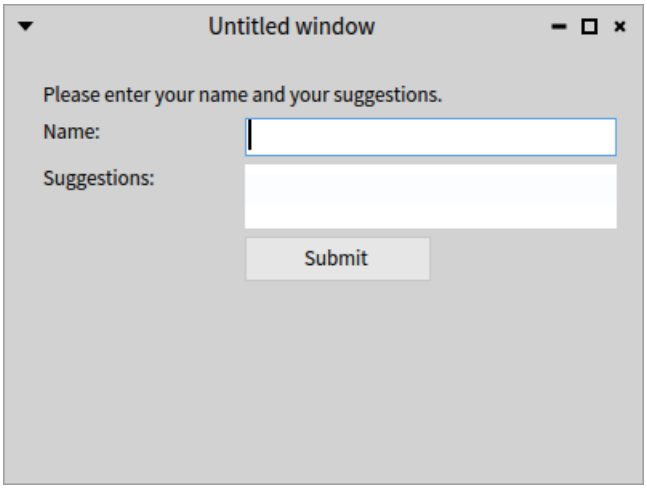


Рис. 10.13. Проста сітка для невеликої форми

Приклад демонструє побудову вікна з макетом сіткою з кількома компонентами як на рис. 10.13.

```
SpPresenter << #GridExample
   slots: { #promptLabel . #nameText . #suggestionsText .
      #submitButton };
   package: 'CodeOfSpec20Book'

GridExample >> initializePresenters
   promptLabel := self newLabel
      label: 'Please enter your name and your suggestions.';
      yourself.
   nameText := self newTextInput.
   suggestionsText := self newText.
   submitButton := self newButton
      label: 'Submit';
      yourself

GridExample >> defaultLayout
   ^ SpGridLayout new
      add: #promptLabel at: 1@1 span: 3@1;
      add: 'Name:' at: 1@2;
      add: #nameText at: 2@2 span: 2@1;
      add: 'Suggestions:' at: 1@3;
      add: #suggestionsText at: 2@3 span: 2@1;
      add: #submitButton at: 2@4 span: 1@1;
      yourself
```

Макет визначає сітку з трьома стовпцями. Підказка «*Please enter your name and your suggestions.*» охоплює три стовпці. Підписи двох полів розміщено в першому стовпці. Поля охоплюють другий і третій стовпці. У другому стовпці стоїть кнопка. Друге поле – це багаторядкова панель для редагування тексту. Тому вона вища за перше поле, призначене для введення одного рядка. Розмір сітки визначено максимальними значеннями координат комірок, зайнятих компонентами.

Ось перелік додаткових параметрів:

- *columnHomogeneous* – демонстратори у стовпці матимуть однакову ширину;
- *rowHomogeneous* – демонстратори у рядку матимуть однакову висоту;

- *colSpacing:* – задає горизонтальний інтервал між комірками;
- *rowSpacing:* – задає вертикальний інтервал між комірками.

Такий метод *defaultLayout*, як у прикладі, може бути важко зрозуміти, особливо, коли сітка містить багато демонстраторів. Читач повинен обчислити позиції та охоплення компонентів. Щоб полегшити створення сітки (і розуміння коду), можна використати *SpGridLayoutBuilder*. Цей клас не використовують безпосередньо, натомість надсилають повідомлення *build:* до *SpGridLayout*. Нижче наведено альтернативний метод *defaultLayout*, який дає той самий результат, що й раніше. Якщо помістити всі демонстратори одного ряду сітки в один рядок тексту, то стане зрозуміло, що є чотири рядки, і які компоненти є частиною одного рядка.

```
GridExample >> defaultLayout
   ^ SpGridLayout build: [ :builder |
      builder
         add: promptLabel span: 3@1; nextRow;
         add: 'Name:'; add: nameText span: 2@1; nextRow;
         add: 'Suggestions:'; add: suggestionsText span: 2@1; nextRow;
         nextColumn; add: submitButton ]
```

Від перекладача. Зверніть увагу на те, що в тілі цього методу розміщення написів у сітці виконано двома різними способами. Змінна *promptLabel* містить готовий демонстратор *SpLabelPresenter*, а інші написи додають за допомогою звичайних рядків. Наприклад, повідомлення *add: 'Name:'* і створить, і додасть демонстратор напису.

10.15. Розділений макет (*SpPanedLayout*)

Панельний (або розділений) макет схожий на послідовний, але містить тільки два дочірні елементи, які називають панелями. Він розміщує їх вертикально або горизонтально, додає роздільник між ними, який користувач може перетягувати, щоб змінювати розмір панелей. Повідомлення *positionOfSlider:* задає початкову позицію розділювача. Аргумент може мати значення *nil* (тоді буде використане усталене значення 50%), або число, яке означає відсоток: наприклад, 70% можна задати дійсним 0,7, або раціональним 7/10. Зазвичай використовують дійсні числа, тому що вони простіші та дешевші.

Розглянемо один простий приклад.

```
SpPresenter << #PanedLayoutExample
   slots: { #leftList . #rightList };
   package: 'CodeOfSpec20Book'

PanedLayoutExample >> initializePresenters
   leftList := self newList
      items: (1 to: 10);
      yourself.
   rightList := self newList
      items: ($a to: $z);
      yourself

PanedLayoutExample >> defaultLayout
   ^ SpPanedLayout newLeftToRight
      positionOfSlider: 0.7;
```

```
        add: #leftList;
        add: #rightList;
        yourself
```

Тепер відкриємо демонстратор за допомогою *PanedLayoutExample new open*.

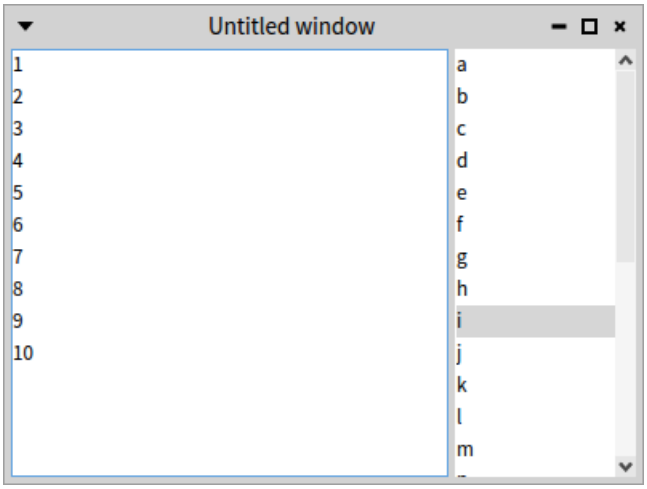


Рис. 10.14. Панельний макет з двома списками

На рис. 10.14 зображено результат. Лівий список займає 70% ширини вікна, а правий – 30%.

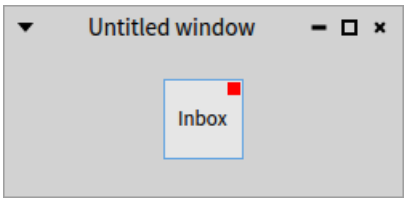


Рис. 10.15. Макет з накладанням графічного елемента на кнопку

10.16. Макет з накладанням (*SpOverlayLayout*)

Такий макет дає змогу накладати демонстратори: один поверх іншого.

Для прикладу створимо демонстратор, який містить кнопку з написом «Inbox» і накладеним поверх червоним індикатором у верхньому правому кутку (див. рис. 10.15). Такий демонстратор можна використати для сповіщення про наявність непрочитаних повідомлень у папці «Вхідні».

```
SpPresenter << #OverlayLayoutExample
   slots: { #button . #indicator };
   package: 'CodeOfSpec20Book'
```

Метод *initializePresenters* створює кнопку та індикатор, екземпляр *SpRoassalPresenter*. Для того, щоб повідомити, яку фігуру потрібно зобразити, використано допоміжний метод *indicatorShape*.

```
OverlayLayoutExample >> initializePresenters
   button := self newButton
      label: 'Inbox';
      yourself.
```

```
        indicator := (self instantiate: SpRoassalPresenter)
           script: [ :view | view addShape: self indicatorShape ];
           yourself

OverlayLayoutExample >> indicatorShape
   ^ RSBox new
         extent: 10@10;
         color: Color red;
         yourself
```

Написано три методи, щоб зробити структуру накладання зрозумілою. Перший, *defaultLayout*, визначає макет вікна. З демонстраційною метою кнопку помістили посередині вікна і задали розмір кнопки 50 на 50 пікселів.

```
OverlayLayoutExample >> defaultLayout
   | buttonVBox |
   buttonVBox := SpBoxLayout newTopToBottom
         vAlignCenter;
         add: self buttonLayout height: 50;
         yourself.
   ^ SpBoxLayout newLeftToRight
         hAlignCenter;
         add: buttonVBox width: 50;
         yourself
```

Метод *defaultLayout* надсилає повідомлення *buttonLayout*, щоб отримати макет накладання для кнопки та індикатора. Визначити метод *buttonLayout* можна так:

```
OverlayLayoutExample >> buttonLayout
   ^ SpOverlayLayout new
         child: button;
         addOverlay: self indicatorLayout
            withConstraints: [ :constraints |
               constraints vAlignStart; hAlignEnd ];
         yourself
```

У повідомленні *child:* передають демонстратор, поверх якого хочуть накласти інший. Тут передають кнопку, на яку накладають індикатор. Є можливість додати кілька накладок, але в цьому прикладі є лише одна, визначена методом *indicatorLayout*. Зауважте, що *addOverlay:withConstraints:* використовують для налаштування місця відображення накладеного демонстратора. У прикладі за допомогою повідомлень *vAlignStart* і *hAlignEnd* його буде вирівняно до верхнього правого кута.

Тепер визначимо метод *indicatorLayout* так:

```
OverlayLayoutExample >> indicatorLayout
   | counterVBox |
   counterVBox := SpBoxLayout newTopToBottom
         add: indicator withConstraints: [ :constraints |
            constraints height: 12; padding: 2 ];
         yourself.
   ^ SpBoxLayout newLeftToRight
         add: counterVBox withConstraints: [ :constraints |
            constraints width: 12; padding: 2 ];
         yourself
```

Метод *indicatorLayout* визначає макет для індикатора. Щоб застосувати вертикальні та горизонтальні відступи, доведеться огорнути вертикальний послідовний макет горизонтальним макетом. Можна було б огорнути горизонтальний макет вертикальним і досягти такого ж результату. Застосовано відступ 2 пікселі, щоб індикатор не перекривав межу кнопки.

З усіма цими методами можна відкрити демонстратор.

```
| presenter |
presenter := OverlayLayoutExample new open.
presenter window extent: 250@120.
```

Відкриється вікно, зображене на рис. 10.15.

10.17. Підсумки розділу

Spec пропонує кілька попередньо визначених макетів. Ймовірно, згодом з'являться нові, але з підтримкою сумісності. Важливим завершальним пунктом є те, що макети можна динамічно компонувати. Це означає, що можна розробляти програми, здатні адаптуватися до конкретних умов.

Від перекладача. Уважний читач міг помітити, що метод екземпляра *defaultLayout* у частині прикладів використовує для додавання до макета змінні екземпляра демонстратора, а в інших прикладах – символи, що збігаються з іменами таких змінних. Трохи дивно, що обидва варіанти працюють. Можливість використовувати символи в методах екземпляра – данина сумісності зі Spec 1.0, у якому *defaultLayout* був методом класу і не міг використовувати нічого іншого крім символів (як у методах з п. 10.8).

У Pharo 13 користувачі Spec отримають можливість використовувати ще один макет, не описаний у цій книзі – *SpFrameLayout*. Його використовують як контейнер для готового вкладеного макета. Він зображає рамку навколо дочірнього макета і може містити напис в розриві рамки у лівому верхньому куті.

Розділ 11
Динамічні демонстратори

На відміну від Spec 1.0, у Spec 2.0 усі макети динамічні. Це означає, що відображувані елементи можна змінювати на льоту. Це значне вдосконалення порівняно зі Spec 1.0, де більшість макетів були статичними, а побудова динамічних компонентів – громіздкою.

У цьому розділі описано, що за допомогою макетів демонстратори можна компонувати динамічно. З'ясуємо це у процесі невеликої інтерактивної сесії. Потім створимо невеликий браузер із динамічною складовою.

11.1. Макети такі ж прості як об'єкти

Створення динамічних програм за допомогою Spec справді просте. Адже будь-який макет у Spec динамічний і компонований. Дослідимо як це працює. Почнемо з такого фрагмента коду[12]:

```
presenter := SpPresenter new.
presenter application: SpApplication new.
```

Використаємо для цього демонстратора макет *SpPanedLayout*, який може приймати два демонстратори (або макети) і розміщувати їх в протилежних половинах вікна.

Щоб побачити всі доступні в Spec макети, можна переглянути пакет «Spec2-Layout».

```
presenter layout: SpPanedLayout newTopToBottom.
presenter open.
```

Звичайно, відкриється порожнє вікно, як на рис. 11.1, оскільки ще нічого не розмістили в макеті.

Рис. 11.1. Вікно з порожнім макетом

[12] Фрагмент коду треба виконати в Пісочниці, причому тимчасову змінну *presenter* не оголошувати, щоб вона асоціювалася з вікном Пісочниці та існувала, поки вона відкрита (прим. – *Ярошко С.*).

Тепер, не закриваючи вікно, можемо динамічно редагувати макет основного демонстратора. Додамо демонстратор кнопки за допомогою таких рядків (див. рис.11.2).

```
button1 := presenter newButton.
button1 label: 'I am a button'.
presenter layout add: button1.
```

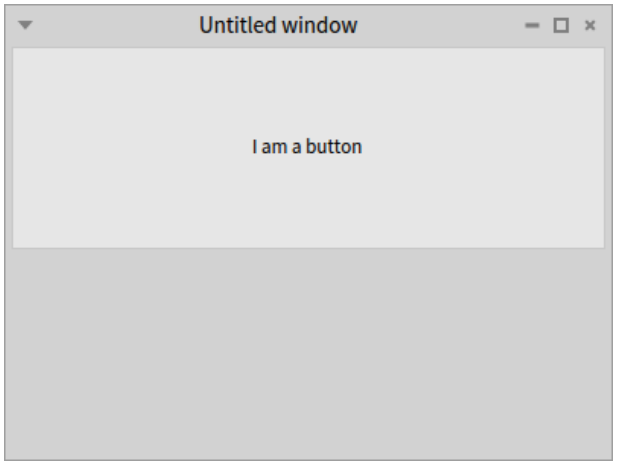


Рис. 11.2. Панельний макет з однією кнопкою

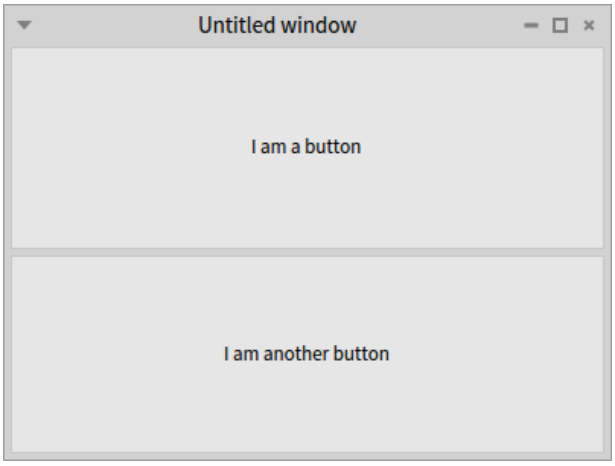


Рис. 11.3. Панельний макет з двома кнопками

Рис. 11.4. Панельний макет з двома кнопками, перша з яких зі значком

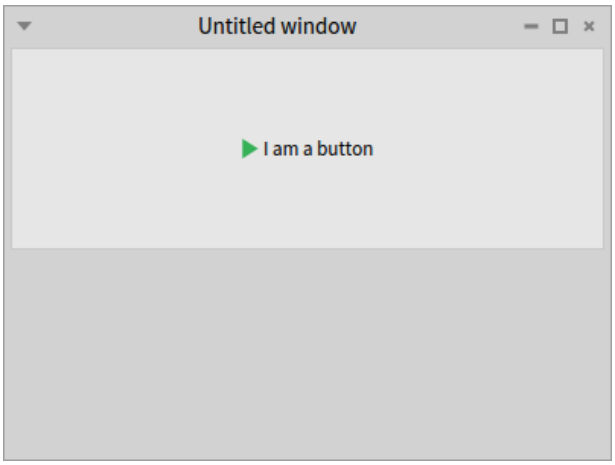


Рис. 11.5. Результат вилучення кнопки

Тепер можемо додати ще одну кнопку. Немає потреби закривати і знову відкривати вікно. Усе оновлюється динамічно й без виконання перебудови вікна. Макет створили за допомогою *newTopToBottom*, тому кнопки розташуються у стовпець (див. рис. 11.3).

```
button2 := presenter newButton.
button2 label: 'I am another button'.
presenter layout add: button2.
```

На першій кнопці можна розмістити піктограму (див. рис. 11.4).

```
button1 icon: (button1 iconNamed: #smallDoIt).
```

Або видалити з макета одну з кнопок, як на рис. 11.5.

```
presenter layout remove: button2.
```

Видно, що всі зміни відбуваються через створення нового екземпляра певного макета та надсилання йому повідомлень. Це означає, що програми можуть визначати складну логіку динамічної поведінки демонстратора.

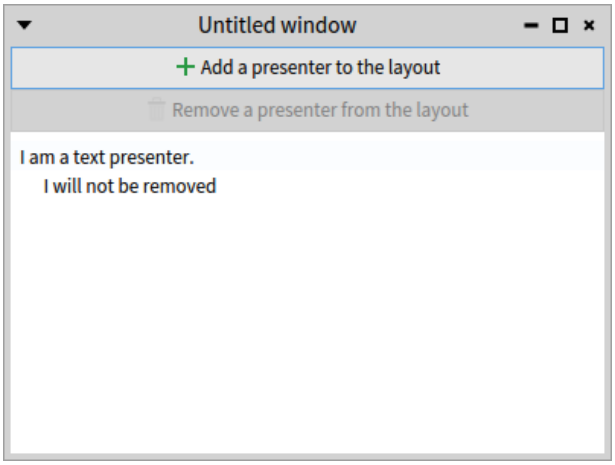


Рис. 11.6. Демонстратор, що може додавати кнопки

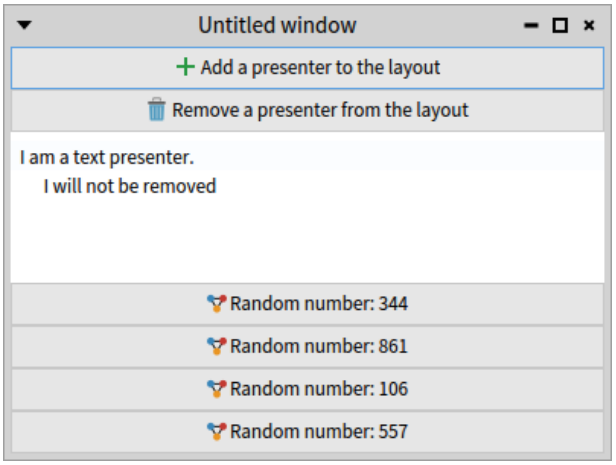


Рис. 11.7. Результат додавання випадкових кнопок

11.2. Динамічне додавання кнопок

Створимо тепер демонстратор, здатний динамічно додавати та вилучати кнопки у відповідь на натискання на відповідні кнопки. Написи на доданих кнопках міститимуть випадкові числа (див. рис. 11.6, 11.7). Почнімо! Створимо новий клас *DynamicButtons*.

```
SpPresenter << #DynamicButtons
   slots: { #addButton . #removeButton . #text };
   package: 'CodeOfSpec20Book'
```

У методі *initializePresenters* додаємо кнопку, натискання на якій додає нову кнопку до макета. Потрібна також кнопка, яка видалятиме останню додану кнопку, якщо така є. Наприкінці додаємо текстове поле лише для читання, яке ніколи не буде видалене.

```
DynamicButtons >> initializePresenters
   addButton := self newButton.
   addButton
      action: [ self addToLayout ];
      label: 'Add a presenter to the layout';
      icon: (self iconNamed: #smallAdd).

   removeButton := self newButton.
   removeButton
      action: [ self removeFromLayout ];
      label: 'Remove a presenter from the layout';
      icon: (self iconNamed: #smallDelete);
      disable.

   text := self newText.
   text
      text: 'I am a text presenter.
      I will not be removed';
      beNotEditable
```

11.3. Визначення методів додавання/вилучення кнопок

Тепер потрібно реалізувати методи *addToLayout* і *removeFromLayout*, задіяні в блоках реагування кнопок. Методи, як підказують їхні назви, динамічно додають і видаляють з макета демонстратори кнопок.

Почнімо з методу *addToLayout*. Створити та налаштувати нову кнопку допоможе тимчасова змінна *newButton*. Напис нової кнопки містить випадкове число. Після виконання *add:expand:* кнопка потрапить у колекцію дочірніх компонентів макета і зберігатиметься там. Активуємо також кнопку вилучення, щоб можна було видалити щойно додану кнопку.

```
DynamicButtons >> addToLayout
   | randomButtonName newButton |
   removeButton enable.
   randomButtonName := 'Random number: ', 1000 atRandom asString.
   newButton := self newButton
      label: randomButtonName;
      icon: (self iconNamed: #smallObjects);
      yourself.
   self layout add: newButton expand: false
```

Метод *removeFromLayout* вилучає останній елемент з колекції дочірніх компонентів макета. Щоб запобігти небажаному видаленню текстового поля, перевіряємо, чи є ще кнопка наприкінці колекції. Якщо ні, то робимо кнопку видалення неактивною.

```
DynamicButtons >> removeFromLayout
   self layout remove: self layout presenters last.
   self layout presenters last = text ifTrue: [ removeButton disable ]
```

Єдине, чого ще не вистачає, то це макет за замовчуванням.

```
DynamicButtons >> defaultLayout
   ^ SpBoxLayout newTopToBottom
         add: addButton expand: false;
         add: removeButton expand: false;
         add: text;
         yourself
```

Виконання коду *DynamicButtons new open* мало б відкрити вікно, як на рис. 11.6.

На рис. 11.7 показано, як виглядає вікно після чотириразового натискання кнопки додавання демонстраторів.

11.4. Побудова невеликого динамічного оглядача

З усіма знаннями, отриманими на цей момент, спробуємо створити нову мініверсію системного оглядача, як на рис. 11.8. Хотілося б мати:

- Дерево, яке показує всі класи системи.
- Список, який показує всі методи вибраного класу.
- Текстовий редактор, який показує код вибраного методу.
- Кнопку для перемикання режимів редагування.

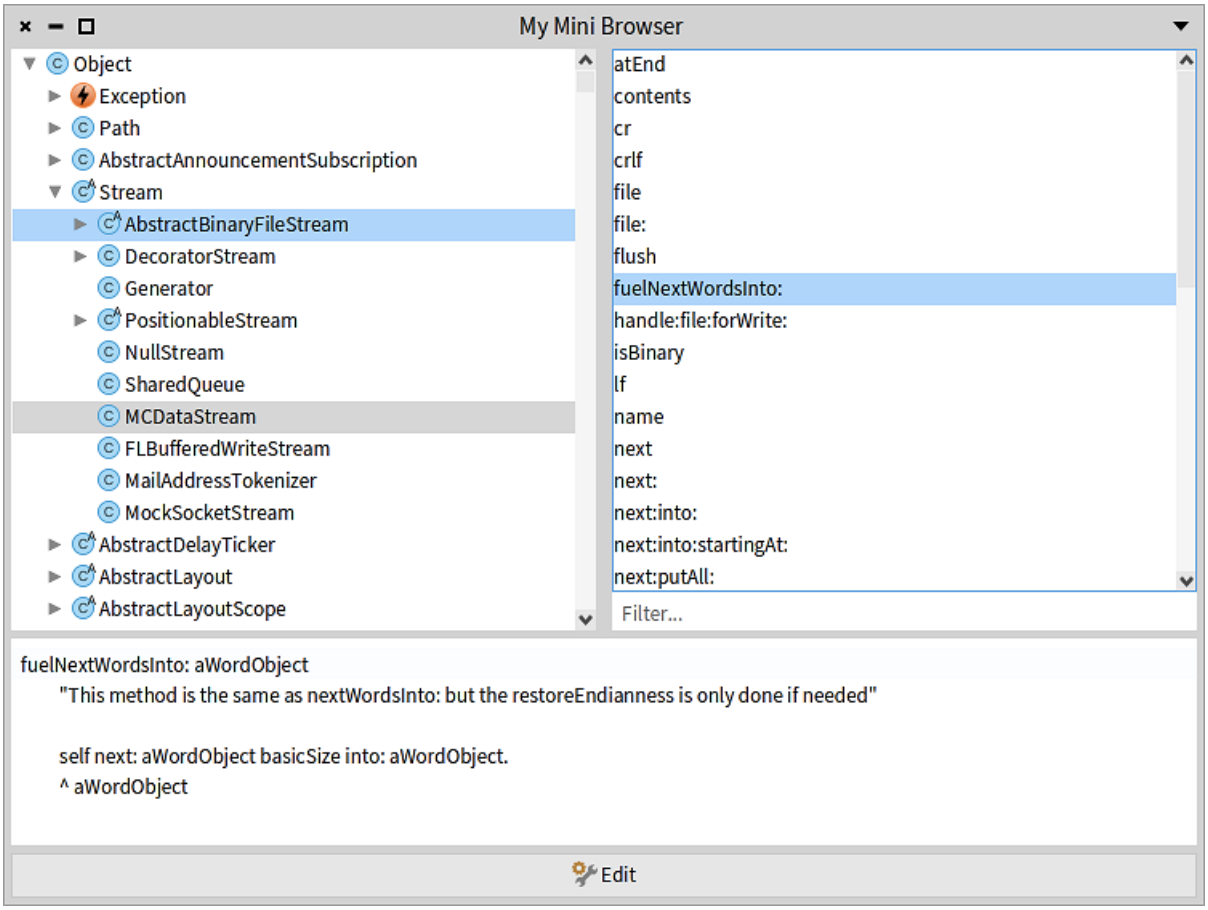


Рис. 11.8. Мініоглядач у дії

Спочатку редактор тексту буде в режимі «Тільки для читання». Натисканням на кнопку можна перевести його в режим «Редагування».

Розпочнімо.

```
SpPresenter << #MyMiniBrowser
   slots: { #classTree . #code . #methodList . #button };
   package: 'CodeOfSpec20Book'
```

Метод *initializePresenters* створює екземпляр демонстратора дерева. Він має показувати усі класи, наявні в образі Pharo. Відомо, що (майже) усі підкласи наслідують *Object*, тому це буде єдиний корінь дерева. Щоб отримати нащадків вузла дерева, можна надіслати класу повідомлення *subclasses*. Добре було б, щоб кожен із вузлів дерева мав гарний значок. Піктограму класу можна отримати за допомогою повідомлення *systemIconName*. Нарешті, налаштуємо демонстратор так, щоб його можна було «активувати» простим клацанням мишки, а не подвійним.

```
MyMiniBrowser >> initializePresenters
   classTree := self newTree
      activateOnSingleClick;
      roots: { Object };
      children: [ :each | each subclasses ];
      displayIcon: [ :each | self iconNamed: each systemIconName ];
      yourself.
```

Для відображення методів варто використати список з фільтром, щоб легко було знаходити селектори методів. Елементами списку будуть селектори методів, впорядковані за зростанням.

```
        methodList := self newFilteringList
              display: [ :method | method selector ].
        methodList listPresenter
           sortingBlock: [ :method | method selector ] ascending.
```

Як було з'ясовано, спочатку редактор коду буде в режимі «Тільки для читання». Напис ***Edit*** на кнопці повідомлятиме, що натискання на неї переводить у режим «Редагування». Також варто оздобити кнопку відповідною піктограмою.

```
        button := self newButton
           label: 'Edit';
           icon: (self iconNamed: #smallConfiguration);
           yourself.
```

Оскільки початковий режим редактора лише для читання, то для відображення коду використаємо демонстратор тексту, в якому не можна редагувати.

```
        code := self newText.
        code beNotEditable
```

Ось повний код методу:

```
     MyMiniBrowser >> initializePresenters
        classTree := self newTree
           activateOnSingleClick;
           roots: { Object };
           children: [ :each | each subclasses ];
           displayIcon: [ :each | self iconNamed: each systemIconName ];
           yourself.
        methodList := self newFilteringList
           display: [ :method | method selector ].
        methodList listPresenter
           sortingBlock: [ :method | method selector ] ascending.
        button := self newButton
           label: 'Edit';
           icon: (self iconNamed: #smallConfiguration);
           yourself.
        code := self newText.
        code beNotEditable
```

11.5. Візуальне розташування компонентів

Вкладені демонстратори створено, але ще не відомо, як їх відображати.

Дерево класів і список методів розташуємо у верхній частині макета в ряд, як у Системному оглядачі. Використаємо для цього орієнтований зліва направо послідовний макет з інтервалом 10 пікселів між компонентами.

Додамо його до основного макета, теж послідовного, орієнтованого згори донизу. Під ним розташуємо демонстратор тексту і кнопку, задамо інтервал між ними 5 пікселів. Демонстратор тексту налаштуємо так, щоб він не розгортався на все вільне місце.

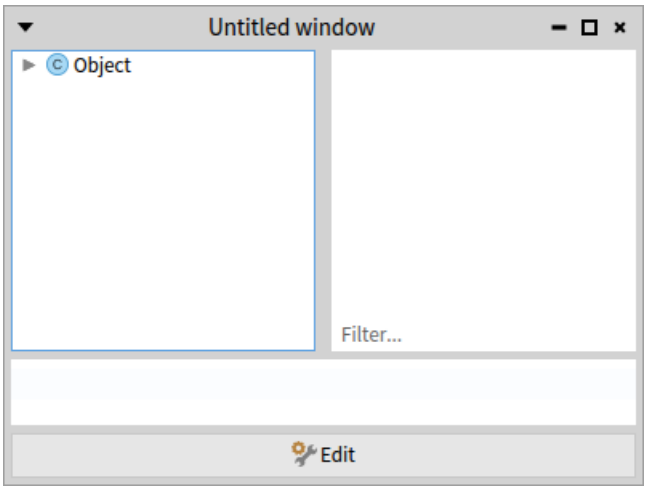


Рис. 11.9. Невеликий оглядач у режимі тільки для читання

```
MyMiniBrowser >> defaultLayout
   | classesAndMethodsLayout |
   classesAndMethodsLayout := SpBoxLayout newLeftToRight.
   classesAndMethodsLayout
      spacing: 10;
      add: classTree;
      add: methodList.
   ^ SpBoxLayout newTopToBottom
      spacing: 5;
      add: classesAndMethodsLayout;
      add: code;
      add: button expand: false;
      yourself
```

Демонстратор мініоглядача можна відкрити за допомогою коду *MyMiniBrowser new open*. Мало б з'явитися вікно як на рис. 11.9.

11.6. Налаштування взаємодії

Поки що все добре, але демонстратори ще не наділені жодною поведінкою. Треба реалізувати метод *connectPresenters*.

Коли натискають на клас у дереві, список методів мав би оновитися методами вибраного класу. Коли ж натискають на методі, демонстратор тексту мав би відобразити його програмний код.

```
MyMiniBrowserPresenter >> connectPresenters
   classTree whenActivatedDo: [ :selection |
      methodList items: selection selectedItem methods ].
    methodList listPresenter
      whenSelectedDo: [ :selectedMethod |
         code text: selectedMethod ast formattedCode ].
   button action: [ self buttonAction ]
```

Тимчасово визначимо метод *buttonAction*, який нічого не робить.

```
MyMiniBrowserPresenter >> buttonAction
   ^ self
```

11.7. Перемикання режимів редагування/тільки для читання

Коли користувач натисне на кнопку, має відбутися кілька речей, тому ліпше створити окремий метод.

1. Треба почергово міняти напис на кнопці з «Edit» на «Read only» і навпаки.
2. Треба змінити демонстратор коду. Якщо мінібраузер перебуває в режимі лише для читання, то достатньо використати демонстратор тексту, який не можна редагувати. Якщо ж мінібраузер перемкнули в режим редагування, то треба мати демонстратор коду, який застосовує синтаксичне підсвічування та показує номери рядків. Але в обох випадках демонстратор відображатиме той самий текст (код вибраного методу).

```
MyMiniBrowserPresenter >> buttonAction
   | newCode |
   button label = 'Edit'
      ifTrue: [
         button label: 'Read only'.
         newCode := self newCode
            beForMethod: methodList selectedItem;
            text: methodList selectedItem ast formattedCode;
         yourself ]
      ifFalse: [
         button label: 'Edit'.
         newCode := self newText
            text: methodList selectedItem ast formattedCode;
            beNotEditable;
            yourself ].

   self layout replace: code with: newCode.
   code := newCode
```

І наостанок, оскільки деталі важливі, то замість безликого заголовка вікна «Untitled window» задамо змістовний і змінимо початковий розмір вікна. Для цього визначимо метод *initializeWindow:*.

```
MyMiniBrowserPresenter >> initializeWindow: aWindowPresenter
   aWindowPresenter
      title: 'My Mini Browser';
      initialExtent: 750@650
```

Вуаля! Нова мінімальна версія системного браузера з режимом лише для читання готова. Коли після запуску *MyMiniBrowser* вибрати клас, метод і натиснути кнопку ***Edit***, то побачимо вікно, схоже до зображеного на рис. 11.10.

Від перекладача. Деталі справді важливі, тому варто було б зробити кілька доповнень до написаного. Логічно було б сподіватися, що після зміни вибраного класу мініоглядач мав би оновити також і текстове поле, але зараз це не так: він продовжує відображати код попереднього вибраного методу. Доповнимо опрацювання події активації дерева класів у методі *connectPresenters*.

```
classTree whenActivatedDo: [ :selection |
    methodList items: selection selectedItem methods.
    code text: '' ].
```

Дерево класів *Object* дуже велике, тому знайти в ньому потрібний клас досить складно. Щоб трохи полегшити ситуацію, можна було б впорядковувати імена підкласів у алфавітному порядку. Дерево стане впорядкованим, якщо оголосити метод *initializePresenters* у такій редакції, як на рис. 11.10.

Можливість редагувати код методу в мініоглядачі позірна, бо автори посібника не розповіли, як зберігати, компілювати відредагований код.

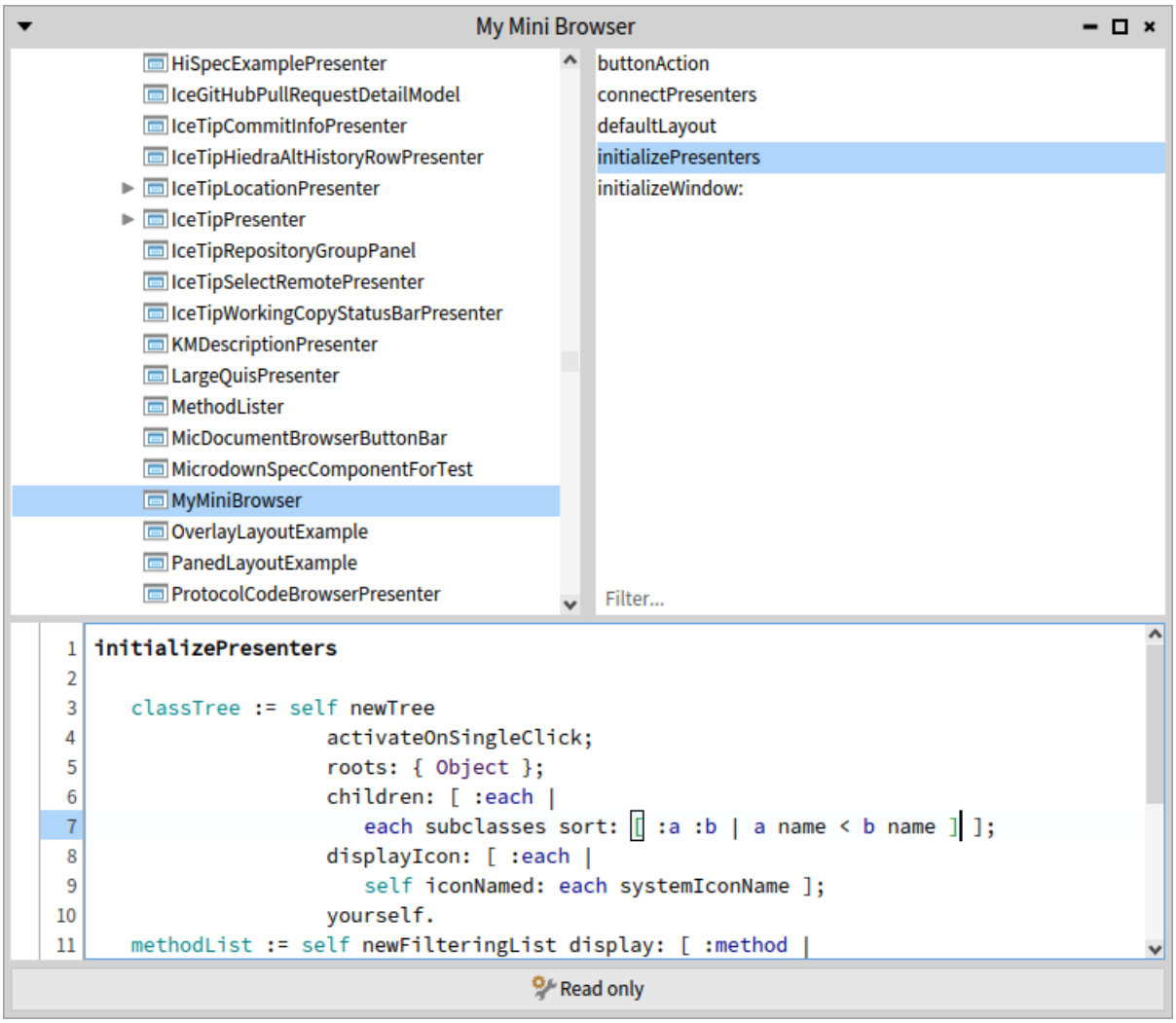


Рис. 11.10. Невеликий оглядач класів у режимі редагування

11.8. Про перебудову макета

Треба враховувати затрати на перебудову макета, бо вони можуть призвести до зниження продуктивності.

Уявімо, що демонстратор має макет із багатьма вкладеними демонстраторами, а ті також мають макети з піддемонстраторами. Макети дають змогу додавати і видаляти демонстратори. Такі операції не безкоштовні. Кожна зміна макета спричиняє перерахунок, оскільки будь-яке додавання чи видалення впливає на те, як демонстратори в макеті відображаються на екрані. Отже, коли демонстратор змінює кілька окремих піддемонстраторів макета, відбудеться кратна кількість обчислень.

Бажано виконувати зміни макета за один раз. Під час побудови початкового макета краще будувати вкладені макети знизу догори і задавати загальний макет один раз. Якщо треба оновити наявний макет, то ліпше повністю створити новий і встановити його, замість хірургічного додавання та/або видалення окремих демонстраторів.

11.9. Підсумки розділу

За допомогою Spec програми можна створювати від дуже простих до дуже складних. Динамічні макети дають змогу змінювати вигляд на льоту. Макети можна налаштувати кількома способами, тому перегляньте їхні класи та доступні приклади. Spec має багато готових до використання демонстраторів. Почніть копатися в коді, щоб побачити, які демонстратори доступні, і щоб дізнатися їхній API.

Від перекладача. Звернемо увагу читача на деякі «новинки» цього розділу. Доступністю демонстраторів можна керувати за допомогою повідомлень *enable*, *disable*. Макет зберігає колекцію вкладених демонстраторів, а доступ до неї можна отримати завдяки повідомленню *presenters*. Вигляд застосунку і склад його графічного інтерфейсу можна змінювати динамічно, під час виконання, а зручним способом заміни одного з піддемонстраторів є використання методу *replace:with:* макета застосунку.

Розділ 12

Конкретний випадок: поштовий застосунок

Створимо невелику клієнтську програму електронної пошти. Дороблятимемо її та пристосовуватимемо і в наступних розділах. Вона об'єднає в одному місці чимало з побаченого в попередніх розділах у конкретних умовах застосування. На рис. 12.1 зображено мету розробки[13].

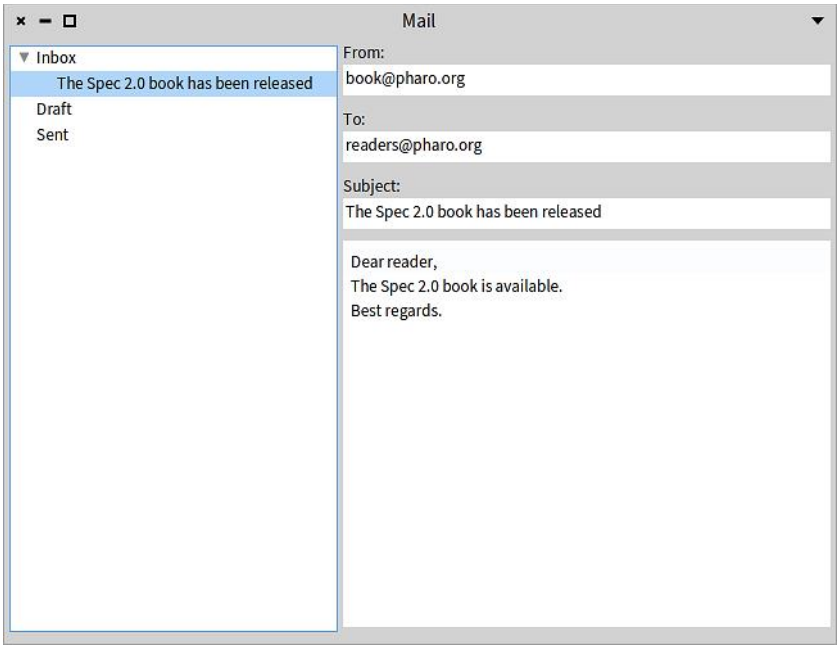


Рис. 12.1. Клієнтська програма електронної пошти

Приклад обширний, з достатньою кількістю класів і методів. Будемо реалізовувати його знизу догори. Почнемо з моделей. Після цього реалізуємо демонстратори, які становлять застосунок. Пориньмо в роботу.

12.1. Моделі

Щоб створити поштового клієнта, потрібні три моделі (див. рис. 12.2):

- *Email* представляє електронного листа.
- *MailFolder* представляє папку, яка містить електронні листи, наприклад, «Вхідні», «Чернетки» та «Надіслані».
- *MailAccount* представляє обліковий запис електронної пошти. Він містить усі електронні листи.

[13] Варто зазначити, що увага авторів зосереджена на розробці графічного інтерфейсу користувача застосунку та демонстрації різноманітних можливостей Spec. Приклад дуже цікавий і повчальний, буде з читачем до кінця книжки. Але готова програма *не взаємодіятиме* з поштовим сервером. Про нього тут навіть не згадують. Це зрозуміло, бо приклад не про справжнє надсилання електронних листів, а про розробку цікавого та зручного інтерфейсу (прим. – *Ярошко С.*).

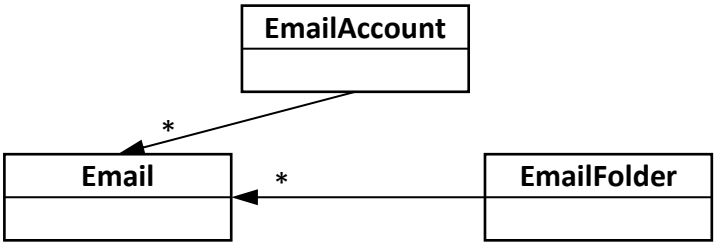


Рис. 12.2. Модель даних застосунку

12.2. Електронний лист

На рис. 12.1 видно, що програма показує для електронного листа чотири поля. «From» містить відправника, «To» – адресата, «Subject» – тему повідомлення. Текстове поле без назви в нижньому правому куті відображає текст листа. Визначимо клас *Email* з переліком полів для зберігання таких даних.

```
Object << #Email
   slots: { #from . #to . #subject . #body . #status };
   package: 'CodeOfSpec20Book'
```

Методи доступу до полів *from*, *to*, *subject*, *body* не показуємо, бо вони тривіальні.

Існує ще п’ята змінна екземпляра – *status*. Її використовуватимемо для відстеження статусу електронного листа: *#received*, *#draft* або *#sent*. Відповідно до статусу лист потраплятиме у програмі до однієї з папок «Вхідні» (Inbox), «Чернетки» (Draft) або «Надіслані» (Sent). Визначимо методи зміни статусу електронного листа. Вони стануть у пригоді, коли листа отримують, створюють чи надсилають.

```
Email >> beReceived
   status := #received

Email >> beDraft
   status := #draft

Email >> beSent
   status := #sent
```

Визначимо ще три методи, щоб довідуватися статус електронного листа.

```
Email >> isReceived
   ^ status = #received

Email >> isDraft
   ^ status = #draft

Email >> isSent
   ^ status = #sent
```

Немає потреби визначати методи доступу до змінної екземпляра *status*. Наведені вище шість методів добре інкапсулюють її.

Тепер можна визначити метод *initialize*. У ньому зазначено, що новий електронний лист отримує статус чернетки.

```
Email >> initialize
   super initialize.
   self beDraft
```

Визначимо два останні методи. Вони потрібні для включення електронних листів у демонстратор дерева. Перший метод повертає рядок, який відображатиметься в списку листів.

```
Email >> displayName
   ^ subject
```

Другий метод повідомляє, що відображати дочірніми елементами вузла дерева. Дочірні елементи є у папки, її електронні листи, а лист не має дочірніх елементів, тому цей метод повертає порожній масив. Тут було б недоречно використовувати термінологію, пов'язану з деревами, тому метод назвали *content* (вміст), що може означати «вміст папки».

```
Email >> content
   ^ Array new
```

12.3. Папка для листів

Дерево в лівій частині вікна відображає не тільки електронні листи, а й папки, в які групують листи відповідно до їхнього статусу. Модель *MailFolder* визначимо дуже спрощено. Папка має назву та містить електронні листи.

```
Object << #MailFolder
   slots: { #emails . #name };
   package: 'CodeOfSpec20Book'
```

На момент ініціалізації екземпляр *MailFolder* не має жодних листів і називається «New folder».

```
MailFolder >> initialize
   super initialize.
   emails := OrderedCollection new.
   name := 'New folder'
```

Цей метод задає стандартний стан екземпляра *MailFolder*, але на практиці зручно буде використовувати метод створення екземпляра.

```
MailFolder class >> named: aString emails: aCollection
   ^ self new
      name: aString;
      emails: aCollection;
      yourself
```

Метод створення потребує оголошення методів-модифікаторів.

```
MailFolder >> emails: aCollection
   emails := aCollection

MailFolder >> name: aString
   name := aString
```

Подібно до класу *Email*, потрібні деякі методи, пов'язані з деревом.

```
MailFolder >> displayName
   ^ name

MailFolder >> content
   ^ emails
```

З цієї реалізації видно, що *MailFolder* – це лише іменований об'єкт-контейнер для електронних листів, який можна використовувати для структурування відображення листів у демонстраторі дерева.

Як відрізнити лист від папки

У задуманій програмі папки та електронні листи відображаються у вигляді дерева. Користувач може вибрати на дереві папку або листа. Демонстратор мав би діяти по-різному залежно від типу вибору, тому йому потрібен спосіб розрізняти папки та листи. Щоб зробити все просто, додамо по одному методу до вже оголошених класів моделі.

```
Email >> isEmail
   ^ true

Folder >> isEmail
   ^ false
```

12.4. Обліковий запис

MailAccount містить усі електронні листи, тому визначення класу просте.

```
Model << #MailAccount
   slots: { #emails };
   package: 'CodeOfSpec20Book'
```

Зверніть увагу на те, що це перший клас моделі даних застосунку, який наслідує *Model*. Щоб спростити ситуацію, клієнтська програма електронної пошти залежатиме тільки від екземпляра *MailAccount*, але не від екземплярів *Email* і *MailFolder*.

Ініціалізація тривіальна.

```
MailAccount >> initialize
   super initialize.
   emails := OrderedCollection new
```

Відомо, що електронні листи мають статус, і що статус використовують для розподілу електронних листів у окремі папки. Ось тут стануть у пригоді такі методи:

```
MailAccount >> receivedEmails
   ^ emails select: [ :each | each isReceived ]

MailAccount >> draftEmails
   ^ emails select: [ :each | each isDraft ]

MailAccount >> sentEmails
   ^ emails select: [ :each | each isSent ]
```

Клас *MailAccount* є головною моделлю в програмі, тому він визначає деякі дії.

Перш за все, електронного листа можна отримати. У реальній програмі листи надходять із сервера. Поки що не будемо йти так далеко. Достатньо розмістити в обліковому записі один електронний лист.

```
MailAccount >> fetchMail
   | email |
   email := Email new
      from: 'book@pharo.org';
      to: 'readers@pharo.org';
      subject: 'The Spec 2.0 book has been released';
      body: 'Dear reader,
The Spec 2.0 book is available.
Best regards.';
      beReceived;
      yourself.
   (emails includes: email) ifFalse: [ emails add: email ].
   self changed
```

Цей метод створює електронний лист і надає йому статус «отримано». Потім додає його до колекції збережених раніше. Додавання виконується з перевіркою, бо не варто зберігати двічі той самий електронний лист після кількаразового отримання.

Зверніть увагу на *self changes* наприкінці. Повідомлення сповіщає зацікавлених про те, що екземпляр *MailAccount* якось змінився. Можливі деталізованіші повідомлення про зміни, але вони не дуже потрібні в цьому прикладі програми. Пригадаймо, що зараз робимо все просто.

Користувач застосунку може створювати нові листи та зберігати їх. Зберігатимуться вони у статусі чернетки, як визначено в методі нижче.

```
MailAccount >> saveAsDraft: anEmail
   anEmail beDraft.
   (emails includes: anEmail) ifFalse: [ emails add: anEmail ].
   self changed
```

Збереження листа методом *saveAsDraft* використовує зміну статусу листа на «чернетка» та додавання його до колекції, якщо його там ще немає. Умовне додавання дає змогу зберігати лист кілька разів, але додавати тільки раз.

Спосіб надсилання електронного листа подібний до методу зберігання.

```
MailAccount >> send: anEmail
   anEmail beSent.
   (emails includes: anEmail) ifFalse: [ emails add: anEmail ].
   self changed
```

Нарешті, електронний лист можна видалити. Реалізація проста. Видаліть листа з облікового запису та повідомте про це зацікавлених.

```
MailAccount >> delete: anEmail
   emails remove: anEmail.
   self changed
```

На цьому побудова моделі завершена. Тепер можна взятися до демонстраторів.

12.5. Демонстратори

Багато демонстраторів складаються з менших демонстраторів. Так буде і тут. Потрібний демонстратор для показу електронного листа. Також потрібний демонстратор для відображення дерева листів. Якщо в дереві не вибрано жодного листа, то було б доцільно відобразити інформаційне повідомлення. Це ще один демонстратор. Динамічну заміну демонстратора інформаційного повідомлення на демонстратор вмісту листа і навпаки виконуватиме окремий демонстратор. Увесь застосунок, який все об'єднує, також є демонстратором. Отже, йдеться про п'ять демонстраторів.

- *EmailPresenter* відображає екземпляр *Email*, доступний для редагування або лише для читання. Поля можна редагувати, коли електронний лист є чернеткою. Коли лист отримано або надіслано, поля доступні лише для читання.
- *NoEmailPresenter* відображає інформаційне повідомлення про те, що жоден лист не вибраний.
- *MailReaderPresenter* відповідає за показ листа або інформаційного повідомлення. Щоб зробити це, він використовує два демонстратори, згадані вище.
- *MailAccountPresenter* відображає дерево папок і електронних листів.
- *MailClientPresenter* – головний демонстратор. Його скомпоновано з *MailAccount-Presenter* і *MailReaderPresenter* для реалізації функціонала застосунку.

12.6. Демонстратор листа

Цей демонстратор досить простий. Він показує вміст одного електронного листа. Тому клас визначає змінні екземпляра для всіх атрибутів екземпляра *Email*, крім *status*.

```
SpPresenterWithModel << #EmailPresenter
   slots: { #from . #to . #subject . #body };
   package: 'CodeOfSpec20Book'
```

Зверніть увагу, клас демонстратора наслідує *SpPresenterWithModel*, а це означає, що він отримує поле *model* і методи доступу до нього. Екземпляр *EmailPresenter* не може працювати без електронного листа, як і написано в методі *initialize*. Метод задає моделлю порожній *Email*. Нагадуємо, що новостворений лист усталено має статус чернетки.

```
EmailPresenter >> initialize
   self model: Email new.
   super initialize
```

Як завжди, потрібно визначити деякі важливі методи.

```
EmailPresenter >> initializePresenters
   from := self newTextInput.
   to := self newTextInput.
   subject := self newTextInput.
   body := self newText

EmailPresenter >> defaultLayout
   | toLine subjectLine fromLine |
   fromLine := SpBoxLayout newTopToBottom
      add: 'From:' expand: false;
      add: from expand: false;
      yourself.
```

```
        toLine := SpBoxLayout newTopToBottom
           add: 'To:' expand: false;
           add: to expand: false;
           yourself.
        subjectLine := SpBoxLayout newTopToBottom
           add: 'Subject:' expand: false;
           add: subject expand: false;
           yourself.
        ^ SpBoxLayout newTopToBottom
              spacing: 10;
              add: fromLine expand: false;
              add: toLine expand: false;
              add: subjectLine expand: false;
              add: body;
              yourself
```

Поля введення *from*, *to* і *subject* об'єднують з відповідними написами власні макети. Зауважте, що *body* не має над собою напису. З контексту зрозуміло, що поле містить текст електронного листа. Загальний макет – послідовний вертикальний макет з інтервалами 10 пікселів між полями.

Метод *connectPresenters* зазначає, що вміст полів уведення після змін має зберігатися в електронному листі, який міститься в змінній *model* екземпляра *EmailPresenter*.

```
EmailPresenter >> connectPresenters
   from whenTextChangedDo: [ :text | self model from: text ].
   to whenTextChangedDo: [ :text | self model to: text ].
   subject whenTextChangedDo: [ :text | self model subject: text ].
   body whenTextChangedDo: [ :text | self model body: text ]
```

Від перекладача. Автори книги описали, як перенести зміни з демонстратора в модель, але чомусь змовчали про важливу обернену дію: зміни моделі мають потрапляти в демонстратор. Адже користувач застосунку може вибрати іншого листа зі списку, і тоді нові дані повинні потрапляти і в модель, і на екран. Щоб забезпечити зворотний зв'язок модель-демонстратор, визначають метод *modelChanged*. Тут це можна зробити так.

```
EmailPresenter >> modelChanged
   from text: (self model from ifNil: [ '' ]).
   to text: (self model to ifNil: [ '' ]).
   subject text: (self model subject ifNil: [ '' ]).
   body text: (self model body ifNil: [ '' ])
```

До речі, цей метод є в оголошенні класу в репозиторії коду книги на GitHub.

Для зручності в майбутньому визначимо два додаткові методи, які перемикають поля введення у режим для редагування або лише для читання.

```
EmailPresenter >> beEditable
   from editable: true.
   to editable: true.
   subject editable: true.
   body editable: true

EmailPresenter >> beReadOnly
   from editable: false.
```

```
        to editable: false.
        subject editable: false.
        body editable: false
```

12.7. Демонстратор інформаційного повідомлення

Цей демонстратор використовуватиметься, доки користувач не вибере листа в дереві папок і електронних листів. Зробити його просто, бо він не виконує ніяких дій.

```
SpPresenter << #NoEmailPresenter
   slots: { #message };
   package: 'CodeOfSpec20Book'

NoEmailPresenter >> initializePresenters
   message := self newLabel
      label: 'Select an email from the list to read it.';
      yourself
```

Розташуємо напис у центрі демонстратора за допомогою повідомлень *hAlignCenter* і *vAlignCenter*.

```
NoEmailPresenter >> defaultLayout
   ^ SpBoxLayout newTopToBottom
      hAlignCenter;
      vAlignCenter;
      add: message;
      yourself
```

Ось і все, що треба зробити з цим демонстратором.

12.8. Демонстратор-обгортка

Настав час об'єднати два попередні демонстратори. Це відповідальність *MailReaderPresenter*. Він містить обидва демонстратори, але показує тільки одного з них. Така поведінка слугує хорошою ілюстрацією того, що можна динамічно змінювати макети, щоб відображати різні піддемонстратори.

```
SpPresenter << #MailReaderPresenter
   slots: { #content . #noContent };
   package: 'CodeOfSpec20Book'
```

Як бачите, є дві змінні екземпляра для зберігання екземплярів двох попередніх класів демонстратора. Зауважте, що клас наслідує від *SpPresenter*, а не від *SpPresenterWithModel*. Це означає, що *MailReaderPresenter* не має моделі. Припускаємо, що екземплярам *MailReaderPresenter* буде сказано про потребу оновити свій стан.

```
MailReaderPresenter >> initializePresenters
   content := EmailPresenter new.
   noContent := NoEmailPresenter new
```

Демонстратор перебуває в одному з двох станів. Або є *Email*, або його немає. Для кожного стану є макет. Коли електронний лист є, будемо використовувати *emailLayout*.

```
MailReaderPresenter >> emailLayout
   ^ SpBoxLayout newLeftToRight
         add: content;
         yourself
```

Якщо електронного листа немає, то будемо використовувати *noEmailLayout*.

```
MailReaderPresenter >> noEmailLayout
   ^ SpBoxLayout newLeftToRight
         add: noContent;
         yourself
```

Припускаємо, що в початковому стані електронного листа немає, адже ще не виконано ніякого методу, який отримує чи створює листа. Тому *defaultLayout* – це *noEmailLayout*.

```
MailReaderPresenter >> defaultLayout
   ^ self noEmailLayout
```

Як було вже згадано, припускаємо, що екземплярам *MailReaderPresenter* буде наказано оновлюватися. Повідомлення *read:* і є той наказ[14].

```
MailReaderPresenter >> read: folderOrEmail
   (folderOrEmail isNotNil and: [ folderOrEmail isEmail ])
      ifTrue: [ self updateLayoutForEmail: folderOrEmail ]
      ifFalse: [ self updateLayoutForNoEmail ]
```

Метод *read:* делегує всю роботу двом іншим методам.

```
MailReaderPresenter >> updateLayoutForEmail: email
   content model: email.
   self layout: self emailLayout.
   email isDraft
      ifTrue: [ content beEditable ]
      ifFalse: [ content beReadOnly ]

MailReaderPresenter >> updateLayoutForNoEmail
   self layout: self noEmailLayout
```

Ці методи за допомогою *self layout:* просто перемикають макет. Зауважте, що перший з них налаштовує режим демонстратора листа на основі статусу екземпляра *Email*: чи вмикати режим редагування залежить від того, чернетка то чи ні.

12.9. Демонстратор облікового запису

Тепер визначимо важливу частину функціональних можливостей програми поштового клієнта. *MailAccountPresenter* містить дерево папок і електронних листів.

```
SpPresenterWithModel << #MailAccountPresenter
   slots: { #foldersAndEmails };
   package: 'CodeOfSpec20Book'
```

[14] Метод *read:* викладено в редакції з репозиторію програмного коду книги на GitHub. Вона відрізняється від опублікованої в книзі і працює без помилок і тоді, коли користувач вибирає зі списку не листа, а папку (прим. – *Ярошко С.*).

Клас демонстратора наслідує *SpPresenterWithModel*, тому що він міститиме екземпляр *MailAccount* як свою модель, яка містить електронні листи для показу в дереві. Метод *initializePresenters* визначає дерево.

```
MailAccountPresenter >> initializePresenters
   foldersAndEmails := self newTree
      roots: Array new;
      display: [ :node | node displayName ];
      children: [ :node | node content ];
      expandRoots
```

Розберемо цей метод.

- Спочатку дерево не має коренів. Пізніше зробимо кореневими елементами папки «Вхідні», «Чернетки» та «Надіслані» (див. метод *modelChanged* нижче).
- Демонстратор дерева використовує блок методу *display:*, щоб отримати рядкове зображення кожного вузла дерева. У блоці вузлові надсилають повідомлення *displayName*. Відповідні методи вже визначені в класах моделі *Email* і *MailFolder*.
- Демонстратор дерева використовує блок методу *children:*, щоб отримати дочірні елементи вузла дерева. Папки мають дочірні елементи, а електронні листи – ні. У блоці вузлові надсилають повідомлення *content*. Пригадуємо, що екземпляр *MailFolder* повертатиме у відповідь свою колекцію листів, а екземпляр *Email* – порожній масив, що означає, що електронні листи – листки дерева.
- Наприкінці надсилаємо повідомлення *expandRoots*, щоб розгорнути все дерево.

Макет – послідовний вертикальний з демонстратором дерева.

```
MailAccountPresenter >> defaultLayout
   ^ SpBoxLayout newTopToBottom
            add: foldersAndEmails;
            yourself
```

За замовчуванням дерево порожнє. Коли модель змінюється, дерево треба оновити. Для цього можна перевизначити метод *modelChanged*, адже клас *MailAccountPresenter* наслідує клас *SpPresenterWithModel*.

```
MailAccountPresenter >> modelChanged
   | inbox draft sent |
   inbox := MailFolder named: 'Inbox' emails: self model receivedEmails.
   draft := MailFolder named: 'Draft' emails: self model draftEmails.
   sent := MailFolder named: 'Sent' emails: self model sentEmails.
   foldersAndEmails
      roots: { inbox . draft . sent };
      expandRoots
```

Моделлю є екземпляр *MailAccount*. Метод отримує від нього вибрані на основі статусу електронні листи та створює відповідні папки. У кожній папці міститимуться листи з однаковим статусом. Метод надсилає *receivedEmails*, *draftEmails* і *sentEmails*. Відповідні методи були визначені в класі *MailAccount*. Три створені папки стають коренями дерева. Їх розгортають за допомогою повідомлення *expandRoots*, щоб користувач бачив усе дерево.

Під час створення демонстратора з деревом, або будь-якого компонента, що підтримує вибір, завжди доцільно визначати метод, який дає змогу реагувати на зміни виділення. Такий метод знадобиться пізніше, щоб підключити *MailAccountPresenter* до *MailReader*.

```
MailAccountPresenter >> whenSelectionChangedDo: aBlock
   foldersAndEmails whenSelectionChangedDo: aBlock
```

Метод просто делегує демонстраторові дерева, який зберігається у *foldersAndEmails*.

Визначимо два додаткові методи, пов'язані з вибором, які стануть у нагоді пізніше. Перший з них повертає логічне значення, яке повідомляє, чи вибрано електронний лист. У дереві є лише два рівні, тому, якщо шлях до вибору містить два елементи, то можна стверджувати, що вибрано електронний лист. Другий метод просто повертає вибраний елемент у дереві.

```
MailAccountPresenter >> hasSelectedEmail
   ^ foldersAndEmails selection selectedPath size = 2

MailAccountPresenter >> selectedItem
   ^ foldersAndEmails selectedItem
```

MailAccountPresenter не надає інших функцій, окрім опрацювання вибору. Поки що ні. Впровадимо їх пізніше, коли вони знадобляться.

Майже все готово. Залишається один демонстратор.

12.10. Демонстратор поштового клієнта

Цей демонстратор поєднує в собі всі розроблені до тепер демонстратори. Почнемо з початкової версії класу. Розвинемо його в наступних розділах.

```
SpPresenterWithModel << #MailClientPresenter
   slots: { #account . #reader . #editedEmail };
   package: 'CodeOfSpec20Book'
```

Клас наслідує *SpPresenterWithModel*. Модель є екземпляром *MailAccount*. Є три змінні екземпляра. Перші дві – для демонстраторів. Третя містить електронного листа, який редагують.

```
MailClientPresenter >> initializePresenters
   account := MailAccountPresenter on: self model.
   reader := MailReaderPresenter new
```

Використаємо панельний макет, у якому 40% простору виділено для *MailAccount-Presenter*.

```
MailClientPresenter >> defaultLayout
   ^ SpPanedLayout newLeftToRight
         positionOfSlider: 40 percent;
         add: account;
         add: reader;
         yourself
```

Поєднаймо два демонстратори так, щоб вибір у дереві ліворуч спричиняв відображення деталей вибраного листа праворуч. Визначимо для цього методи *connectPresenters* і *updateAfterSelectionChangedTo:*.

- Метод *connectPresenters* надсилає вибраний елемент дерева до *reader* і викликає другий метод.
- Метод *updateAfterSelectionChangedTo:* виконує дії після опрацювання вибору.

```
MailClientPresenter >> connectPresenters
   account whenSelectionChangedDo: [ :selection |
      | selectedFolderOrEmail |
      selectedFolderOrEmail := selection selectedItem.
      reader read: selectedFolderOrEmail.
      self updateAfterSelectionChangedTo: selectedFolderOrEmail ]
```

У методі *updateAfterSelectionChangedTo:* використовуємо кілька повідомлень, визначених раніше.

```
MailClientPresenter >> updateAfterSelectionChangedTo: selectedFolderOrEmail
     editedEmail := (self isDraftEmail: selectedFolderOrEmail)
        ifTrue: [ selectedFolderOrEmail ]
        ifFalse: [ nil ]
```

Чернетки електронних листів можна редагувати. Метод *updateAfterSelectionChangedTo:* перевіряє, чи є вибраний елемент дерева чернеткою листа, і зберігає його в змінній екземпляра, якщо так, то щоб демонстратор мав його під рукою, коли він знадобиться. Для виконання перевірки викликається метод *isDraftEmail:*, визначений нижче.

```
MailClientPresenter >> isDraftEmail: folderOrEmailOrNil
   ^ folderOrEmailOrNil isNotNil and: [
      folderOrEmailOrNil isEmail and: [ folderOrEmailOrNil isDraft ] ]
```

Рис. 12.3. Основа поштового клієнта

Метод *connectPresenters* визначає, що вміст демонстратора *MailReaderPresenter*, який зберігається в *reader*, залежить від вибору в дереві. Якщо вибрано електронний лист, то *reader* показує його поля. Якщо ж не вибрано нічого, або вибрано папку, то *reader* показує інформаційне повідомлення. Якщо вибрано чернетку електронного листа, то його поміщають в змінну екземпляра *editedMail*. Це буде зручно, коли почнемо виконувати дії з вибраним листом.

Визначимо також метод *initializeWindow*, щоб вікно мало заголовок і було достатньо великим для легкого читання електронних листів.

```
MailClientPresenter >> initializeWindow: aWindowPresenter
   super initializeWindow: aWindowPresenter.
   aWindowPresenter
      title: 'Mail';
      initialExtent: 650@500
```

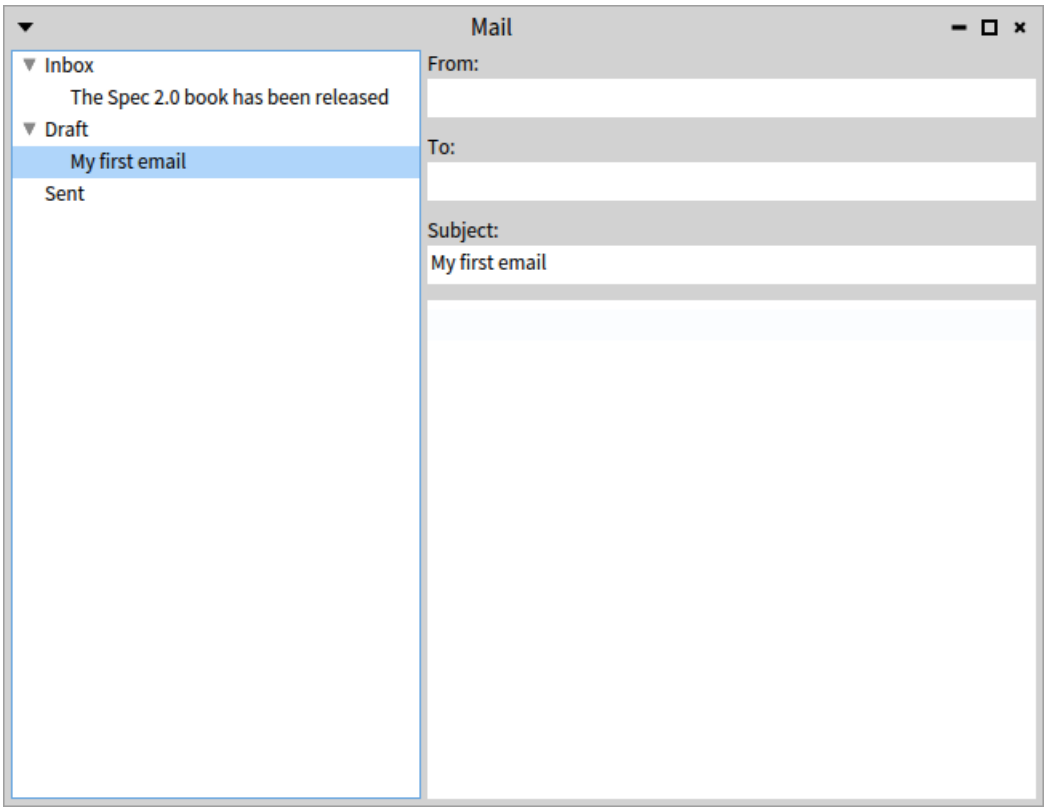


Рис. 12.4. Поштовий клієнт з чернеткою електронного листа

12.11. Увесь застосунок

Увесь програмний код набрано, настав час відкрити поштового клієнта.

```
(MailClientPresenter on: MailAccount new) open
```

Рис. 12.3 демонструє результат. Немає на що дивитися. Тільки три порожні папки. Якщо вибрати одну з них, то інформаційне повідомлення все одно відображатиметься праворуч.

Можна діяти розумніше. Створимо екземпляр *MailAccount* і наповнимо його кількома листами за допомогою раніше визначених методів.

```
account := MailAccount new.
email := Email new subject: 'My first email'.
account saveAsDraft: email.
account fetchMail.
(MailClientPresenter on: account) open
```

Відкриється вікно з двома електронними листами. Якщо вибрати чернетку, то вікно виглядатиме як на рис. 12.4.

12.12. Підсумки розділу

Це був довгий розділ з великим прикладом з кількома моделями та демонстраторами. Він закладає основу для наступних розділів, де ми розширимо основний демонстратор і доробимо піддемонстратори, щоб пояснити більше функцій Spec.

Від перекладача. Читач міг помітити, що в цьому розділі вперше екземпляри демонстраторів створили за допомогою повідомлення *new*, а не *instantiate:* у методах *MailReaderPresenter >> initializePresenters* та *MailClientPresenter >> initializePresenters*. Мабуть, це виправдано в демонстраторі-обгортці *MailReaderPresenter*, бо він щоразу перемикає «активний» демонстратор, залежно від потреби. Але в *MailClientPresenter* варто було б використати *instantiate: on:* та *instantiate:*.

Розділ 13

Рядок меню, панель інструментів, рядок статусу і контекстні меню

Багато віконних застосунків мають рядок меню, в якому зібрано всі команди, які вони вміють виконувати. Вікно застосунку також може мати панель інструментів з кнопками для часто вживаних команд. Деякі програми мають лише панель інструментів. Крім підтримки рядка меню та панелі інструментів, Spec підтримує рядок стану внизу вікна. Деякі візуальні компоненти, наприклад, текстові поля, таблиці та списки, оснащені контекстними меню. Усі ці аспекти є предметом обговорення у цьому розділі.

Ми поліпшимо створену в розділі 12 клієнтську програму електронної пошти: додамо рядок меню, панель інструментів, рядок статусу та контекстне меню. На рис. 13.1 показано результат, якого треба досягти.

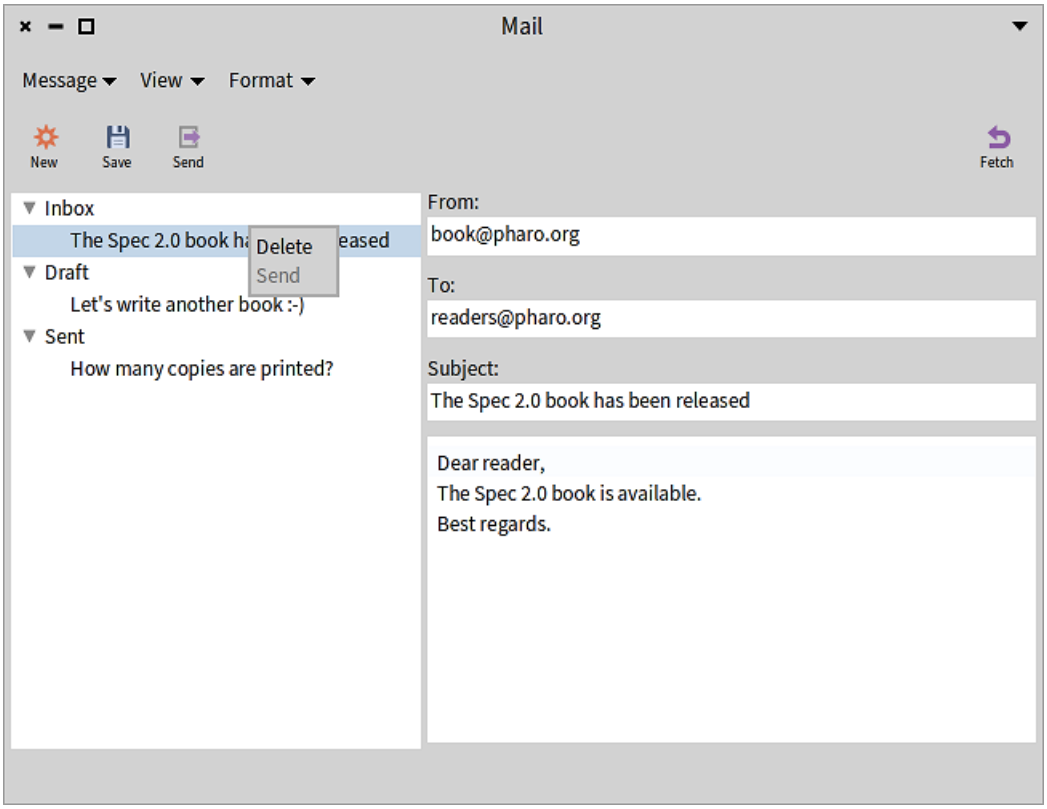


Рис. 13.1. Поштовий клієнт з меню та панеллю інструментів

13.1. Додавання до вікна рядка меню

З усіма моделями та демонстраторами, збудованими в попередньому розділі, можемо поринути у вивчення теми цього розділу. Пам'ятаймо, що весь код доступний у репозиторії, як було описано в розділі 1. Почнемо з додавання рядка меню з командами для опрацювання електронних листів.

Рядок меню є частиною демонстратора вікна, тому його налаштовують у методі *initializeWindow:*. Щоб задати рядок меню, екземплярові *SpWindowPresenter* надсилають повідомлення *menu:*.

```
MailClientPresenter >> initializeWindow: aWindowPresenter
   super initializeWindow: aWindowPresenter.
   aWindowPresenter
      title: 'Mail';
      initialExtent: 650@500;
      menu: menuBar
```

Змінна екземпляра *menuBar* ще не визначена, тому найперше це треба виправити. Додамо її до оголошення класу.

```
SpPresenterWithModel << #MailClientPresenter
   slots: { #account . #reader . #editedEmail . #menuBar };
   package: 'CodeOfSpec20Book'
```

Тоді треба присвоїти їй значення. Розширимо для цього метод *initializePresenters*. Він делегує таку відповідальність методу *initializeMenuBar*.

```
MailClientPresenter >> initializePresenters
   account := MailAccountPresenter on: self model.
   reader := MailReaderPresenter new.
   self initializeMenuBar

MailClientPresenter >> initializeMenuBar
   menuBar := self newMenuBar
      addItem: [ :item |
         item
            name: 'Message';
            subMenu: self messageMenu;
            yourself ];
      addItem: [ :item |
         item
            name: 'View';
            subMenu: self viewMenu;
            yourself ];
      addItem: [ :item |
         item
            name: 'Format';
            subMenu: self formatMenu;
            yourself ];
      yourself
```

Вираз *self newMenuBar* створює новий екземпляр *SpMenuBarPresenter*. До нього додаємо три пункти. Вони і будуть пунктами меню вікна застосунку. Для кожного з них задаємо назву та підменю.

13.2. Реалізація команд підменю «Message»

У цьому розділі реалізуємо лише команди підменю «Message». Два інші пункти меню, «View» і «Format», включені тільки для того, щоб показати в рядку меню кілька розділів. Методи *viewMenu* і *formatMenu* майже порожні й не роблять нічого, крім

створення пунктів підменю. Почнемо з тих розділів головного меню, які не будемо реалізовувати. Вони короткі.

```
MailClientPresenter >> viewMenu
   "Empty placeholder. Not defined in this chapter"
   ^ self newMenu
         addItem: [ :item | item name: 'Show CC field' ];
         addItem: [ :item | item name: 'Show BCC field' ];
         yourself

MailClientPresenter >> formatMenu
   "Empty placeHolder. Not defined in this chapter"
   ^ self newMenu
         addItem: [ :item | item name: 'Plain text' ];
         addItem: [ :item | item name: 'Rich text' ];
         yourself
```

Тепер можна зосередитися на командах підменю «Message». Реалізуємо їх усі. Для цього доведеться написати трохи програмного коду.

```
MailClientPresenter >> messageMenu
   ^ self newMenu
      addGroup: [ :group |
         group
            addItem: [ :item |
               item
                  name: 'New';
                  shortcut: $n meta;
                  action: [ self newMail ] ];
            addItem: [ :item |
               item
                  name: 'Save';
                  shortcut: $s meta;
                  enabled: [ self hasDraft ];
                  action: [ self saveMail ] ];
            addItem: [ :item |
               item
                  name: 'Delete';
                  shortcut: $d meta;
                  enabled: [ self hasSelectedEmail ];
                  action: [ self deleteMail ] ];
            addItem: [ :item |
               item
                  name: 'Send';
                  shortcut: $l meta;
                  enabled: [ self hasDraft ];
                  action: [ self sendMail ] ] ];
      addGroup: [ :group |
         group
            addItem: [ :item |
               item
                  name: 'Fetch';
                  shortcut: $f meta;
                  action: [ self fetchMail ] ];
            yourself ]
```

Це підменю більше від двох попередніх, які охоплювали по дві команди кожне. Воно ж містить кілька команд, згруповані у дві групи. Групу до меню додають за допомогою повідомлення *addGroup:*. Групі надсилають повідомлення *addItem:*, щоб вкласти в неї пункт меню. Видно, що пункти меню мають назву, комбінацію клавіш і блок дій. Кілька пунктів мають блок, який визначає, чи вони доступні. Блок-аргумент повідомлення *enabled:* обчислюється кожного разу, коли відображається пункт меню, тому доступний такий пункт, чи ні, з'ясовується динамічно. Зауважимо, що в тілі згаданих блоків надсилають повідомлення *hasDraft* і *hasSelectedEmail*. Відповідні методи ще не визначили, тож зробимо це зараз. Реалізації прості.

```
MailClientPresenter >> hasDraft
   ^ editedEmail isNotNil

MailClientPresenter >> hasSelectedEmail
   ^ account hasSelectedEmail
```

Подивіться на клавіатурні скорочення пунктів меню в методі *messageMenu*. *$n meta* означає, що, щоб запустити команду, можна натиснути [*n*] разом із метаклавішею ([*Command*] у macOS, або [*Control*] у Windows і Linux).

13.3. Встановлення гарячих клавіш

Додавання клавіатурних скорочень (гарячих клавіш) до пунктів меню не встановлює їх автоматично. Комбінації клавіш встановлюють після відкриття вікна, тому доведеться доповнити метод *initializeWindow:*. Тут у нагоді стане метод *addKeybindingsTo:*, реалізований в *SpMenuPresenter*, надкласі *SpMenuBarPresenter*.

```
MailClientPresenter >> initializeWindow: aWindowPresenter
   super initializeWindow: aWindowPresenter.
   aWindowPresenter
      title: 'Mail';
      initialExtent: 650@500;
      menu: menuBar.
   menuBar addKeybindingsTo: aWindowPresenter
```

13.4. Визначення дій

Завдяки надсиланню повідомлень блоки дій пунктів меню вдалося визначити просто. Звичайно, відповідні методи треба реалізувати, то зробімо це.

Створення, зберігання, надсилання електронного листа впливає на стан моделі застосунку. Доступ до моделі має екземпляр *MailAccountPresenter*, збережений у змінній *account*. Пригадуємо з попереднього розділу, що демонстратор облікового запису не може похвалитися багатим функціоналом: він уміє тільки повідомляти вибір користувача. Настав час навчити його робити значно більше[15].

```
MailAccountPresenter >> saveAsDraft: draftEmail
   self model saveAsDraft: draftEmail.
   self modelChanged
```

[15] Доповнення коду класу *MailAccountPresenter* і пояснення щодо нього зробив перекладач книги на основі репозиторію книги (прим. – *Ярошко С.*).

```
MailAccountPresenter >> deleteMail
   | pathIndexes email |
   pathIndexes := foldersAndEmails selection selectedPath.
   email := foldersAndEmails itemAtPath: pathIndexes.
   self model delete: email.
   self modelChanged

MailAccountPresenter >> sendMail: draftEmail
   self model send: draftEmail.
   self modelChanged

MailAccountPresenter >> fetchMail
   self model fetchMail.
   self modelChanged
```

Бачимо, що кожен з методів делегує роботу моделі й оновлює стан демонстратора відповідно до її змін. Тепер реалізація дій пунктів меню досить очевидна.

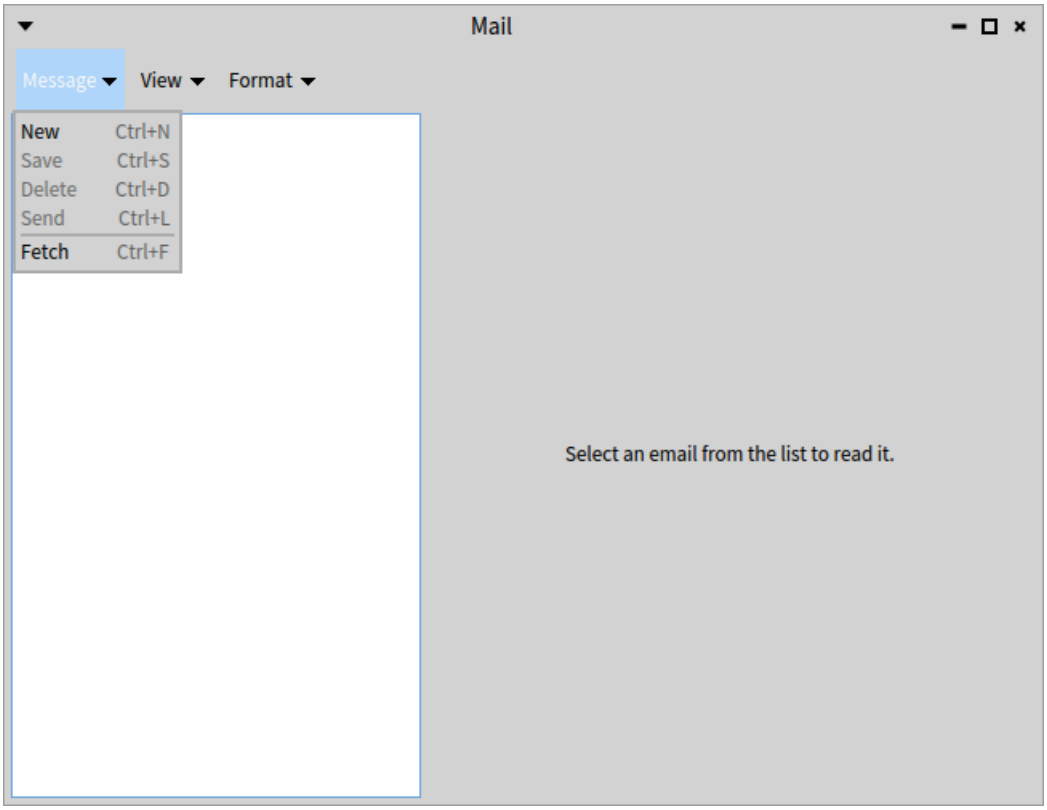


Рис. 13.2. Поштовий клієнт з розгорнутим підменю

```
MailClientPresenter >> newMail
   editedEmail := Email new.
   reader updateLayoutForEmail: editedEmail.

MailClientPresenter >> saveMail
   account saveAsDraft: editedEmail.
   editedEmail := nil.

MailClientPresenter >> deleteMail
   account deleteMail.

MailClientPresenter >> sendMail
```

```
        account sendMail: editedEmail.
        editedEmail := nil.

    MailClientPresenter >> fetchMail
        account fetchMail.
```

Настав час випробувати застосунок. Щоб побачити рядок меню в дії, відкриємо вікно.

```
    (MailClientPresenter on: MailAccount new) open
```

На рис. 13.2 зображене вікно застосунку. Рядок меню містить три визначені пункти меню. Видно відкрите підменю «Message». Воно складається з двох груп пунктів меню, розділених горизонтальною лінією. Доступні два з них. Три інші недоступні, бо вони виконують дії з електронними листами, а ні одного листа не вибрано.

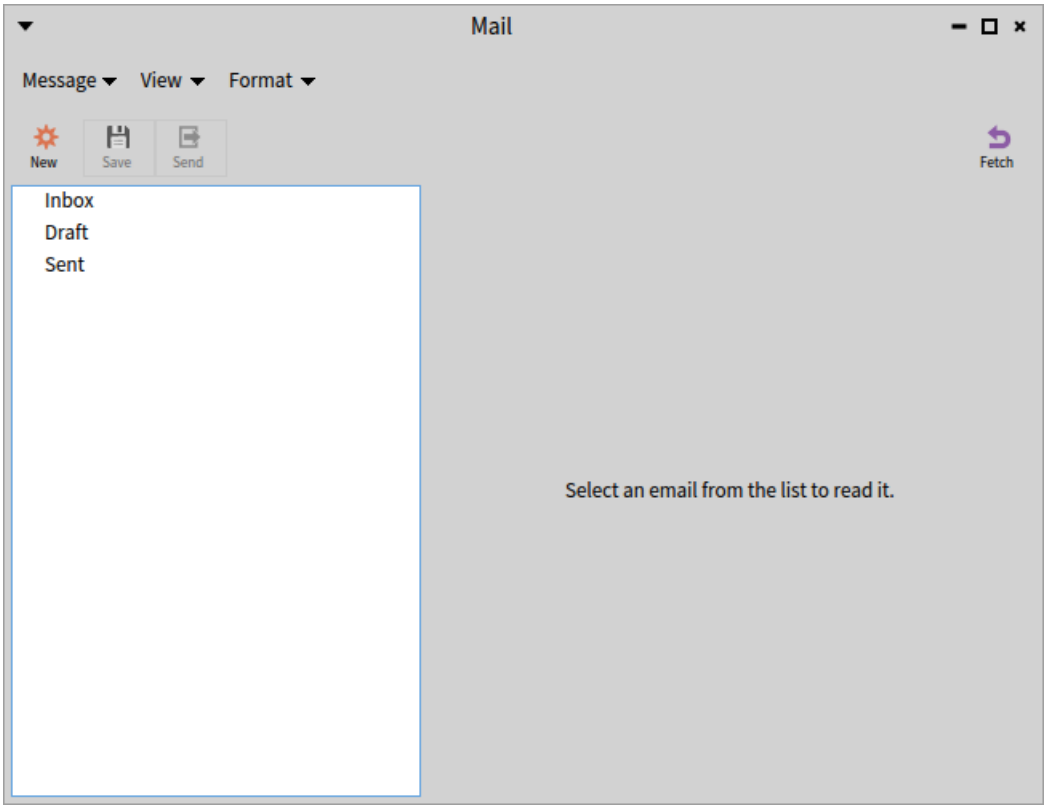


Рис. 13.3. Поштовий клієнт з панеллю інструментів (окремі кнопки на ній деактивовані)

13.5. Додавання панелі інструментів

Деякі дії так часто вживані, що корисно мати їх на відстані одного клацання. Саме тут на арені з'являється панель інструментів. Панель інструментів дає змогу розміщувати дії як кнопки в інтерфейсі користувача (див. рис. 13.3).

Не дивно, що панель інструментів, як і рядок меню, є частиною демонстратора вікна. Тому доведеться ще раз переглянути метод *initializeWindow:*. Екземпляр *SpWindow-Presenter* розуміє повідомлення *toolbar:* для встановлення панелі інструментів.

```
    MailClientPresenter >> initializeWindow: aWindowPresenter
        aWindowPresenter
            title: 'Mail';
            initialExtent: 650@500;
```

```
            menu: menuBar;
            toolbar: toolBar.
         menuBar addKeybindingsTo: aWindowPresenter
```

toolbar – це змінна екземпляра, тому треба доповнити визначення класу.

```
      SpPresenterWithModel << #MailClientPresenter
         slots: { #account . #reader . #editedEmail . #menuBar . #toolBar };
         package: 'CodeOfSpec20Book'
```

Рядок меню налаштовує окремий метод. Оголосимо такий і для панелі інструментів, метод *initializeToolBar*, і використаємо його в *initializePresenters*.

```
      MailClientPresenter >> initializePresenters
         account := MailAccountPresenter on: self model.
         reader := MailReaderPresenter new.
         self initializeMenuBar.
         self initializeToolBar

      MailClientPresenter >> initializeToolBar
         | newButton fetchButton |
         newButton := self newToolbarButton
            label: 'New';
            icon: (self iconNamed: #smallNew);
            help: 'New email';
            action: [ self newMail ];
            yourself.
         saveButton := self newToolbarButton
            label: 'Save';
            icon: (self iconNamed: #smallSave);
            help: 'Save email';
            action: [ self saveMail ];
            yourself.
         sendButton := self newToolbarButton
            label: 'Send';
            icon: (self iconNamed: #smallExport);
            help: 'Send email';
            action: [ self sendMail ];
            yourself.
         fetchButton := self newToolbarButton
            label: 'Fetch';
            icon: (self iconNamed: #refresh);
            help: 'Fetch emails from server';
            action: [ self fetchMail ];
            yourself.
         toolBar := self newToolbar
            addItem: newButton;
            addItem: saveButton;
            addItem: sendButton;
            addItemRight: fetchButton;
            yourself.
         fetchButton click
```

Цей метод визначає чотири кнопки, дві з яких зберігаються в змінних екземпляра. Незабаром стане зрозуміло, чому. Що ж, доведеться знову адаптувати визначення класу.

```
SpPresenterWithModel << #MailClientPresenter
   slots: { #account . #reader . #editedEmail . #menuBar . #toolBar .
      #sendButton . #saveButton };
   package: 'CodeOfSpec20Book'
```

Метод *initializeToolBar* додає на панель інструментів чотири кнопки. Всяка панель інструментів містить дві частини: одну – біля лівого краю, на початку панелі, іншу – біля правого краю, наприкінці. За допомогою повідомлення *addItem:* перші три кнопки додаємо до частини ліворуч, а четверту кнопку за допомогою *addItemRight:* – до частини праворуч. На завершення методу ініційовано клацання на кнопці отримання пошти з сервера, щоб застосунок відкрився відразу з актуальним вмістом папки вхідних листів.

Кожна кнопка має напис, піктограму, текст підказки і дію. Як і в *initializeMenuBar*, блоки дій надсилають повідомлення демонстратору поштового клієнта. Це ті ж повідомлення, які використовували в блоках дій пунктів підменю «Message» головного меню. Це означає, що ми закінчили.

13.6. Керування доступністю кнопок

Ми сказали, що закінчили, але, насправді, ще ні. Пункти меню отримували повідомлення з блоком для визначення стану: доступні вони чи ні. Це не стосується кнопок панелі інструментів, адже їх видно весь час. Тому потрібно явно керувати доступністю кнопок. Кожного разу, коли змінюється стан поштового клієнта, треба оновлювати активацію кнопок панелі інструментів. Для цього запровадимо новий метод *updateToolBarButtons*. За допомогою визначених раніше повідомлень можна встановити стан доступності *saveButton* і *sendButton*. Ось чому ці кнопки зберегли в змінних екземпляра. Дві інші кнопки завжди ввімкнені, тому їх не потрібно утримувати в змінних екземпляра.

```
MailClientPresenter >> updateToolBarButtons
   | hasSelectedDraft |
   hasSelectedDraft := self hasDraft.
   saveButton enabled: hasSelectedDraft.
   sendButton enabled: hasSelectedDraft
```

Щоб завершити розробку функціонала панелі інструментів, потрібно у відповідних місцях надіслати *updateToolBarButtons*: усюди, де змінюється стан демонстратора поштового клієнта. Їх не так уже й багато.

По-перше, *MailClientPresenter* наслідує *SpPresenterWithModel*, тому кожного разу, коли змінюється модель екземпляра, вона надсилає повідомлення *modelChanged*. Отже, можна оновити кнопки панелі інструментів цим методом.

```
MailClientPresenter >> modelChanged
   self updateToolBarButtons
```

По-друге, треба встановити початковий стан кнопок панелі інструментів під час ініціалізації презентатора поштового клієнта. Метод *updateAfterSelectionChangedTo:*, викликаний у тілі методу *connectPresenters*, є хорошим місцем для цього. Додамо один рядок у нижній частині методу, який визначили раніше.

```
MailClientPresenter >> updateAfterSelectionChangedTo: selectedFolderOrEmail

   editedEmail := (self isDraftEmail: selectedFolderOrEmail)
```

```
            ifTrue: [ selectedFolderOrEmail ]
            ifFalse: [ nil ].
        self updateToolBarButtons
```

На стан кнопок впливатиме також виконання команд меню, тому доведеться додати рядок *self updateToolBarButtons* у кожен з методів реагування на вибір пункту меню: *newMail*, *saveMail*, *deleteMail*, *sendMail*, *fetchMail*.

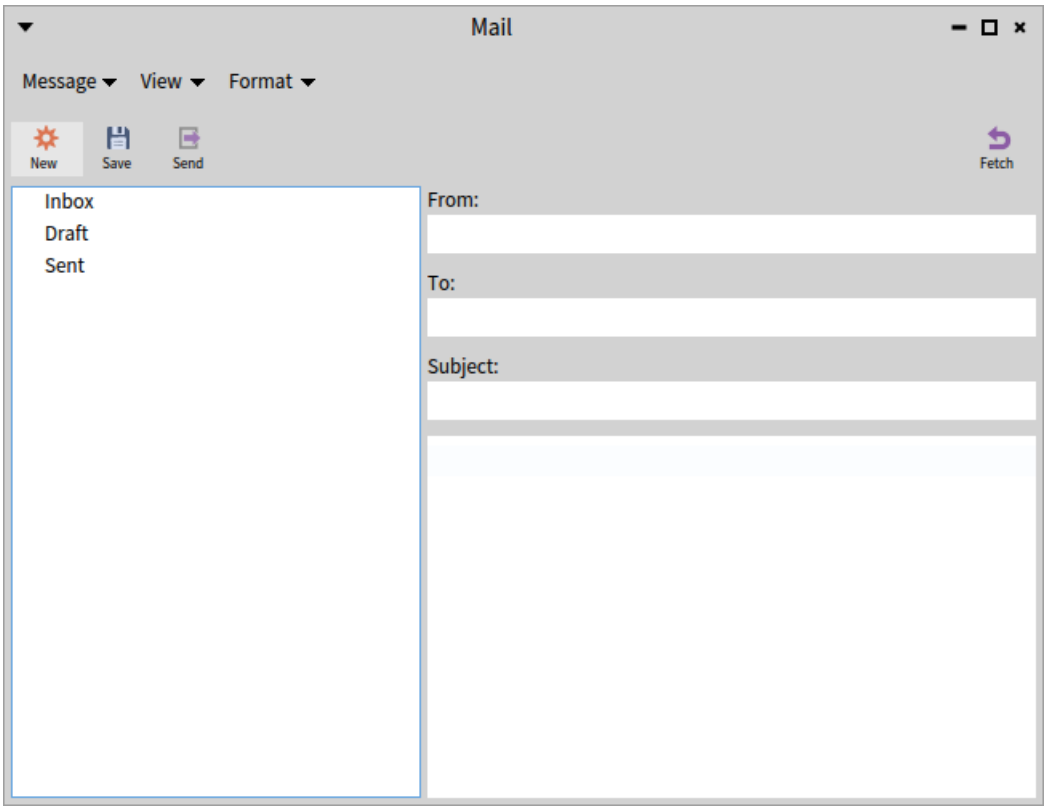


Рис. 13.4. Поштовий клієнт з активними кнопками на панелі інструментів

Для побудови і налаштування панелі інструментів, як і для головного меню, потрібно було багато коду, але все готове. Відкриймо вікно знову.

```
    (MailClientPresenter on: MailAccount new) open
```

На рис. 13.3. зображено вікно застосунку. Воно має рядок меню і панель інструментів. Три кнопки панелі розташовані ліворуч, а одна– праворуч. Це відповідає заданій конфігурації панелі інструментів. Кнопки ***Save*** та ***Send*** мають сірий колір, бо вони вимкнені.

Створимо новий електронний лист, натиснувши кнопку ***New*** на панелі інструментів, і подивимося, як змінюється стан кнопок на панелі. На рис. 13.4 показано, що всі кнопки увімкнулися – стали доступними.

13.7. Додавання рядка стану

Після додавання рядка меню та панелі інструментів додамо рядок стану (див. рис. 13.5, 13.6). Рядок стану корисний для показу коротких повідомлень протягом деякого часу або до появи наступного повідомлення. Ми доповнимо демонстратор поштового клієнта здатністю показувати повідомлення, які інформуватимуть користувача про виконані дії.

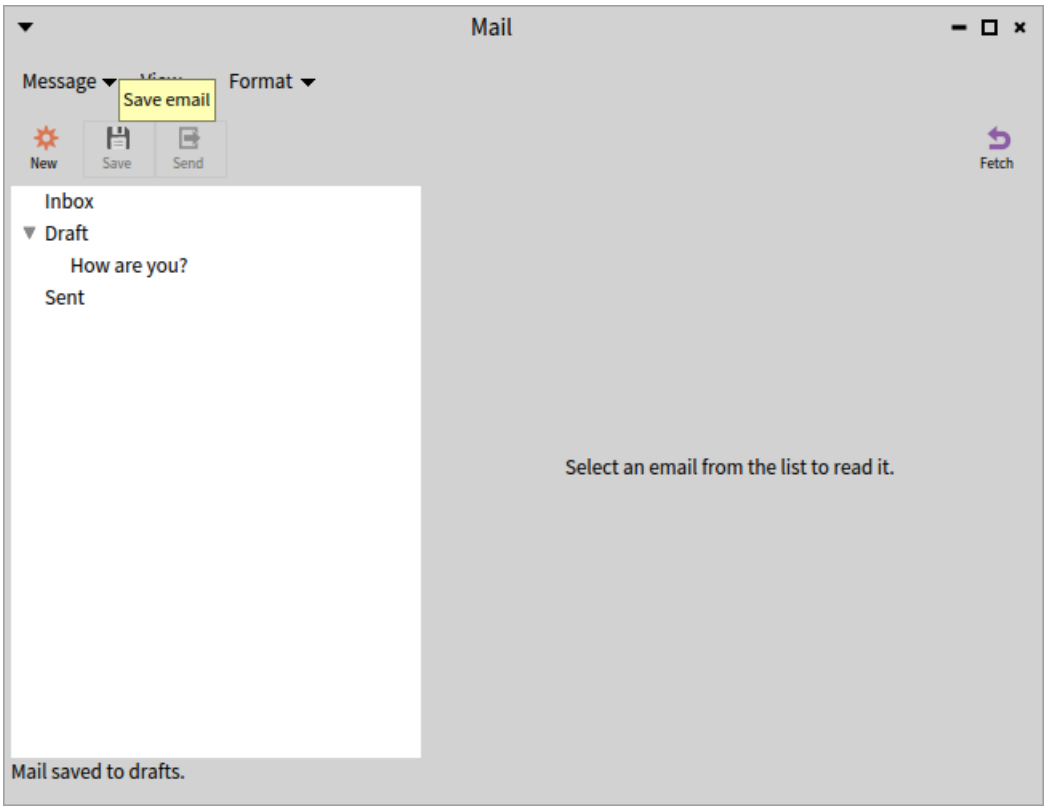


Рис. 13.5. Електронного листа збережено

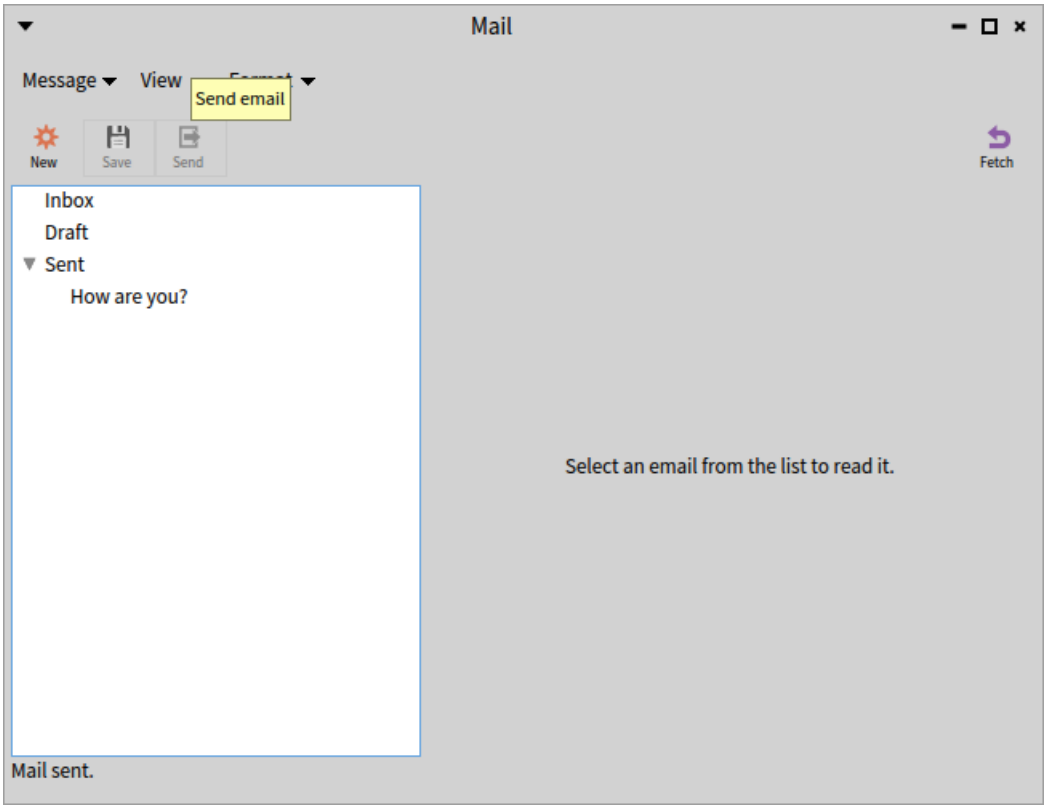


Рис. 13.6. Електронного листа надіслано

Рядок стану з’являється внизу вікна. Як і рядок меню та панель інструментів, його додають у методі *initializeWindow:*.

```
MailClientPresenter >> initializeWindow: aWindowPresenter
   aWindowPresenter
      title: 'Mail';
      initialExtent: 650@500;
      menu: menuBar;
      toolbar: toolBar;
      statusBar: statusBar.
   menuBar addKeybindingsTo: aWindowPresenter
```

statusBar – це нова змінна екземпляра, яку треба додати до визначення класу демонстратора.

```
SpPresenterWithModel << #MailClientPresenter
   slots: { #account . #reader . #editedEmail . #menuBar . #toolBar .
      #sendButton . #saveButton . #statusBar };
   package: 'CodeOfSpec20Book'
```

Як уже було зроблено двічі, доповнимо метод *initializePresenters*. Повідомлення *newStatusBar* створює новий екземпляр *SpStatusBarPresenter*.

```
MailClientPresenter >> initializePresenters
   account := MailAccountPresenter on: self model.
   reader := MailReaderPresenter new.
   statusBar := self newStatusBar.
   self initializeMenuBar.
   self initializeToolBar
```

Рядок стану – це не більше ніж контейнер для текстового повідомлення. Пристосуємо деякі методи реагування на команди меню для надсилання тексту в рядок стану. Екземпляр *SpStatusBarPresenter* відповідає на *pushMessage:* і *popMessage*. Почнемо з методу *fetchMail*. Перешлемо рядок «Mail fetched.», щоб зазначити, що операція отримання пошти була успішною.

```
MailClientPresenter >> fetchMail
   account fetchMail.
   self updateToolBarButtons.
   statusBar pushMessage: 'Mail fetched.'
```

Подібно доповнимо й інші методи реагування.

```
MailClientPresenter >> newMail
   editedEmail := Email new.
   reader updateLayoutForEmail: editedEmail.
   self updateToolBarButtons.
   statusBar pushMessage: 'Ready to write a new message'

MailClientPresenter >> saveMail
   account saveAsDraft: editedEmail.
   editedEmail := nil.
   self updateToolBarButtons.
   statusBar pushMessage: 'Mail saved to drafts.'

MailClientPresenter >> deleteMail
   account deleteMail.
   self updateToolBarButtons.
```

```
        statusBar pushMessage: 'Mail deleted.'

    MailClientPresenter >> sendMail
        account sendMail: editedEmail.
        editedEmail := nil.
        self updateToolBarButtons.
        statusBar pushMessage: 'Mail sent.'
```

Щоб завершити розробку функціонала рядка стану, потурбуємося про його початковий вигляд. Одразу після запуску застосунку рядок стану мав би бути порожній. Тому знову адаптуємо метод *updateAfterSelectionChangedTo:*. Цього разу надішлемо повідомлення *popMessage*, щоб переконатися, що рядок стану порожній.

```
    MailClientPresenter >> updateAfterSelectionChangedTo: selectedFolderOrEmail
        editedEmail := (self isDraftEmail: selectedFolderOrEmail)
            ifTrue: [ selectedFolderOrEmail ]
            ifFalse: [ nil ].
        self updateToolBarButtons.
        statusBar popMessage
```

Протестуємо демонстратор поштового клієнта, відкривши його ще раз.

```
    (MailClientPresenter on: MailAccount new) open
```

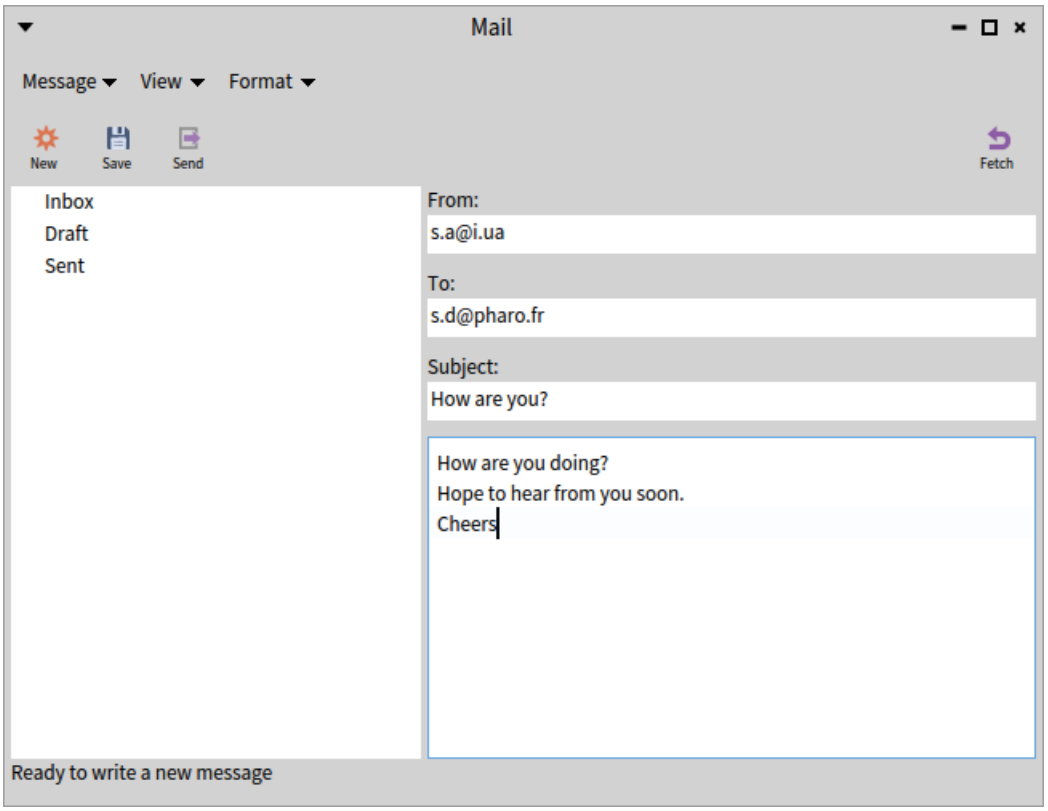


Рис. 13.7. Новий електронний лист

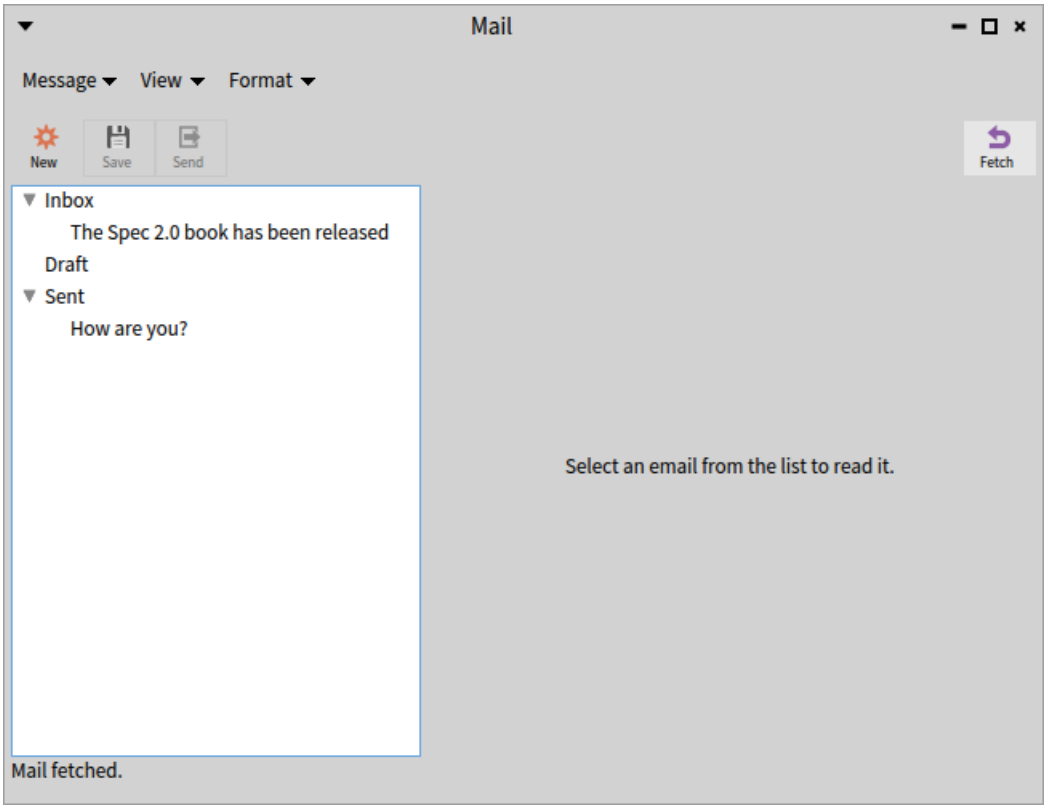


Рис. 13.8. Електронну пошту отримано

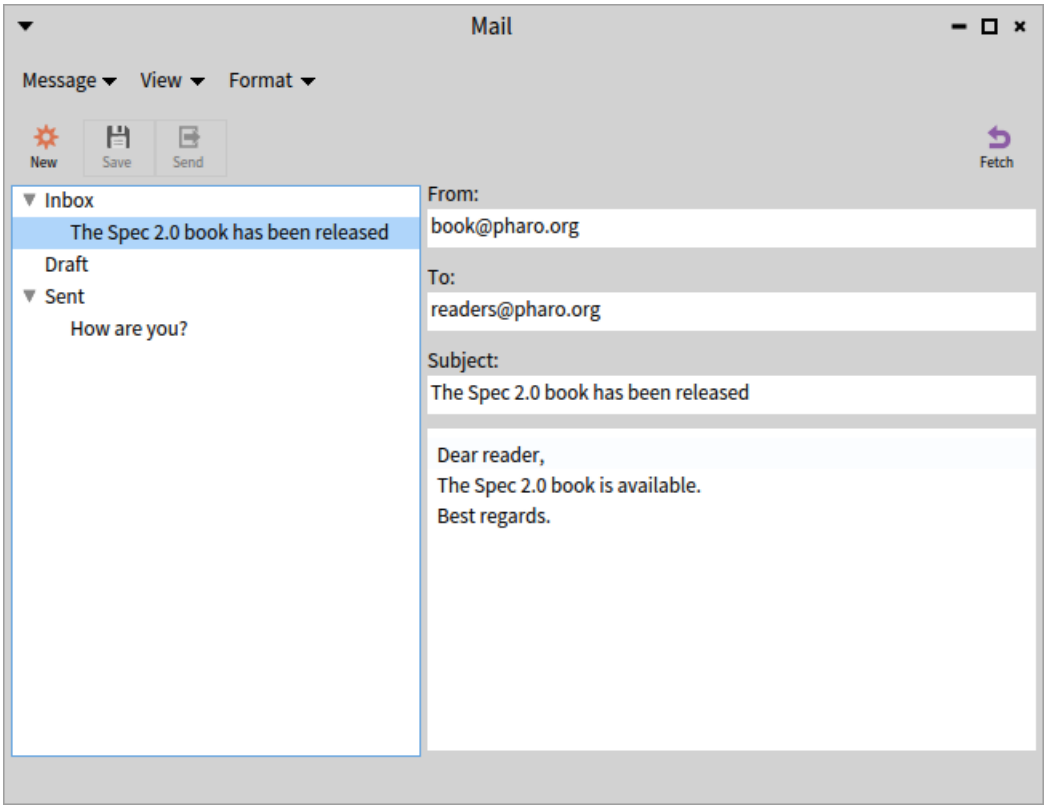


Рис. 13.9. Вибрано листа, рядок стану спорожнів

Перевіримо повний сценарій.

- Відкрийте вікно застосунку, натисніть кнопку ***New*** і заповніть поля нового електронного листа. На рис. 13.7 показано початковий стан програми перед тим, як виконати інші дії з листом.
- Коли поля заповнено, збережемо електронний лист, натиснувши кнопку ***Save*** (див. рис. 13.5). У рядку стану відображено «Mail saved to drafts.», і видно тему електронного листа, вкладену в папку «Draft» у списку ліворуч.
- Виберіть електронний лист у списку та натисніть кнопку ***Send***. Ситуація мала б стати такою, як на рис. 13.6. Лист переміщено з папки «Draft» до папки «Sent», а в рядку стану відображено «Mail sent.».
- Після натискання кнопки ***Fetch*** отриманий електронний лист з'являється в папці «Inbox». Рядок стану відображає «Mail fetched.» (див. рис. 13.8).
- Виберіть отриманий електронний лист в папці «Inbox». Демонстратор праворуч відобразить всі поля листа, а рядок стану спорожніє (див. рис. 13.9).
- Тепер виберіть команду «*Delete*» в меню «*Message*» (або натисніть комбінацію [*Ctrl* + *D*]) – електронний лист буде видалено зі списку, а в рядку стану відобразиться «Mail deleted.» (див. рис. 13.10).

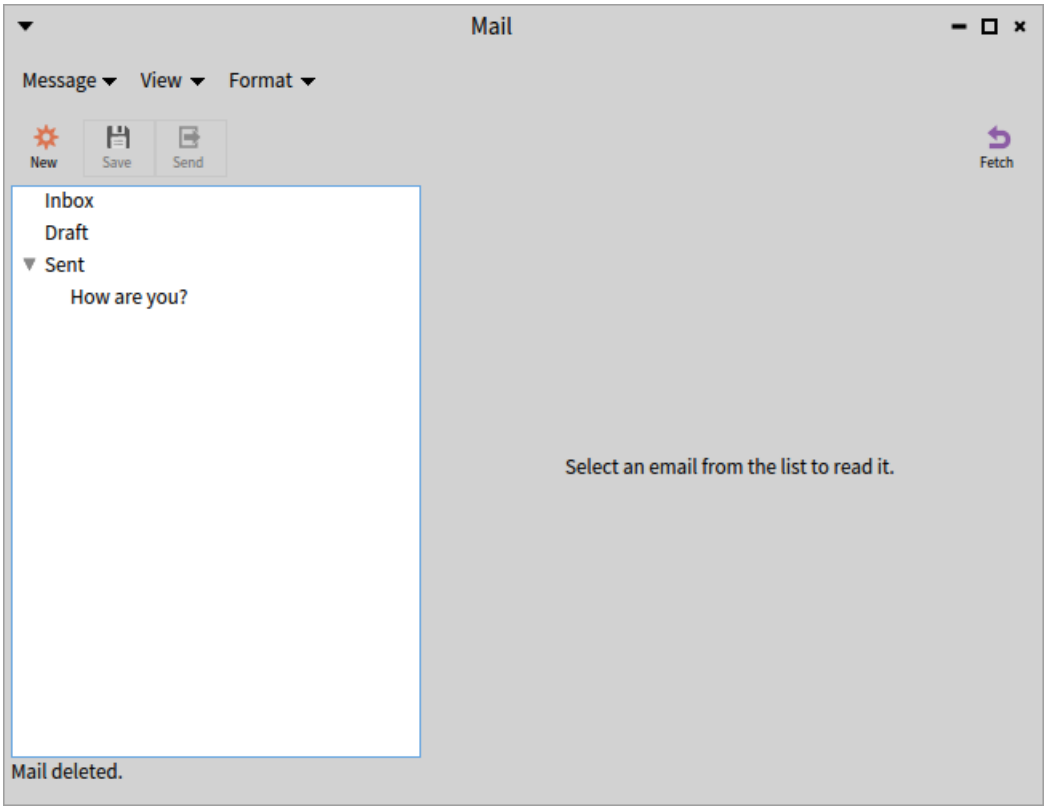


Рис. 13.10. Електронного листа видалено

Усі дії, які змінюють рядок стану, перевірено.

13.8. Додавання до демонстратора контекстного меню

Останнім кроком до завершення демонстратора поштового клієнта є додавання контекстного меню. Додамо його до дерева з папками та електронними листами. Не будемо робити велике меню. Для демонстрації достатньо обмежитися двома пунктами: один – для видалення електронного листа, інший – для надсилання.

Дерево містить папки та електронні листи, тому, коли вибрано папку, згадані пункти меню треба вимкнути. Їх також треба вимкнути, коли не зроблено жодного вибору. Є ще одна умова: команду надсилання можна застосувати лише до чернеток електронних листів, оскільки отримані та надіслані листи не можна надсилати.

Зазвичай контекстне меню вкладеному демонстратору додає його батьківський демонстратор. Демонстратор дерева папок і листів вкладений в *MailAccountPresenter*, тому можна було б сподіватися, що контекстне меню на демонстратор дерева встановить саме він. Проте *MailAccountPresenter* не знає, як видалити чи надіслати електронного листа. За ці дії відповідає *MailClientPresenter*, який визначає методи *deleteMail* і *sendMail*. Обидва методи роблять те, що вони повинні зробити, щоб виконати дію, а потім оновлюють стан кнопок панелі інструментів і рядок стану.

Тому контекстне меню визначатиме *MailClientPresenter*.

```
MailClientPresenter >> accountMenu
   ^ self newMenu
         addItem: [ :item |
            item
               name: 'Delete';
               enabled: [ self hasSelectedEmail ];
               action: [ self deleteMail ] ];
         addItem: [ :item |
            item
               name: 'Send';
               enabled: [ self hasSelectedEmail
                                 and: [ account selectedItem isDraft] ];
               action: [ self sendMail ] ];
         yourself
```

Блоки дій такі ж прості, як і блоки дій пунктів головного меню чи кнопок на панелі інструментів. Вони надсилають повідомлення *deleteMail* і *sendMail*, а відповідні методи визначені раніше.

13.9. Блоки перевірки доступності

Цікавішими за блоки дій є блоки *enabled:*, які визначають доступність пунктів меню. Видалення електронного листа можливе лише тоді, коли вибрано листа. Це виражає блок, аргумент повідомлення *enabled:* пункту меню «Delete». Як вже було сказано, надіслати можна тільки чернетку листа. Це саме те, що виражає блок *enabled:* для пункту меню «Send».

Зверніть увагу на назву методу. Використано *accountMennu*, бо *MailClientPresenter* встановить контекстне меню на вкладений *MailAccountPresenter*. Але його треба встановити на демонстратор дерева. Тому *MailAccountPresenter* передає меню далі – демонстратору дерева. Знайдімо це в коді. Спочатку в тілі методу *initializePresenters* класу *MailClient-Presenter* надсилають повідомлення *contextMenu:*, щоб встановити контекстне меню на *MailAccountPresenter*.

```
MailClientPresenter >> initializePresenters
   account := MailAccountPresenter on: self model.
   account contextMenu: [ self accountMenu ].
   reader := MailReaderPresenter new.
   statusBar := self newStatusBar.
```

```
        self initializeMenuBar.
        self initializeToolBar
```

Потім визначимо *contextMenu:* в класі *MailAccountPresenter*. Він делегує демонстратору дерева.

```
MailAccountPresenter >> contextMenu: aBlock
   foldersAndEmails contextMenu: aBlock
```

На цьому реалізація завершується. Настав час знову відкрити вікно та випробувати нове контекстне меню.

```
(MailClientPresenter on: MailAccount new) open
```

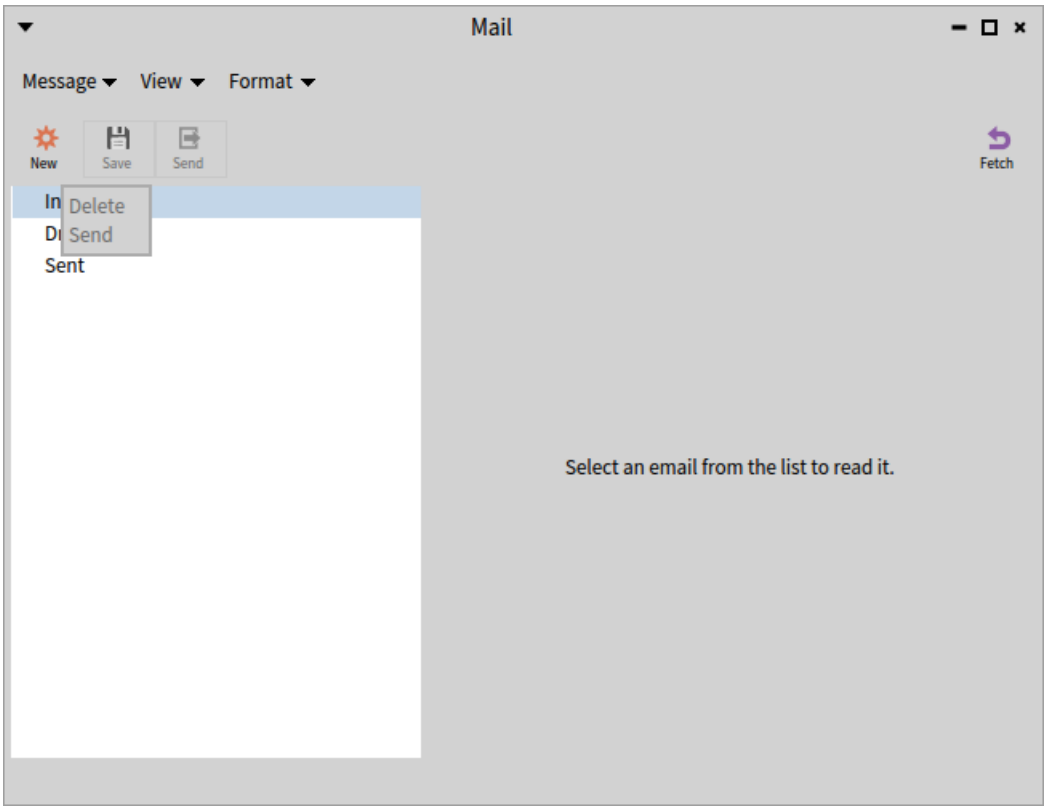


Рис. 13.11. Пункти контекстного меню недоступні

Клацання правою кнопкою мишки викликає контекстне меню. На рис. 13.11 видно, що обидва пункти меню вимикаються, коли вибрано папку.

Після отримання електронної пошти та вибору отриманого листа меню містить увімкнений пункт «Delete» та вимкнений «Send», як показано на рис. 13.12.

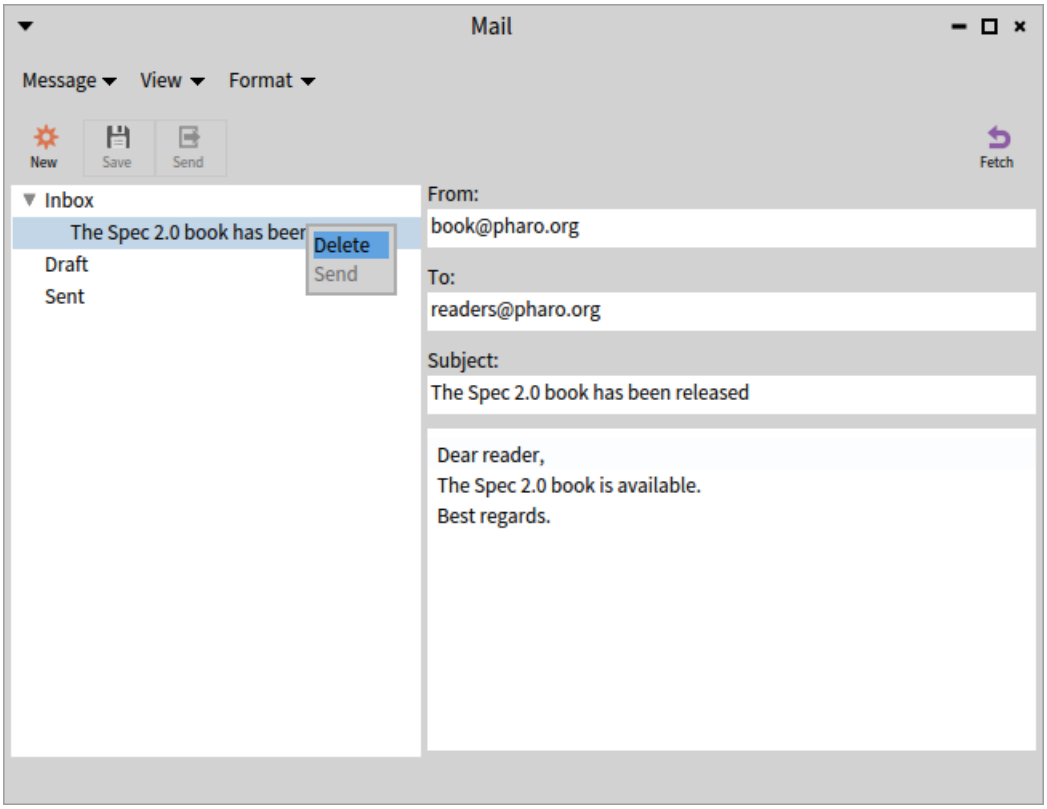


Рис. 13.12. Отриманого листа можна вилучити, але не можна надіслати

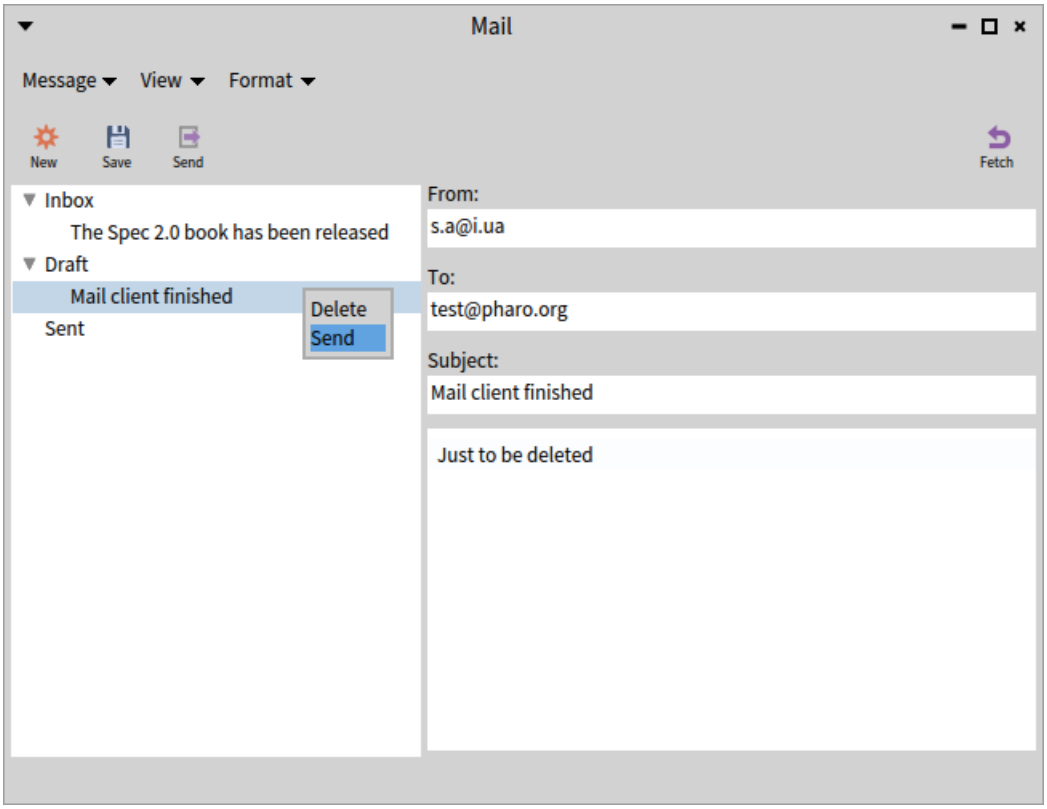


Рис. 13.13. Чернетку можна надіслати

На завершення випробувань створимо новий електронний лист, збережемо його та виберемо зі списку. Це чернетка листа, тому її можна надіслати. Що й видно в контекстному меню на рис. 13.13. Обидва пункти меню активні.

13.10. Підсумки розділу

У розділі описано, як додати до вікна рядок меню, панель інструментів і рядок стану. Щоб визначити пункти меню та кнопки панелі інструментів, знадобилося чимало коду. Описано, як окремі методи можуть відображати повідомлення в рядку стану внизу вікна. Наприкінці розділу описано, як додати контекстне меню до демонстратора дерева.

Важливим аспектом налаштування пунктів меню та кнопок панелі інструментів є їхня активація на основі стану демонстратора. Показано, як програмно керувати доступністю компонентів, запрограмовану поведінку проілюстровано багатьма рисунками.

Розділ 14
Використання портів і перенесень

У цьому розділі описано перенесення, які надають компактніший спосіб поєднання демонстраторів, ніж події, про які йшлося в попередніх розділах. Різні аспекти перенесень будуть пояснені на окремих прикладах.

14.1. Що таке перенесення?

Перенесення – це однорідний спосіб зв'язку демонстраторів, який більше враховує «потік» даних, ніж спосіб їхнього відображення.

Кожен демонстратор визначає **вихідні порти**, які надсилають дані, та **вхідні порти**, які їх отримують.

Якщо взаємодія з демонстратором неможлива, то він не матиме вихідного порту (наприклад, *SpLabelPresenter*). Деякі демонстратори не мають вхідного порту (наприклад, *SpMenuPresenter*). Якщо не визначити вхідні та вихідні порти для своїх демонстраторів, то вони не матимуть ніяких портів.

Якщо в демонстратора є вхідні та вихідні порти, то він визначає, який один вхідний і один вихідний усталені.

Існують різні класи портів. Якщо не знайдеться підхожого класу порту для вашого демонстратора, то можна визначити власний.

Перенесення з'єднує вихідний порт одного демонстратора з вхідним портом іншого. Коли використовують перенесення, то замість того, щоб думати про події, та як на них реагувати, думають про те, як дані перетікають від вихідного порту до вхідного. Про обробку подій турбуються вихідні порти.

14.2. Простий приклад

Погляньмо на один простий приклад. Створимо демонстратор, який одночасно показує огляд (колекції) і деталізацію (її елемента). Визначимо клас *OverviewDetailPresenter* з двома змінними екземпляра для зберігання *SpListPresenter* і *SpTextPresenter*.

```
SpPresenter << #OverviewDetailPresenter
   slots: { #overview . #detail };
   package: 'CodeOfSpec20Book'
```

Наповнимо список деякими екземплярами *Point*.

```
OverviewDetailPresenter >> initializePresenters
   overview := self newList
      items: { 1@1 . 7@5 . 10@15 . 12@0 . 0@ -9 . -5@ -5 };
      yourself.
   detail := self newText
```

Метод *defaultLayout* простий. Він визначає послідовний горизонтальний макет.

```
OverviewDetailPresenter >> defaultLayout
   ^ SpBoxLayout newLeftToRight
      add: overview expand: false;
      add: detail;
      yourself
```

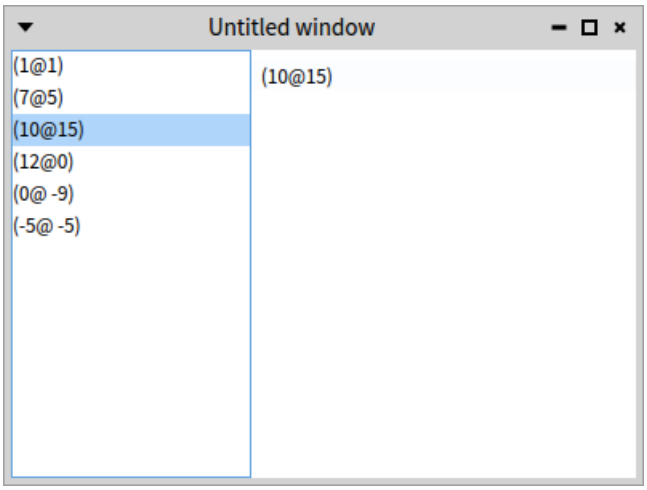


Рис. 14.1. Спрощений демонстратор «огляд-деталізація»

Ось найцікавіше. Метод *connectPresenters* пов'язує список з текстовим вікном. Почнемо з простої логіки, яка використовує події, а не перенесення. Коли в списку вибрали точку, то просто покажемо її в текстовому вікні.

```
OverviewDetailPresenter >> connectPresenters
   overview whenSelectedItemChangedDo: [ :selectedPoint |
      detail text: selectedPoint asString ]
```

Якщо відкрити демонстратор за допомогою фрагмента коду *OverviewDetailPresenter new open*, то вікно виглядатиме як на рис. 14.1.

14.3. Основа перенесення

Показаний вище метод *connectPresenters* використовує традиційний спосіб поєднання демонстраторів. Використаємо натомість перенесення.

```
OverviewDetailPresenter >> connectPresenters
   overview transmitTo: detail
```

Найпростіший спосіб налаштувати перенесення – використати метод *transmitTo:*. Його оголошено так.

```
SpAbstractPresenter >> transmitTo: aPresenter
   ^ self defaultOutputPort transmitTo: aPresenter defaultInputPort
```

У нашому прикладі показаний на початку параграфа метод з'єднує усталений вихідний порт демонстратора списку з усталеним вхідним портом демонстратора тексту. Методи *SpAbstractPresenter >> defaultOutputPort* і *SpAbstractPresenter >> defaultInputPort* визначають, що будь-який демонстратор може мати усталений вихідний і вхідний порт. Знайдіть і перегляньте класи, в яких реалізовані ці два методи, щоб дізнатися, як різні класи демонстраторів використовують вихідні та вхідні порти.

Демонстратори можуть мати кілька вихідних і вхідних портів. Їх можна з'єднати, надіславши повідомлення *transmitTo:* до будь-якого вихідного порту, подібно до того, що *SpAbstractPresenter >> transmitTo:* робить з усталеними вихідним і вхідним портами.

Коли знову відкрити демонстратор і вибрати точку в списку, виникне виняток. Це пов'язано з тим, що екземпляр *Point*, переданий із усталеного вихідного порту демонстратора списку, не сумісний із типом об'єкта, якого очікує усталений вхідний порт демонстратора тексту. Він очікує *String*, а не *Point*. Це поширена ситуація. Лише у простих випадках переданий об'єкт задовольнятиме вимоги вхідного порту. У багатьох випадках переданий об'єкт потрібно трансформувати, щоб передати адекватний об'єкт до вхідного порту. Ось тут і розпочинаються перетворення.

14.4. Перетворення перенесеного об'єкта

Об'єкт, переданий із вихідного порту демонстратора, може бути невідповідним для вхідного порту іншого демонстратора. Для цього є дві причини.

- Тип об'єкта, що надходить із вихідного порту, може бути неприйнятним для вхідного порту. У нашому простому прикладі це так і є. Вхідний порт очікує на *String*, а не на *Point*.
- Сам об'єкт, що надходить із вихідного порту, може бути не таким, який би хотілося надіслати до вхідного порту.

Наведемо приклади обох випадків.

Щоб виправити виняток, який виникає під час вибору точки в списку, доповнимо метод *connectPresenters* так, щоб порт надсилав *String* замість *Point*.

```
OverviewDetailPresenter >> connectPresenters
   overview
      transmitTo: detail
      transform: [ :selectedPoint | selectedPoint asString ]
```

Тепер демонстратор поводиться правильно.

Припустімо, що виникла потреба показувати в тексті не просто вибрану точку, а відстань від неї до початку системи координат. Традиційний метод *connectPresenters* виглядатиме так.

```
OverviewDetailPresenter >> connectPresenters
   overview whenSelectedItemChangedDo: [ :selectedPoint |
      | distanceToOrigin |
      distanceToOrigin := selectedPoint
         ifNil: [  ]
         ifNotNil: [ (selectedPoint distanceTo: 0@0) asString ].
      detail text: distanceToOrigin ]
```

З використанням перенесень його можна переписати так.

```
OverviewDetailPresenter >> connectPresenters
   overview transmitTo: detail
      transform: [ :selectedPoint |
         selectedPoint
            ifNil: [  ]
            ifNotNil: [ (selectedPoint distanceTo: 0@0) asString ] ]
```

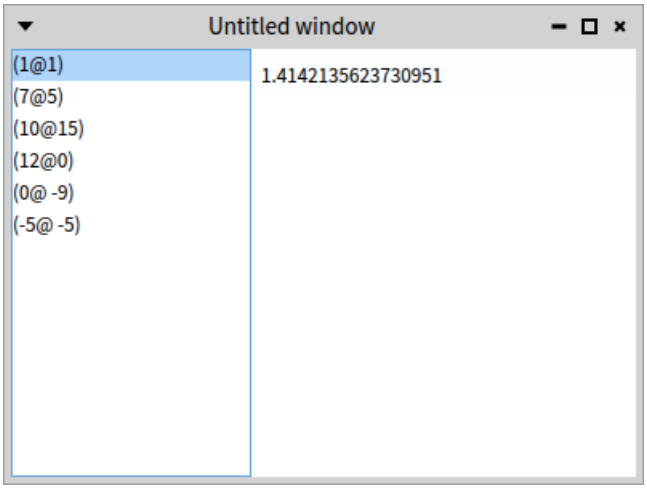


Рис. 14.2. Демонстратор «огляд-деталізація», який виконує перетворення

Після відкриття бачимо вікно як на рис. 14.2.

14.5. Виконання перенесень без вхідного порту

Іноді немає необхідності надсилати переданий об'єкт на вхідний порт демонстратора. Якщо відправник повинен зробити щось додаткове з об'єктом, який передається через вихідний порт, то він може використати повідомлення *transmitDo:*. Це повідомлення приймає блок, який буде виконано в момент перенесення.

Розширимо попередній приклад, щоб проілюструвати сказане. Припустімо, що з метою налагодження ми хочемо вивести вибрану точку до Transcript. Реалізувати *connectPresenters* традиційним способом можна так.

```
OverviewDetailPresenter >> connectPresenters
   overview whenSelectedItemChangedDo: [ :selectedPoint |
      | distanceToOrigin |
      distanceToOrigin := selectedPoint
         ifNil: [  ]
         ifNotNil: [ (selectedPoint distanceTo: 0@0) asString ].
      detail text: distanceToOrigin.
      selectedPoint crTrace ]
```

Досягти такої самої поведінки за допомогою перенесень можна так.

```
OverviewDetailPresenter >> connectPresenters
   overview
      transmitTo: detail
      transform: [ :selectedPoint |
         selectedPoint
            ifNil: [  ]
            ifNotNil: [ (selectedPoint distanceTo: 0@0) asString ] ].
   overview transmitDo: [ :selectedPoint | selectedPoint crTrace ]
```

14.6. Дії після перенесення

Іноді після перенесення демонстраторові потрібно щось змінити, враховуючи новий стан вкладеного демонстратора. Прикладами можуть бути попередній вибір чогось,

оновлення стану кнопок панелі інструментів. Ось тут і з'являються дії після перенесення. Повідомлення, які ми бачили досі, мають варіанти з додатковим ключем *postTransmission:* і аргументом до нього.

Розгляньмо востаннє простий приклад. Припустімо, що треба виділити текст у демонстраторі відразу після того, як він там з'явився. За традиційного підходу надішлемо демонстраторові тексту *selectAll* у методі *connectPresenters*.

```
OverviewDetailPresenter >> connectPresenters
   overview whenSelectedItemChangedDo: [ :selectedPoint |
      | distanceToOrigin |
      distanceToOrigin := selectedPoint
         ifNil: [  ]
         ifNotNil: [ (selectedPoint distanceTo: 0@0) asString ].
      detail text: distanceToOrigin.
      detail selectAll.
      selectedPoint crTrace ]
```

У випадку використання перенесень додамо додаткове ключове слово *postTransmission:* до повідомлення, яке використовували раніше. Додатковий аргумент – це блок, який приймає від одного до трьох аргументів. Перший аргумент часто називають *destination*. Це той демонстратор, чий вхідний порт. Другий аргумент, який часто називають *origin*, є демонстратором вихідного порту. Третій аргумент – переданий об'єкт без застосованих до нього перетворень. У нашому прикладі потрібен доступ тільки до *destination*. Тому в блоці *postTransmission:* є тільки один аргумент.

```
OverviewDetailPresenter >> connectPresenters
   overview
      transmitTo: detail
      transform: [ :selectedPoint |
         selectedPoint
            ifNil: [  ]
            ifNotNil: [ (selectedPoint distanceTo: 0@0) asString ] ]
      postTransmission: [ :destination | destination selectAll ].
   overview transmitDo: [ :selectedPoint | selectedPoint crTrace ]
```

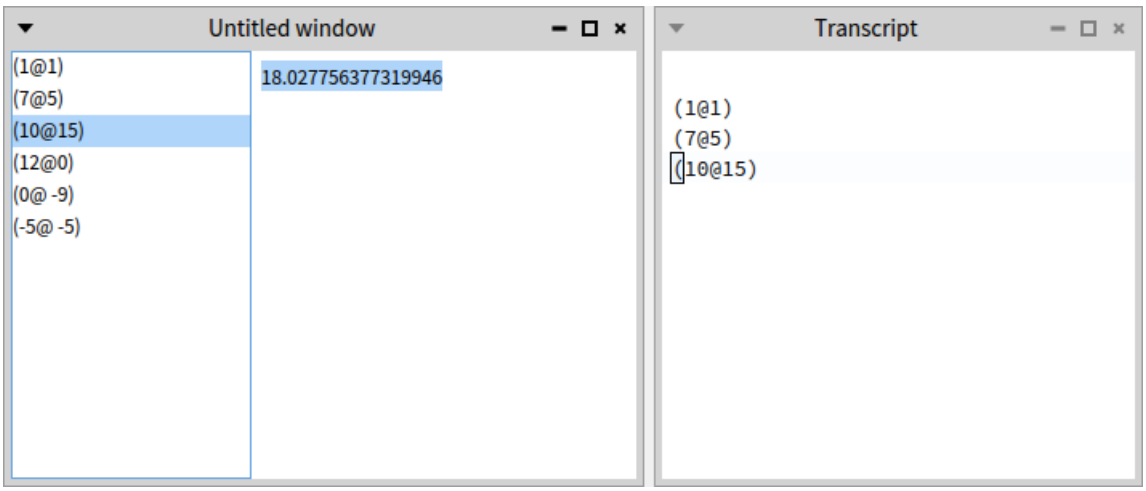


Рис. 14.3. Остаточний демонстратор «огляд-деталізація»

Відкриймо демонстратора знову, щоб випробувати дію перенесень.

```
OverviewDetailPresenter new open
```

Виберемо кілька точок зі списку. Тоді побачимо вікна як на рис. 14.3. Відстань від останньої вибраної точки до початку системи координат виділяється в тексті, а кожна вибрана точка з'являється у вікні консолі.

14.7. Наявні класи портів

Ми описали дуже простий і водночас часто вживаний варіант використання з двома типами портів, коли вибраний користувачем об'єкт є вхідним для деякого демонстратора. Вихідний порт *SpListPresenter* – це екземпляр класу *SpSelectionPort*, а вхідний порт *SpTextPresenter* – екземпляр *SpTextPort*. Екземпляри *SpSelectionPort* використовують для створення вихідних портів демонстраторів, які реалізують механізм вибору. Вхідними портами демонстраторів, які відображають або редагують текст, слугують екземпляри *SpTextPort* (див. ієрархію класів *SpAbstractTextPresenter*). Усі класи демонстраторів, які реалізують візуальні компоненти, мають свої окремі класи вихідного порту та вхідного порту. Тому існують різні класи портів. Їхнім спільним надкласом є *SpAbstractPort* з двома безпосередніми підкласами *SpInputPort* і *SpOutputPort*.

14.8. Порти та вкладені демонстратори

Для реалізації власного демонстратора використовують вкладені демонстратори, які можна під'єднувати до перенесень. Але що трапиться, якщо використати свій демонстратор в інших демонстраторах? Відповідь проста: для його приєднання так само можна використати перенесення.

Щоб власний демонстратор був придатним для використання з перенесеннями, його клас має визначити вихідні та вхідні порти, а також реалізувати методи *defaultOutputPort* і *defaultInputPort*. Залежно від поведінки демонстратора, його клас реалізує один або обидва методи. Для визначення портів можна використати один із наявних класів портів. Якщо ж не знайдеться відповідного класу порту для проєктованого демонстратора, то можна визначити свій власний.

У деяких випадках немає потреби визначати нові порти, натомість можна застосувати делегування, щоб повторно використати порт вкладеного демонстратора. У наступному параграфі описано приклад делегування.

14.9. Складніший приклад

У цьому параграфі ми повернемося до клієнтського застосунку електронної пошти з розділу 12.

Почнемо з класу демонстратора *MailClientPresenter*. Він мав такий метод:

```
MailClientPresenter >> connectPresenters
   account whenSelectionChangedDo: [ :selection |
      | selectedFolderOrEmail |
      selectedFolderOrEmail := selection selectedItem.
      reader read: selectedFolderOrEmail.
      self updateAfterSelectionChangedTo: selectedFolderOrEmail ]
```

Пристосуємо його для використання перенесень.

```
MailClientPresenter >> connectPresenters
   account
      transmitTo: reader
      postTransmission: [ :destination :origin :selectedFolderOrEmail |
         self updateAfterSelectionChangedTo: selectedFolderOrEmail ]
```

Тут демонстраторові *account* (екземпляру *MailAccountPresenter*) надсилають повідомлення *transmitTo:postTransmission:*, щоб виразити те, що зроблений у ньому вибір треба перенести до демонстратора *reader* (екземпляра *MailReaderPresenter*), і що після перенесення потрібно виконати додаткову дію.

Метод *connectPresenters* дуже лаконічно виражає бажану поведінку перенесення, але він поки що не працює без додаткових змін у коді. Адже ми ще не визначили вихідний порт демонстратора *account* і вхідний порт демонстратора *reader*.

Для демонстратора *account* питання вирішується просто: він може делегувати *defaultOutputPort* вкладеному демонстраторові дерева, тому що стандартним вихідним портом *SpTreePresenter*, який містить папки та електронні листи, є *SpSelectionPort*. Він надасть вибрану папку або листа.

```
MailAccountPresenter >> defaultOutputPort
   ^ foldersAndEmails defaultOutputPort
```

Визначення вхідного порту демонстратора *reader* дещо складніше. Екземпляр *MailReaderPresenter* очікує отримання екземпляра *Folder* або *Email*. Справді, в оригінальній реалізації методу *MailClientPresenter >> connectPresenters* було сказано *reader read: selectedFolderOrEmail*. Папка чи електронний лист є моделлю даних демонстратора *MailReaderPresenter*, тому їх можна було б передавати в повідомленні *setModel:*. Але в методі використали повідомлення *read:*, бо концептуально це мало сенс у контексті *MailReaderPresenter*.

Немає класу вхідного порту, який би знав протокол *read:*, тому можливі два варіанти. Або реалізувати новий клас вхідного порту, або використати *SpModelPort*. Оберемо друге, бо надаємо перевагу використанню наявних класів. Але це означає, що доведеться реалізувати *setModel:* у класі *MailReaderPresenter*, бо для того, щоб надати перенесений об'єкт демонстратору-отримувачу, *SpModelPort* надсилає повідомлення *setModel:*. Можна делегувати методу *read:*, щоб зробити демонстратор сумісним із протоколом, на який очікує *SpModelPort*.

```
MailReaderPresenter >> defaultInputPort
   ^ SpModelPort newPresenter: self

MailReaderPresenter >> setModel: email
   self read: email
```

На цьому зміни щодо впровадження перенесень на рівні *MailClientPresenter* завершено. Після відкриття поштового клієнта за допомогою *(MailClientPresenter on: MailAccount new) open* поштовий застосунок поводиться як і раніше, але тепер він використовує перенесення.

14.10. Ще одна видозміна

Є ще один демонстратор, де можна використати перенесення – *EmailPresenter*. Його оригінальний метод *connectPresenters* було реалізовано так.

```
EmailPresenter >> connectPresenters
   from whenTextChangedDo: [ :text | self model from: text ].
   to whenTextChangedDo: [ :text | self model to: text ].
   subject whenTextChangedDo: [ :text | self model subject: text ].
   body whenTextChangedDo: [ :text | self model body: text ]
```

Хоча зміни від використання перенесень є поверхневими, але цікаво, що такої ж поведінки можна досягти за допомогою *transmitDo:*.

```
EmailPresenter >> connectPresenters
   from transmitDo: [ :text | self model from: text ].
   to transmitDo: [ :text | self model to: text ].
   subject transmitDo: [ :text | self model subject: text ].
   body transmitDo: [ :text | self model body: text ]
```

Використано повідомлення *transmitDo:* тому, що тут потрібний побічний ефект на моделі. Ніякі вхідні порти в перенесенні не задіяні.

14.11. Підсумки розділу

У цьому розділі описано перенесення і порти, їхнє використання проілюстровано двома прикладами.

Вихідні порти визначають джерело даних і перетворення, які треба застосувати до них перед перенесенням у вхідний порт. Вхідні порти визначають приймач даних. Перенесення з'єднують вихідні порти з вхідними, щоб визначити потік даних між демонстраторами.

Розділ 15
Стилізація застосунку

У цьому розділі описано, як оголошувати та використовувати стилі в застосунках Spec.

Спочатку представимо стилі та таблиці стилів , а потім застосуємо їх до поштової програми, описаної в розділі 12. Продемонструємо, як Spec керує стилями, та як можна адаптувати вигляд демонстратора.

Є два способи опису таблиць стилів: один для Morphic використовує розширену версію STON, інший – CSS для GTK. У цьому розділі зосередимося на таблицях стилів Morphic у Pharo 12. Перш ніж показати, як ефективно використовувати стилі для покращення зовнішнього вигляду програми, закладемо певну основу.

15.1. Кількома словами

Таблиця стилів застосунку

У Spec застосунок має таблицю стилів, яку можна задати повідомленням *stylesheet:*. Потім кожна програма може удосконалювати свою таблицю стилів.

```
app styleSheet: styleSheet.
```

Оголошення стилів

Таблиця стилів для графічної бібліотеки Morphic (на відміну від GTK) визначається як спеціальна версія рядка STON. Такий рядок розпізнають і перетворюють на елементи стилю. Показаний нижче фрагмент коду створює таблицю стилів, у якій усі шрифти задано товстими і визначено три стилі малювання *red*, *bgGray* і *blue*.

```
SpStyleVariableSTONReader fromString:
   '.application [
      Font { #bold: true },
      .red [ Draw { #color: #red } ],
      .bgGray [ Draw { #backgroundColor: #E2E2E2 } ],
      .blue [ Draw { #color: #blue } ]
   ]'
```

Застосування стилів

Кожен демонстратор може застосувати стиль з таблиці за допомогою повідомлення *addStyle:* і скасувати його за допомогою *removeStyle:*. У прикладі змінено колір тексту демонстратора напису за допомогою застосування стилю *red*.

```
label := presenter newLabel.
label label: 'I am a label'.
label addStyle: 'red'
```

15.2. Як працюють стилі?

Стилі в Spec працюють подібно до CSS. Це таблиці стилів, у яких визначено властивості для відображення демонстратора. Приклади властивостей: кольори, ширина, висота, шрифт. Загальне правило полягає в тому, що ліпше використовувати стилі замість фіксованих обмежень, тому що так програма буде сприйнятливішою до налаштувань.

Однак зауважте, що таблиця стилів охоплює не всі аспекти візуального компонента, і можуть знадобитися властивості, не описані в поточній версії Spec. У майбутньому при переході на компоненти Toplo Spec перегляне підтримку стилів і збільшить покриття.

15.3. Таблиці стилів

Spec комплектує стиль для демонстратора, а потім збирає стилі для його вкладених демонстраторів.

Кореневий рівень

Таблиця стилів завжди має кореневий елемент, він повинен називатися *.application*. У зображеній таблиці стилів зазначено, що для цілого застосунку (тобто для всіх його демонстраторів, якщо не перевизначено в іншому стилі) встановлено шрифт розміру 10 пікселів гарнітури «Source Sans Pro».

```
.application [
   Font { #name: "Source Sans Pro", #size: 10 },
   ...
```

Вкладені демонстратори

Рівні стилю задають каскадом, який розпочинається з *.application*. Ось три стилі:

```
.application.label.header
.application.link
.application.checkBox
```

15.4. Оголошення стилю

Стилі для Morphic оголошують за допомогою STON. STON – це текстова нотація об’єкта. Вона описана в окремому розділі книги «Enterprise Pharo», доступній на сайті https://books.pharo.org.

Стилі Spec підтримують п’ять властивостей: *Geometry*, *Draw*, *Font*, *Container* і *Text*, як показано в прикладі.

```
Geometry { #hResizing: true }
Draw { #color:  Color { #red: 1, #green: 0, #blue: 0, #alpha: 1}}
Draw { #color: #blue}
Font { #name: "Lucida Grande", #size: 10, #bold: true }
Container { #borderColor: Color { #rgb: 0, #alpha: 0 },
      #borderWidth: 2,
      #padding: 5 }
```

Можна визначити свій стиль на рівні програми та застосувати його до конкретного демонстратора за допомогою повідомлення *addStyle:*. Наприклад, *aPresenter addStyle: 'section'* вибирає стиль *.section* і призначає його демонстраторові *aPresenter*.

15.5. Приклади таблиць стилів

Ось два приклади таблиць стилів.

```
styleSheet
 ^ SpStyleVariableSTONReader fromString: '
   .application [
      Font { #name: "Source Sans Pro", #size: 10 },
      Geometry { #height: 25 },
      .label [
         Geometry { #hResizing: true }  ],
      .headerError [Draw {
         #color: Color{ #red: 1, #green: 0, #blue: 0, #alpha: 1}}  ],
      .headerSuccess [Draw {
         #color: Color{ #red: 0, #green: 1, #blue: 0, #alpha: 1}}  ],
      .header [
         Draw { #color: Color{ #rgb: 622413393 }},
         Font { #name: "Lucida Grande", #size: 10, #bold: true } ],
      .shortcut [
         Draw { #color: Color{ #rgb: 622413393 } },
         Font { #name: "Lucida Grande", #size: 10 } ],
      .fixed [
         Geometry { #hResizing: false, #width: 100 } ],
      .dim [
         Draw { #color : Color{ #rgb: 708480675 } } ]
   ]'
```

Приклад нижче доповнює усталену таблицю стилів, яку повертає вираз *SpStyle defaultStyleSheet*.

```
styleSheet
   ^ SpStyle defaultStyleSheet, (SpStyleVariableSTONReader fromString: '
   .application [
      Draw { #backgroundColor: #lightRed},
      .section [
         Draw { #color: #green, #backgroundColor: #lightYellow},
         Font { #name: "Verdana", #size: 12, #italic: true,
                #bold: true}],
      .disabled [ Draw { #backgroundColor: #lightGreen} ],
      .textInputField [ Draw { #backgroundColor: #blue} ],
      .label [
         Font { #name: "Verdana", #size: 10, #italic: false,
                #bold: true},
         Draw { #color: #red, #backgroundColor: #lightBlue} ]
   ]')
```

15.6. Анатомія стилю

Кожен вид елемента стилю використовує певні властивості, визначені пов'язаним з ним класом, підкласом *SpPropertyStyle*. *SpPropertyStyle* має 5 підкласів: *SpContainerStyle*, *SpDrawStyle*, *SpFontStyle*, *SpTextStyle* і *SpGeometryStyle*.

Ці підкласи визначають п'ять наявних типів властивостей.

- **Container**: *SpContainerStyle* – керує вирівнюванням демонстраторів. Зазвичай цей стиль задає головний демонстратор, який містить і впорядковує піддемонстратори.
- **Draw**: *SpDrawStyle* – змінює властивості, пов'язані з прорисовуванням демонстратора, наприклад, колір олівця і колір тла.
- **Font**: *SpFontStyle* – визначає властивості, пов'язані зі шрифтами.
- **Text**: *SpTextStyle* – керує властивостями *SpTextInputFieldPresenter*.
- **Geometry**: *SpGeometryStyle* – визначає розміри: ширина, висота, мінімальна висота тощо.

Якщо ви захочете дослідити клас стилю в Системному оглядачі, то перевірте спочатку, чи обрали відповідний клас. Для цього надішліть класу повідомлення *stonName*. Він поверне символ, який використовується в нотації STON. Наприклад, *SpDrawStyle stonName* повертає *#Draw*.

Приклад

Щоб змінити колір демонстратора, потрібно створити стиль і використати властивість *color* класу *SpDrawStyle*. Щоб задати бажаний колір, можна використати або його шістнадцятковий код, або селектор методу класу *Color*, якщо знайдеться потрібний.

Тут визначено два стилі: *lightGreen* і *lightBlue*, які можна застосувати до будь-якого демонстратора.

```
.application [
   .lightGreen [ Draw { #color: #B3E6B5 } ],
   .lightBlue [ Draw { #color: #lightBlue } ] ]
```

15.7. Змінні середовища

Можна також використовувати змінні середовища, щоб отримати значення попередньо визначених кольорів і шрифтів поточної теми інтерфейсу. Наприклад, можна створити два стилі для зміни шрифту тексту та один для кольору фону демонстратора.

```
.application [
   .codeFont [ Font { #name: EnvironmentFont(#code) } ],
   .textFont [ Font { #name: EnvironmentFont(#default) } ],
   .bg [ Draw { #color: EnvironmentColor(#background) } ]
]
```

Дослідіть підкласи *SpStyleEnvironmentVariable*.

15.8. Зміни на рівні кореня

Можна одним махом змінити стилі для всіх демонстраторів застосунку. Наприклад, відображати весь текст товстим можна за допомогою такого стилю.

```
.application [
   Font { #bold: true }
]
```

15.9. Визначення стилів застосунку

Припустимо, що виникла потреба стилізувати поштову програму, описану в розділі 12 і доповнену в розділі 13. Потрібно, щоб мітки в частині інтерфейсу користувача для редагування пошти використовували більший шрифт і синій колір, поля редагування мали світло-жовте тло, а межа поля редагування змісту листа була чорна. Перелічені вимоги приводять до такої таблиці стилів.

```
.application [
   .fieldLabel [ Font { #size: 12 }, Draw { #color: #blue } ],
   .field [ Draw { #backgroundColor: #lightYellow } ],
   .bodyField [ Container { #borderWidth: 1, #borderColor: #black} ]
]
```

Стиль *.fieldLabel* визначає 12-піксельний синій шрифт. Стиль *.field* визначає світло-жовтий колір тла, а стиль *.bodyField* – чорну межу товщини 1 піксель.

Щоб використовувати стилі, потрібно пов'язати головний демонстратор з екземпляром застосунку. Один із способів зробити це такий.

```
| mailClient application styleSheet |
mailClient := MailClientPresenter on: MailAccount new.
application := SpApplication new.
mailClient application: application.

styleSheet := SpStyle defaultStyleSheet,
   (SpStyleVariableSTONReader fromString:
   '.application [
      .fieldLabel [ Font { #size: 12 }, Draw { #color: #blue } ],
      .field [ Draw { #backgroundColor: #lightYellow } ],
      .bodyField [ Container { #borderWidth: 1, #borderColor: #black } ]
   ]').

application styleSheet: SpStyle defaultStyleSheet , styleSheet.
```

Але такий спосіб передбачає створення таблиці стилів поза контекстом поштового застосунку. Замість цього оголосимо новий клас застосунку і перевантажимо метод *styleSheet*.

```
SpApplication << #MailClientApplication
   slots: {};
   package: 'CodeOfSpec20Book'

MailClientApplication >> styleSheet
   | customStyleSheet |
```

```
        customStyleSheet := SpStyleVariableSTONReader fromString:
          '.application [
            .fieldLabel [ Font { #size: 12 }, Draw { #color: #blue } ],
            .field [ Draw { #backgroundColor: #lightYellow } ],
            .bodyField [ Container { #borderWidth: 1, #borderColor: #black } ]
          ]'.
        ^ super styleSheet , customStyleSheet
```

Зауважимо, що метод надсилає *styleSheet* надкласові. *SpApplication >> styleSheet* повертає усталену таблицю стилів таку саму, як *SpStyle defaultStyleSheet*, яку ми бачили раніше. Повідомлення «кома» поєднує усталену таблицю стилів і нашу власну. Так ми гарантуємо, що всі стандартні стилі все ще застосовуються до всіх демонстраторів, а наші стилі застосовуються поверх стандартних.

Щоб легко відкривати поштову програму, визначимо метод *start*.

```
     MailClientApplication >> start
        (MailClientPresenter on: MailAccount new)
           application: self;
           open
```

Тепер можна було б відкрити стилізовану поштову програму.

```
     MailClientApplication new start
```

Звичайно, це не дало б особливого ефекту, адже ми ще не застосовували стилі.

15.10. Застосування стилів

Визначені в попередньому параграфі стилі призначені для класу *EmailPresenter*, який визначає схожий на форму інтерфейс користувача для редагування електронного листа. Початкова реалізація методу *defaultLayout* була така.

```
     EmailPresenter >> defaultLayout
        | toLine subjectLine fromLine |
        fromLine := SpBoxLayout newTopToBottom
           add: 'From:' expand: false;
           add: from expand: false;
           yourself.
        toLine := SpBoxLayout newTopToBottom
           add: 'To:' expand: false;
           add: to expand: false;
           yourself.
        subjectLine := SpBoxLayout newTopToBottom
           add: 'Subject:' expand: false;
           add: subject expand: false;
           yourself.
        ^ SpBoxLayout newTopToBottom
           spacing: 10;
           add: fromLine expand: false;
           add: toLine expand: false;
           add: subjectLine expand: false;
           add: body;
           yourself
```

Щоб стилізувати поля, потрібно внести в код деякі зміни. Показана вище реалізація базується на методі *add:expand:*, який для зручності дозволяє першому аргументу бути рядком, наприклад, *add: 'From:' expand: false*. Стилізувати рядок не вдасться. Стилізувати можна тільки демонстратор, тому доведеться власноруч створювати демонстратори написів. Тоді можна додати необхідний стиль до трьох демонстраторів написів за допомогою повідомлення *addStyle: 'fieldLabel'*. Зауважте, що визначення стилю використовує *.fieldLabel*, а в аргументі повідомлення *addStyle:* передають рядок без крапки на початку.

```
EmailPresenter >> defaultLayout
   | toLine subjectLine fromLine fromLabel toLabel subjectLabel |
   fromLabel := self newLabel
      label: 'From:';
      addStyle: 'fieldLabel';
      yourself.
   fromLine := SpBoxLayout newTopToBottom
      add: fromLabel expand: false;
      add: from expand: false;
      yourself.
   toLabel := self newLabel
      label: 'To:';
      addStyle: 'fieldLabel';
      yourself.
   toLine := SpBoxLayout newTopToBottom
      add: toLabel expand: false;
      add: to expand: false;
      yourself.
   subjectLabel := self newLabel
      label: 'Subject:';
      addStyle: 'fieldLabel';
      yourself.
   subjectLine := SpBoxLayout newTopToBottom
      add: subjectLabel expand: false;
      add: subject expand: false;
      yourself.
   ^ SpBoxLayout newTopToBottom
         spacing: 10;
         add: fromLine expand: false;
         add: toLine expand: false;
         add: subjectLine expand: false;
         add: body;
         yourself
```

Тепер, коли написи налаштовано, наступним кроком буде стилізація полів. Адаптуємо метод *initializePresenters*, де створюють їхні демонстратори. Спочатку метод містив перші чотири твердження. Додамо ще чотири, щоб задати стилі.

```
EmailPresenter >> initializePresenters
   from := self newTextInput.
   to := self newTextInput.
   subject := self newTextInput.
   body := self newText.
   from addStyle: 'field'.
   to addStyle: 'field'.
   subject addStyle: 'field'.
   body addStyle: 'field'; addStyle: 'bodyField'
```

Демонстраторам *from*, *to*, *subject* додаємо один стиль, *'field'*, а демонстраторові *body* – два: і *'field'*, і *'bodyField'*. Перший стиль змінить колір фону поля редагування, а другий – замалює межу поля чорним.

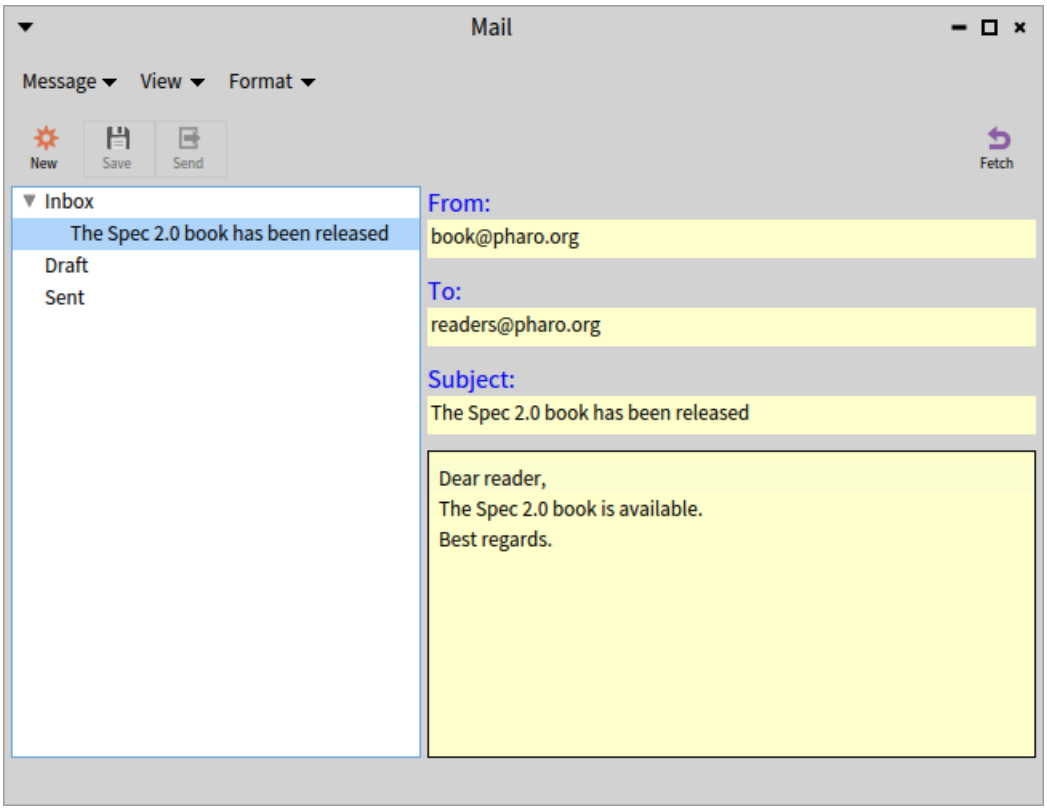


Рис. 15.1. Стилізовані написи та поля редагування

За допомогою *MailClientApplication new start* можна відкрити застосунок «*Mail*» і побачити стилі в дії (див. рис. 15.1).

15.11. Динамічне застосування стилів

Припустимо, що для чернетки листа поля редагування мають мати тло іншого кольору. Ось тут і з'являється динамічний стиль. Можна додавати та видаляти стилі під час виконання застосунку, коли змінюється його стан. Зробімо це зі стилями полів.

Спочатку доповнимо метод *styleSheet* класу застосунку оголошенням нових стилів. Додамо стиль *.draftMail* із вкладеним стилем *.field*, який визначає рожевий колір тла. Вкладення означає те, що стиль *.field* застосовується в контексті стилю *.draftMail*.

```
MailClientApplication >> styleSheet
   | customStyleSheet |
   customStyleSheet := SpStyleVariableSTONReader fromString:
    '.application [
       .fieldLabel [ Font { #size: 12 }, Draw { #color: #blue } ],
       .field [ Draw { #backgroundColor: #lightYellow } ],
       .draftMail [
           .field [ Draw { #backgroundColor: #pink } ]  ],
       .bodyField [ Container { #borderWidth: 1, #borderColor: #black } ]
     ]'.
   ^ super styleSheet , customStyleSheet
```

Наступним кроком буде застосування нового стилю. Екземпляр *EmailPresenter* має модель. Коли вона змінюється, демонстратор отримує сповіщення повідомленням *modelChanged*. Початкову реалізацію методу (перші чотири рядки в коді методу) можна легко розширити для застосування різних стилів залежно від виду моделі, яка містить екземпляр класу *Email*.

```
EmailPresenter >> modelChanged
   from text: (self model from ifNil: [ '' ]).
   to text: (self model to ifNil: [ '' ]).
   subject text: (self model subject ifNil: [ '' ]).
   body text: (self model body ifNil: [ '' ]).
   self model isDraft
      ifTrue: [
         from addStyle: 'draftMail.field'.
         to addStyle: 'draftMail.field'.
         subject addStyle: 'draftMail.field'.
         body addStyle: 'draftMail.field' ]
      ifFalse: [
         from removeStyle: 'draftMail.field'.
         to removeStyle: 'draftMail.field'.
         subject removeStyle: 'draftMail.field'.
         body removeStyle: 'draftMail.field' ]
```

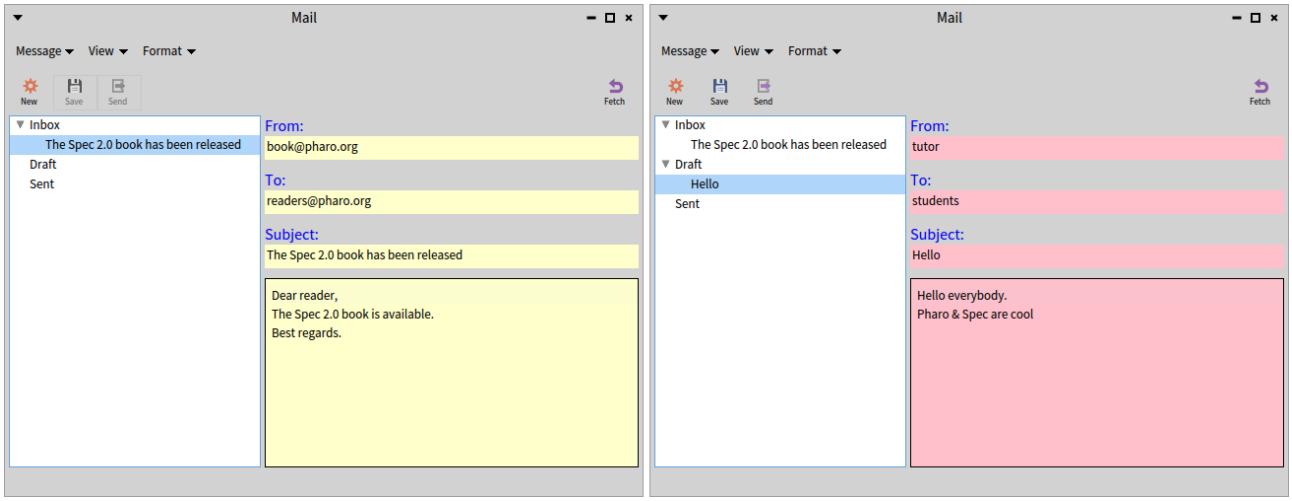


Рис. 15.2. Різні стилі для листів різних видів

Відкриємо знову застосунок «*Mail*» і виберемо різні види листів. На рис. 15.2 зображено два стилі.

15.12. Підсумки розділу

Використання стилів у Spec – це чудовий засіб. Він полегшує створення узгодженого дизайну, оскільки можемо задати однаковий стиль кільком демонстраторам. Якщо треба змінити якийсь стиль, то достатньо відредагувати тільки таблицю стилів. За допомогою заміни стилю можна динамічно змінювати зовнішній вигляд демонстратора[16].

[16] Для цього навіть не потрібно закривати вікно запущеного застосунку (прим. – *Ярошко С.*).

Розділ 16

Використання Athens і Roassal у Spec

Початково частину цього розділу написав Рено де Вільмер (Renaud de Villemeur). Він показав, як можна інтегрувати векторну графіку в компоненти Spec. Дякуємо йому за його внесок. У цьому розділі з'ясовано, як використати Athens (полотно, побудоване на графічних засобах Cairo) для малювання за допомогою низькорівневого API на полотні всередині демонстратора Spec. Потім описано, як використати у демонстраторі Spec механізм візуалізації Roassal. Наприкінці продемонстровано, як інтегрувати в компонент Spec морфу, екземпляр Morph, намальовану на полотні Athens.

16.1. Вступ

Існує два види комп'ютерної графіки: векторна та растрова. Растрова графіка представляє зображення у вигляді набору пікселів. Векторна графіка використовує для цього геометричні примітиви: точки, лінії, криві, многокутники. Такі примітиви описують за допомогою математичних рівнянь.

Обидва види комп'ютерної графіки мають переваги і недоліки. Перевагами векторної графіки перед растровою такі:

- менший розмір збереженого зображення;
- можливість масштабування без обмежень;
- перетворення зображення, як от переміщення, масштабування, заповнення чи обертання, не погіршують його якість.

Зрештою, на комп'ютері зображення відображаються на екрані певного розміру з певною роздільною здатністю. На противагу растровій графіці, яка не дуже добре масштабується, якщо роздільна здатність пристрою надто відрізняється від роздільної здатності зображення, то векторна графіка растеризується відповідно до дисплея, на якому вона відображатиметься. У ході растеризації зображення, описане у форматі векторної графіки, перетворюється на набір пікселів для виведення на екран.

Бібліотека Morphic використовує растровий підхід. Вона перетворює вміст полотна в структуру на основі пікселів (клас Form). Більшість графіки у Pharo растрова: Form – абстракція низького рівня, яку використовує Morphic. Проте Pharo пропонує альтернативну векторну графіку. Для цього він використовує та надає користувачеві Cairo. Доступні два програмні інтерфейси.

- Давніший, Athens, краще захищає розробників від можливих помилок.
- Alexandrie – новий API нижчого рівня. Його більш агресивно оптимізовано. Він становить основу для бібліотеки Bloc, яка прийде на зміну Morphic.

Коли Athens інтегровано зі Spec, програміст використовує його рушій візуалізації для створення зображення. Тоді воно перетворюється на Form і відображається на екрані.

16.2. Безпосередня інтеграція Athens зі Spec

Спочатку створимо демонстратор, який промальовуватиме зображення за допомогою Athens. Назвемо його *AthensExamplePresenter*.

```
SpPresenter << #AthensExamplePresenter
   slots: { #athensPresenter };
   package: 'CodeOfSpec20Book'
```

Розмістимо *athensPresenter* у звичайному послідовному макеті.

```
AthensExamplePresenter >> defaultLayout
   ^ SpBoxLayout newTopToBottom
         add: athensPresenter;
         yourself
```

Цей демонстратор створює та налаштовує екземпляр *SpAthensPresenter* як показано нижче.

```
AthensExamplePresenter >> initializePresenters
   athensPresenter := self instantiate: SpAthensPresenter.
   athensPresenter surfaceExtent: 600@400.
   athensPresenter drawBlock: [ :canvas | self render: canvas ]
```

Він налаштовує *AthensPresenter* так, щоб він будував зображення у відповідь на повідомлення *render:*. Пропонований метод *render:* містить типову послідовність інструкцій для налаштування полотна.

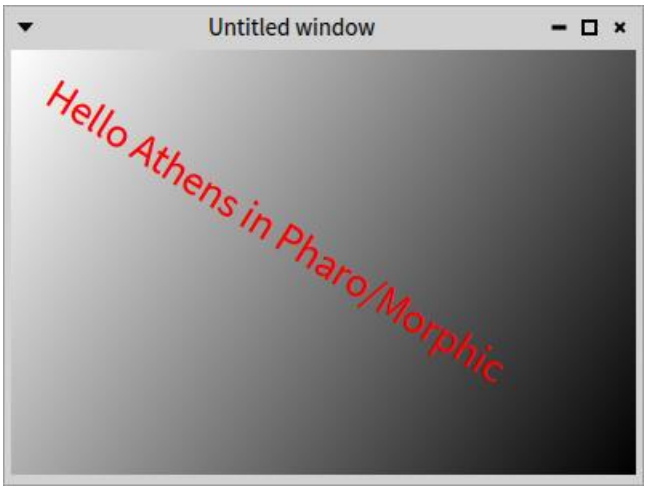


Рис. 16.1. Демонстратор використовує *SpAthensPresenter*

```
AthensExamplePresenter >> render: canvas
   | surface font |
   surface := canvas surface.
   font := LogicalFont familyName: 'Source Sans Pro' pointSize: 10.
   surface clear.
   canvas
      setPaint: ((LinearGradientPaint from: 0@0  to: surface extent)
      colorRamp: {  0 -> Color white. 1 -> Color black }).
   canvas drawShape: (0@0 extent: surface extent).
   canvas setFont: font.
```

```
        canvas setPaint: Color red.
        canvas
           pathTransform translateX: 20 Y: 20 + (font getPreciseAscent);
           scaleBy: 2;
           rotateByDegrees: 32.
        canvas drawString: 'Hello Athens in Pharo/Morphic'
```

Виконання коду *AthensExamplePresenter new open* створює вікно, як на рис. 16.1.

Цей простий приклад не охоплює випадку перемальовування, яке, можливо, доведеться виконати, якщо щось зміниться, але він демонструє ключовий аспект архітектури. Такі самі дії можна виконати і з новим полотном Alexandrie на базі Cairo.

Зверніть увагу на те, що тут малюємо безпосередньо на полотні без маніпулювання об'єктами Morphic. Такі перетворення зробимо в наступному параграфі.

16.3. Інтеграція Roassal зі Spec

У цьому параграфі опишемо, як можна визначити демонстратор Spec, який дає змогу малювати візуалізації Roassal.

Уявімо, що треба за допомогою Roassal намалювати деякі фігури. Тут намалюємо два прямокутники, але зображати можна багато інших графічних елементів.

```
| c blueBox redBox |
c := RSCanvas new.
blueBox := RSBox new
   size: 80;
   color: #blue.
redBox := RSBox new
   size: 80;
   color: #red.
c
   add: blueBox;
   add: redBox.
blueBox translateBy: 40 @ 20.
c inspect
```

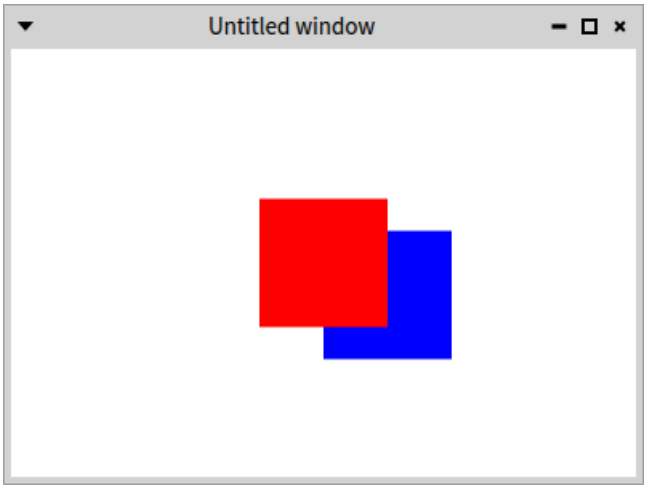


Рис. 16.2. Застосунок використовує візуалізацію Roassal

Використання *SpRoassalInspectorPresenter*

Створити демонстратор Spec, що підтримує Roassal, так само просто, як створити екземпляр *SpRoassalInspectorPresenter* і передати йому полотно, на якому малюємо візуалізацію Roassal. Саме це і робить вираз *SpRoassalInspectorPresenter new canvas: c; open*. Замініть у попередньому фрагменті коду *c inspect* на

```
SpRoassalInspectorPresenter new canvas: c; open
```

виконайте його, і відкриється вікно Spec з демонстратором Roassal усередині, як на рис. 16.2.

16.4. Клас *SpRoassalPresenter*

Spec надає демонстратор, призначений для візуалізації Roassal. Такий демонстратор називається *SpRoassalPresenter*, і створити його можна за допомогою повідомлення *newRoassal* у методі *initializePresenter* як, наприклад, це робить *SpColorPicker*.

```
SpColorPicker >> defaultLayout
   | sp sp2 |
   sp := self newRoassal.
   sp2 := self newPresenter.
   sp2 layout: SpBoxLayout newTopToBottom.
   sp canvas color: Color black translucent.

   ^ SpBoxLayout newTopToBottom
      add: colorMap height: 150;
      add: colorSlider height: 25;
      add: alphaSlider height: 25;
      add: colorCodePresenter expand: false;
      add: sp2 height: 10;
      add: sp height: 1;
      add: paletteChooser;
      spacing: 1;
      yourself.
```

Головним API *SpRoassalPresenter* є метод *canvas* і метод *script:*, використаний у тесті нижче. З полотном Roassal можна взаємодіяти як звичайно, а результат відображається в демонстраторі Roassal.

```
testBasic
   | spec value window |
   self isValid ifFalse: [ ^ self ].
   spec := SpRoassalPresenter new.
   window := spec asWindow open.
   value := 0.

   spec script: [ :view | view addShape: RSBox new. value := value + 1 ].
   self assert: value equals: 1.
   spec script: [ :view | view addShape: RSBox new. value := 0 ].
   self assert: value equals: 0.
   window close
```

16.5. Вітання у Athens через об’єкти Morphic

Команда розробників Pharo активно працює над заміною Morphic на нову графічну бібліотеку Bloc. Ми все ж вважаємо, що описаний тут підхід вартий уваги. Продемонструємо, як можна визначити морфу, яка малює на полотні Athens, і як таку морфу відобразити всередині компонента Spec.

Ми показуємо, як використовувати Athens, безпосередньо інтегровані з Morphic. Саме для цього створимо підклас Morph. Вираз *AthensHello new openInWindow* відображатиме той самий вміст, що й на рис. 16.1.

Почнемо з визначення підкласу, який наслідує Morph.

```
Morph << #AthensHello
   slots: { #surface };
   package: 'CodeOfSpec20Book'
```

На етапі ініціалізації створимо поверхню Athens.

```
AthensHello >> initialize
   super initialize.
   self extent: self defaultExtent.
   surface := AthensCairoSurface extent: self extent
```

тут метод *defaultExtent* визначено так

```
AthensHello >> defaultExtent
   ^ 400@300
```

Обов’язковий у підкласах Morph метод *drawOn:* просить Athens промалювати свій рисунок, а потім відображає його на полотні Morphic як растрове зображення.

```
AthensHello >> drawOn: aCanvas
   self renderAthens.
   surface displayOnMorphicCanvas: aCanvas at: bounds origin
```

Актуальний код Athens записано в методі *renderAthens*, а результат зберігається в змінній екземпляра *surface*.

```
AthensHello >> renderAthens
   | font |
   font := LogicalFont familyName: 'Arial' pointSize: 10.
   surface drawDuring: [ :canvas |
      surface clear.
      canvas
         setPaint: ((LinearGradientPaint from: 0@0  to: self extent)
         colorRamp: {  0 -> Color white. 1 -> Color black }).
      canvas drawShape: (0@0 extent: self extent).
      canvas setFont: font.
      canvas setPaint: Color red.
      canvas
         pathTransform translateX: 20 Y: 20 + (font getPreciseAscent);
         scaleBy: 2; rotateByDegrees: 32.
      canvas drawString: 'Hello Athens in Pharo/Morphic' ]
```

Відкриємо морфу у вікні за допомогою коду

```
AthensHello new openInWindow
```

Рис. 16.3. Морфа *AthensHello* відкрита в окремому вікні

Результат зображено на рис. 16.3. Він відрізняється від рис. 16.1 тільки розміром і написом у заголовку вікна.

16.6. Опрацювання зміни розміру

Тепер можна створити вікно і побачити гарний градієнт з текстом привітання. Проте легко помітити, що під час зміни розміру вікна вміст Athens не змінюється. Щоб це виправити, потрібен один додатковий метод.

```
AthensHello >> extent: aPoint
   | newExtent |
   newExtent := aPoint rounded.
   (bounds extent closeTo: newExtent) ifTrue: [ ^ self ].
   bounds := bounds topLeft extent: newExtent.
   surface := AthensCairoSurface extent: newExtent.
   self layoutChanged.
   self changed
```

Вітаємо, ви створили своє перше морфо-вікно, вміст якого відображається за допомогою Athens. Тепер продемонструємо, як інтегрувати цю морфу в демонстратор Spec.

16.7. Використання морф зі Spec

Тепер, коли є готова морфа, можна використати її в демонстраторі, екземплярі класу *SpMorphPresenter*, як показано далі.

```
SpPresenter << #AthensHelloPresenter
   slots: { #morphPresenter };
   package: 'CodeOfSpec20Book'
```

Визначимо стандартний макет, щоб Spec знав, де її розмістити.

```
AthensHelloPresenter >> defaultLayout
   ^ SpBoxLayout newTopToBottom
        add: morphPresenter;
        yourself
```

У методі *initializePresenters* огорнемо морфу екземпляром *SpMorphPresenter*.

```
AthensHelloPresenter >> initializePresenters
   morphPresenter := self instantiate: SpMorphPresenter.
   morphPresenter morph: AthensHello new
```

Коли відкриємо демонстратор, то він відобразить морфу.

```
AthensHelloPresenter new open
```

Результат отримаємо точно такий, як на рис. 16.1.

16.8. Підсумки розділу

У цьому розділі роз'яснено, як Spec може використати переваги операцій з полотном, які надає Athens або Roassal, щоб отримати доступ до особливих засобів візуалізації.

Розділ 17

Налаштування Інспектора

Текст розділу спершу написала Іона Томас (Iona Thomas), ми дякуємо їй за дозвіл використати цей матеріал.

Інспектор – наш улюблений інструмент для перегляду об'єктів і взаємодії з ними. У Pharo інспектування об'єкта означає відкриття цього інструмента, завантаження в нього цікавого об'єкта і взаємодію з ним. Це ключовий інструмент під час розробки у Pharo. Він дає змогу переміщатися структурою об'єкта, переглядати стан змінних, змінювати їхні значення або надсилати повідомлення. Інспектор, як і більшість інструментів інтегрованого середовища розробки Pharo, написаний на Spec. Крім того, його можна розширити, щоб показати ту інформацію, яка найліпше підходить розробникові. Це та можливість, про яку йтиметься в цьому розділі.

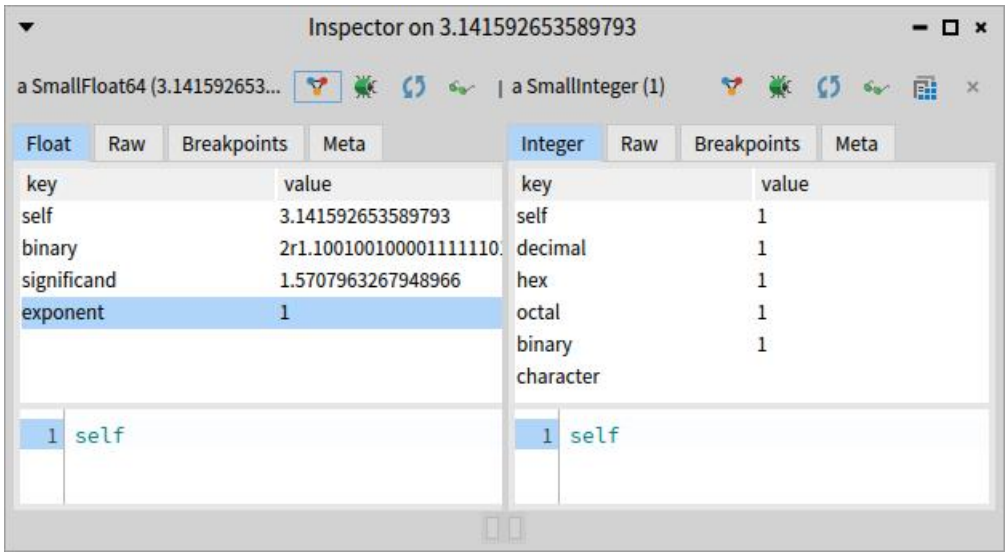


Рис. 17.1. Інспектування чисел

17.1. Створення власних вкладок

Якщо ви трохи користувалися Інспектором, то могли помітити, що деякі об'єкти мають додаткові вкладки, які відображаються в Інспекторі. Наприклад, перші вкладки *Float* та *Integer* зображають різні подання чисел, як на рис. 17.1.

Іншим прикладом є клас *FileReference*. Під час інспектування посилання на файл залежно від типу файлу з'являються різні вкладки з відповідною інформацією.

Створити нову вкладку Інспектора так само просто, як повторно використати наявний демонстратор Spec або визначити новий для конкретного випадку. Наприклад, можна визначити вкладку, яка відображає певну візуалізацію Roassal.

У наступних параграфах пояснено, як додати кілька додаткових вкладок Інспектора для екземплярів *OrderedCollection*. Цей клас уже має спеціальну вкладку, визначену в надкласі *Collection*, яка показує список його елементів.

17.2. Додавання вкладки з текстом

Додамо першу вкладку, що містить текст з описом першого елемента колекції. Для цього визначимо такий метод[17].

```
OrderedCollection << inspectionFirstElement
  <inspectorPresentationOrder: 1 title: 'First Element'>
  ^ SpTextPresenter new
    text: 'The first element is ', self first asString;
    beNotEditable;
    yourself
```

Трохи пояснимо визначення методу.

- *<inspectorPresentationOrder: 1 title: 'First Element'>* – анотація методу, прагма, яку виявляють у ході створення інспектора для об'єкта. Під час створення інспектора для екземпляра *OrderedCollection* цей метод використають для створення вкладки. Вона отримає заголовок *'First Element'* і позицію 1 у переліку вкладок.
- Вміст вкладки повертає анотований метод. Тут створено демонстратор тексту (*SpTextPresenter*), вмістом якого є невелике пояснення і значення першого елемента колекції. Зазначено також, що його не можна редагувати.

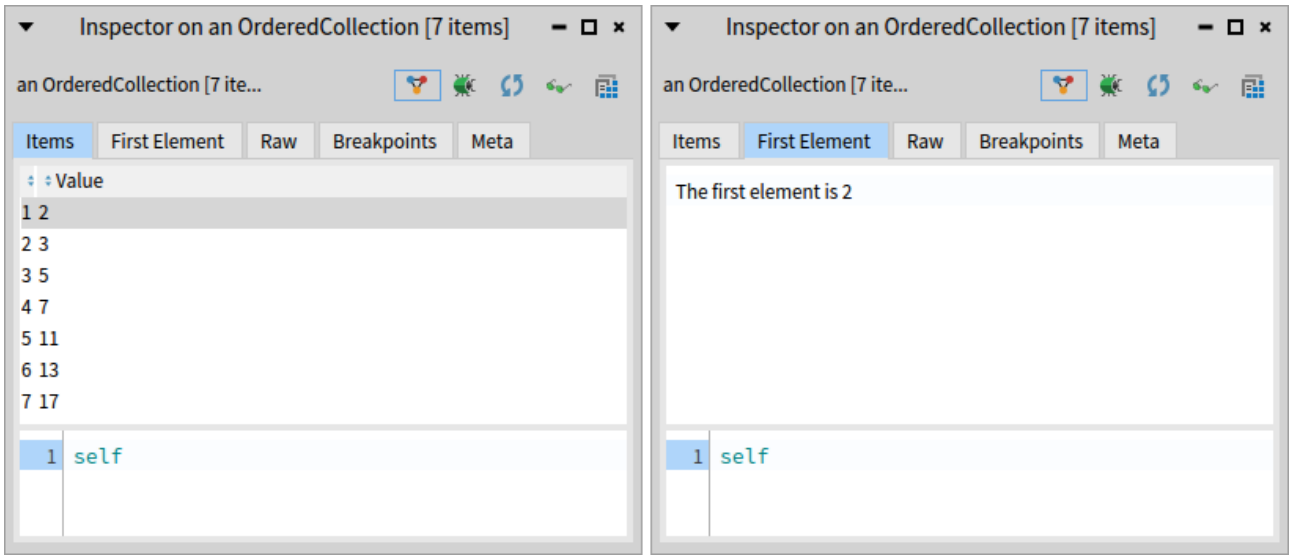


Рис. 17.2. Вкладки інспектора колекції: стандартна ***Items*** і додана ***First element***

Виконаємо такий фрагмент коду *(OrderedCollection withAll: #(2 3 5 7 11 13 17)) inspect* і отримаємо інспектора, як на рис. 17.2.

Зверніть увагу на те, що нова вкладка розташована на другій позиції. Так трапилося тому, що в методі *Collection<<inspectionItems:*, який визначає вкладку ***Items***, значення порядкового параметра задано 0.

[17] Метод доповнює функціонал системного класу *OrderdCollection*, тому доведеться спочатку відкрити Оглядача на цьому класі, а тоді визначити метод. Доповнення стосується навчального прикладу, тому доцільно зберегти метод у пакеті «CodeOfSpec20Book», де розташовано всі класи, створені в цій книзі. Для цього увімкніть перемикач ***extension*** у рядку стану редактора програмного коду і введіть ім'я пакета (прим. – *Ярошко С.*).

17.3. Вкладка з таблицею

Тепер створимо нову вкладку, яка відображатиме таблицю, якщо колекція містить тільки числа. Вона покаже всі числа і результат множення кожного на 2.

Спочатку створимо вкладку з таблицею[18].

```
OrderedCollection << inspectionMultipliedByTwo
  <inspectorPresentationOrder: 10 title: 'Multiply by 2'>
  | itemColumn multipliedByTwoColumn |
  itemColumn := SpStringTableColumn
    title: 'Item'
    evaluated: #yourself.
  itemColumn width: 30.
  multipliedByTwoColumn := SpStringTableColumn
    title: 'Multiply by 2'
    evaluated: [ :each | each * 2 ].
  ^ SpTablePresenter new
      addColumn: itemColumn;
      addColumn: multipliedByTwoColumn;
      items: self;
      beResizable;
      yourself
```

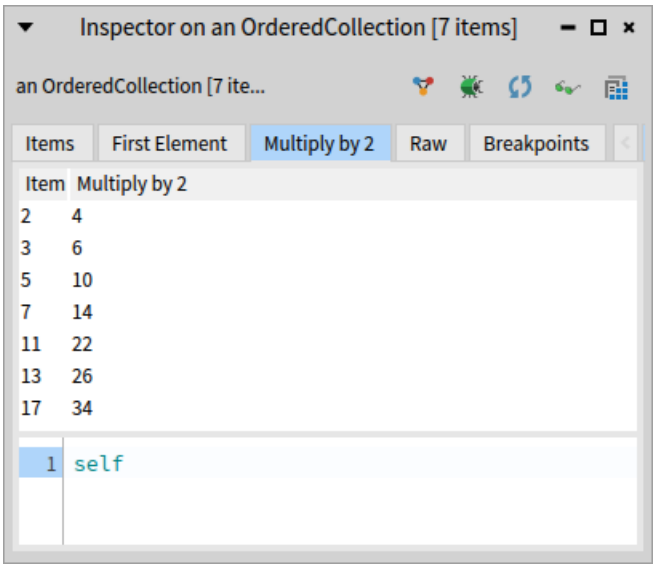


Рис. 17.3. Інспектор колекції з вкладкою ***Multiplied by 2***

Під час інспектування колекції чисел побачимо вкладки, зображені на рис. 17.3.

17.4. Умова активації вкладки

Якщо колекція містить елементи, які не є числами, то вкладка ***Multiplied by 2*** аварійно завершує роботу і виглядає як червоний прямокутник. Щоб запобігти такій неприємності, треба визначити умову активації вкладки. Таку умову оголошує метод *xContext:*, де *x* – назва методу, який визначає вкладку.

[18] Цей метод, як і попередній – метод розширення (прим. – *Ярошко С.*).

Наприклад, якщо нову вкладку визначає метод *inspectionMultipliedByTwo*, то умову активації вкладки оголошує метод *inspectionMultipliedByTwoContext:*. Визначимо його так.

```
OrderedCollection << inspectionMultipliedByTwoContext: aContext
  ^ aContext active: self containsOnlyNumbers

OrderedCollection << containsOnlyNumbers
  ^ self allSatisfy: [ :each | each isNumber ]
```

Ці два методи забезпечують відображення вкладки лише тоді, коли в колекції є лише числа.

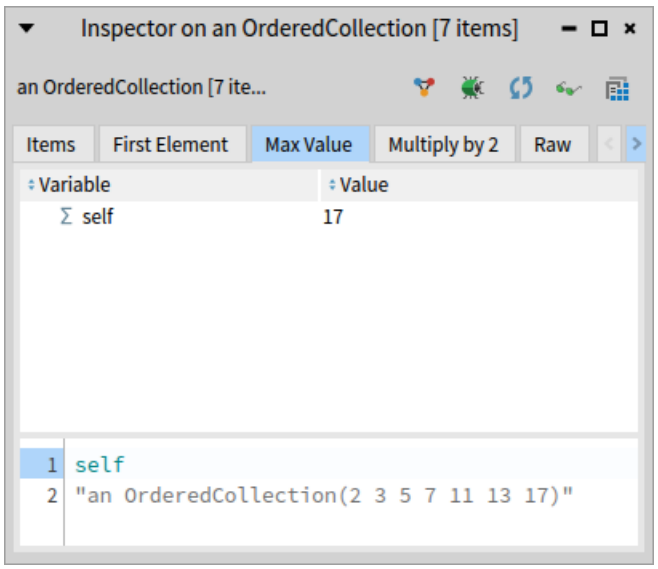


Рис. 17.4. Інспектор колекції з вкладкою ***Max Value***

17.5. Додавання необробленого подання елемента колекції

Іноді може виникнути потреба мати додаткову вкладку без будь-яких інтерпретацій вмісту. Це те, що називають необробленим поданням. Така вкладка використовує екземпляр *StRawInspectionPresenter*.

Додамо вкладку, яка показує необроблене подання максимального значення колекції.

```
OrderedCollection << inspectionMaxValue
  <inspectorPresentationOrder: 5 title: 'Max Value'>
  ^ StRawInspectionPresenter on: self max

OrderedCollection << inspectionMaxValueContext: aContext
  ^ aContext active: self containsOnlyNumbers
```

17.6. Вилучення панелі інтерпретатора

Як видно на рис. 17.4, значення *self* у вікні інтерпретатора не збігається з *self* у вкладці, що вводить в оману. Тому приховаємо панель інтерпретатора.

```
OrderedCollection << inspectionMaxValueContext: aContext
  aContext withoutEvaluator.
  ^ aContext active: self containsOnlyNumbers
```

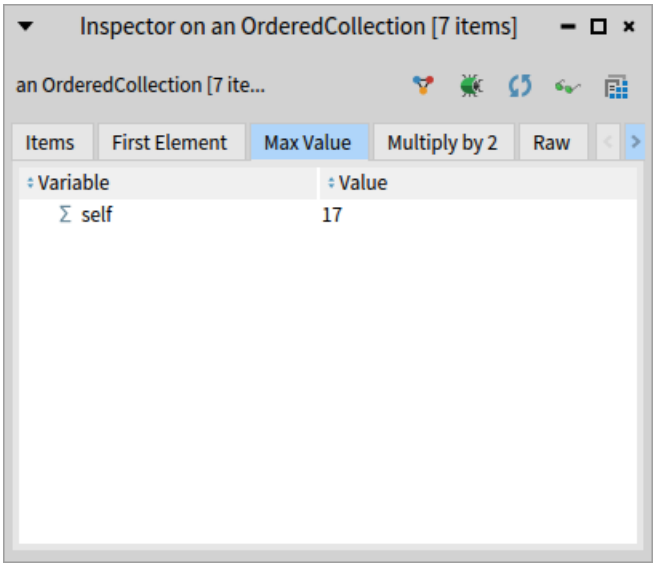


Рис. 17.5. Вилучення панелі інтерпретатора

Інспектуємо повторно ту саму колекцію і побачимо інспектор, як на рис. 17.5.

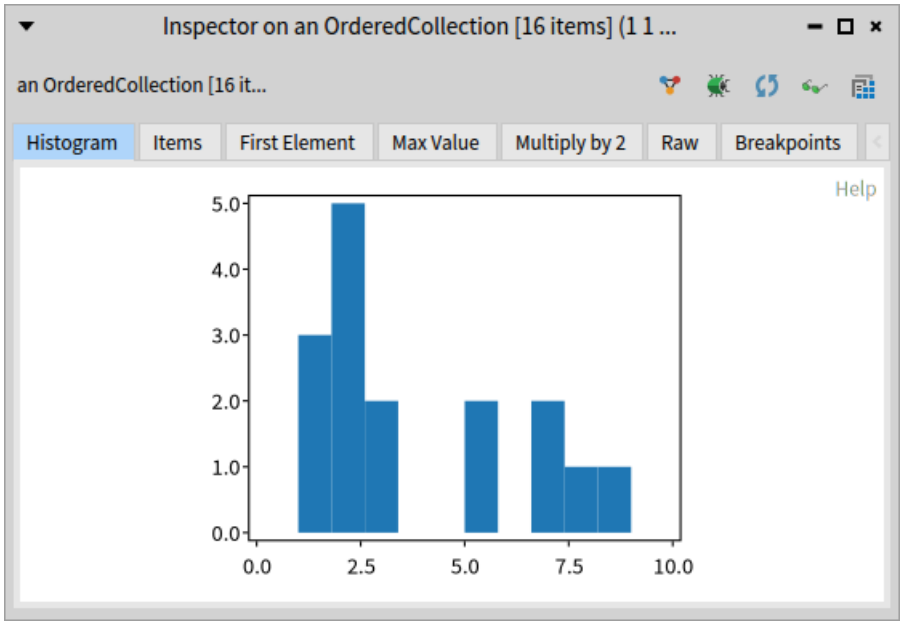


Рис. 17.6. Вкладка з гістограмою

17.7. Додавання діаграм Roassal

Roassal дає змогу створювати різні візуалізації. Їх також можна додати до вкладок інспектора. Бібліотека містить методи побудови деяких загальних графіків, як от гістограма. Додамо до інспектора гістограму значень, якщо в колекції є тільки числа. Візуалізацію Roassal можна вбудувати в демонстратор, надіславши повідомлення *asPresenter* екземпляру *RSBuilder*. У наведеному нижче коді *RSHistogramPlot* є підкласом *RSBuilder*.

Можна також використати *SpRoassalPresenter* або *SPRoassalInspectorPresenter*.

```
OrderedCollection << inspectionIntegerHistogram
  <inspectorPresentationOrder: -1 title: 'Histogram'>
  | plot |
  plot := RSHistogramPlot new x: self.
  ^ plot asPresenter

OrderedCollection << inspectionIntegerHistogramContext: aContext
  aContext active: self containsOnlyNumbers.
  aContext withoutEvaluator
```

Виконавши *#(1 1 3 2 5 2 2 1 9 3 2 2 5 7 7 8) asOrderedCollection inspect*, побачимо інспектор, зображений на рис. 17.6.

17.8. Підсумки розділу

У цьому розділі коротко описано, як можна розширити Інспектор, додавши певні вкладки. Така можливість допомагає формувати особливий спосіб перегляду своїх об’єктів і взаємодії з ними. Продемонстровано, як визначати умовні вкладки, а також вбудовувати візуалізації.

Частина III
Використання команд.
Приклади використання Spec

Розділ 18

Commander: потужний і простий програмний каркас для команд

Початкову бібліотеку Commander розробив Денис Кудряшов. Commander 2.0 є другою ітерацією цієї бібліотеки. Його спроєктували та розробили Жульєн Дельпланк (Julien Delplanque) і Стефан Дюкасс (Stéphane Ducasse). Зверніть увагу на те, що Commander 2.0 несумісний із Commander, але перейти з Commander на Commander 2.0 дуже легко. Тут описано Commander 2.0 у контексті Spec. Надалі, коли згадується Commander, то йдеться про Commander 2.0.

Щоб пояснити концепції, повернемося до розгляду застосунку «*Mail*», програми електронної пошти, яку подали в розділі 12 і розширили в розділі 13. Програмний код можна завантажити з GitHub, як описано в першому розділі цієї книги.

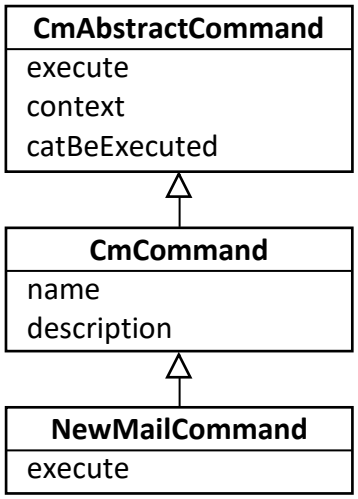


Рис. 18.1. Проста команда та її ієрархія

18.1. Команди

Commander моделює окремі дії застосунку об'єктами першого класу згідно з шаблоном проєктування Команда. Засобами Commander можна виражати команди, використовувати їх для створення меню та панелей інструментів, а також для написання з командного рядка скриптів застосунків.

Кожну дію моделює окремий клас команди (підклас *CmCommand*) з методом *execute* і станом, необхідним для виконання (див. рис. 18.1).

Пізніше ми продемонструємо, що для програмного каркаса інтерфейсу користувача потрібна додаткова інформація, наприклад, піктограма та опис гарячих клавіш. З'ясуємо також, як можна методом розширення оснастити команди додатковою функціональністю.

18.2. Визначення команд

Команда – це простий об'єкт, екземпляр підкласу класу *CmCommand*. Він має опис і назву. Назва може бути статичною або динамічною, як буде описано пізніше. Крім того, команда має контекст, з якого витягає інформацію, потрібну для свого виконання. У своїй основній формі команда більше нічого не має.

Далі розглянемо приклади. Визначимо деякі команди для застосунку «*Mail*» і проілюструємо як їх можна перетворити на меню, головне меню та панель інструментів.

18.3. Оголошення спільного надкласу команд

Для зручності визначимо спільний надклас усіх команд застосунку «*Mail*».

```
CmCommand << #MailClientCommand
   slots: {};
   package: 'CodeOfSpec20Book'
```

Щоб зробити код більш читабельним, визначимо простий допоміжний метод.

```
MailClientCommand >> mailClientPresenter
   ^ self context
```

18.4. Додавання головних команд

Реалізуємо підкласи *MailClientCommand*, щоб визначити команди створення нового електронного листа, зберігання, надсилання, видалення листа й отримання пошти.

NewMailCommand

```
MailClientCommand << #NewMailCommand
   slots: {};
   package: 'CodeOfSpec20Book'
```

У методі *initialize* задаємо назву та опис.

```
NewMailCommand >> initialize
   super initialize.
   self
      name: 'New';
      description: 'Create a new email'
```

Метод *execute* – це найважливіший метод, бо саме він описує, що робить команда. У ньому використано допоміжний метод *mailClientPresenter*, визначений у надкласі. Метод надсилає повідомлення *newMail*, визначене в класі *MailClientPresenter* в розділі 12.

```
NewMailCommand >> execute
   self mailClientPresenter newMail
```

Загалом, методи *execute* прості, бо в них немає достатньо інформації про стан програми, щоб знати, що робити. Тому вони часто делегують застосункові.

Можна сформулювати загальну пораду щодо проєктування команди: не визначайте логіку поведінки застосунку в команді. Команда – це лише представник цієї поведінки.

SaveMailCommand

```
MailClientCommand << #SaveMailCommand
   slots: {};
   package: 'CodeOfSpec20Book'

SaveMailCommand >> initialize
   super initialize.
   self
      name: 'Save';
      description: 'Save the email'

SaveMailCommand >> execute
   self mailClientPresenter saveMail

SaveMailCommand >> canBeExecuted
   ^ self mailClientPresenter hasDraft
```

Визначення цієї команди ілюструє, як можна керувати доступністю команди. Метод *canBeExecuted* визначає умову, за якої можна виконати команду.

У попередньому класі команди ми не реалізували *NewMailCommand >> canBeExecuted*, тому що завжди можна створити новий лист, а за замовчуванням команди можна виконувати (*CmCommand >> canBeExecuted* відповідає *true*).

SendMailCommand

```
MailClientCommand << #SendMailCommand
   slots: {};
   package: 'CodeOfSpec20Book'

SendMailCommand >> initialize
   super initialize.
   self
      name: 'Send';
      description: 'Send the selected email'

SendMailCommand >> execute
   self mailClientPresenter sendMail

SendMailCommand >> canBeExecuted
   ^ self mailClientPresenter hasDraft
```

DeleteMailCommand

```
MailClientCommand << #DeleteMailCommand
   slots: {};
   package: 'CodeOfSpec20Book'

DeleteMailCommand >> initialize
   super initialize.
   self
      name: 'Delete';
      description: 'Delete the selected email'
```

```
DeleteMailCommand >> execute
   ^ self mailClientPresenter deleteMail

DeleteMailCommand >> canBeExecuted
   ^ self mailClientPresenter hasSelectedEmail
```

FetchMailCommand

```
MailClientCommand << #FetchMailCommand
   slots: {};
   package: 'CodeOfSpec20Book'

FetchMailCommand >> initialize
   super initialize.
   self
      name: 'Fetch';
      description: 'Fetch email from the server'

FetchMailCommand >> execute
   self mailClientPresenter fetchMail
```

18.5. Додавання команд-заповнювачів

Визначимо також команди-заповнювачі для функцій, які не були реалізовані в застосунку «*Mail*» у розділі 13. Не будемо реалізовувати їх і тут. Надамо лише назву та опис, необхідні для інтерфейсу користувача.

FormatPlainTextCommand

```
MailClientCommand << #FormatPlainTextCommand
   slots: {};
   package: 'CodeOfSpec20Book'

FormatPlainTextCommand >> initialize
   super initialize.
   self
      name: 'Plain text';
      description: 'Use plain text'
```

FormatRichTextCommand

```
MailClientCommand << #FormatRichTextCommand
   slots: {};
   package: 'CodeOfSpec20Book'

FormatRichTextCommand >> initialize
   super initialize.
   self
      name: 'Rich text';
      description: 'Use rich text'
```

ShowCcFieldCommand

```
MailClientCommand << #ShowCcFieldCommand
```

```
        slots: {};
        package: 'CodeOfSpec20Book'

    ShowCcFieldCommand >> initialize
        super initialize.
        self
            name: 'Show CC field';
            description: 'Turn on the CC field'
```

ShowBccFieldCommand

```
    MailClientCommand << #ShowBccFieldCommand
        slots: {};
        package: 'CodeOfSpec20Book'

    ShowBccFieldCommand >> initialize
        super initialize.
        self
            name: 'Show BCC field';
            description: 'Turn on the BCC field'
```

18.6. Перетворення команд на пункти меню

Тепер, коли команди визначені, можна перетворити їх на меню. У Spec перетворені на пункти меню команди структуровані в дерево екземплярів команд. Метод *buildCommandsGroupWith:forRoot:* класу *SpPresenter* відіграє роль зачіпки, яка дає змогу демонстраторам визначати корінь дерева екземплярів команд.

Метод *buildCommandsGroupWith:forRoot:* реєструє команди і кожній передає екземпляр демонстратора як контекст. Зауважимо, що тут додано тільки прості команди, але так само можна створювати й групи. Наразі ми обмежуємо метод створенням контекстного меню для *MailAccountPresenter*. Пізніше в цьому розділі у цьому ж методі створимо рядок головного меню та панель інструментів.

```
    MailClientPresenter class >> buildCommandsGroupWith: presenter
                                 forRoot: rootCommandGroup
        rootCommandGroup
            register: (self buildAccountMenuWith: presenter)
```

Цей метод делегує методу *MailClientPresenter class >> buildAccountMenuWith:*, який додає команди видалення та надсилання листа. Метод повертає екземпляр *CmCommandGroup* з назвою *'AccountMenu'*. Ім'я не буде видно в інтерфейсі користувача, групі надіслали повідомлення *beRoot*. Екземпляр команди перетворюється на команду для Spec за допомогою повідомлення *forSpec*.

```
    MailClientPresenter class >> buildAccountMenuWith: presenter
        ^ (CmCommandGroup named: 'AccountMenu') asSpecGroup
              beRoot;
              register: (DeleteMailCommand forSpec context: presenter);
              register: (SendMailCommand forSpec context: presenter);
              yourself
```

18.7. Використання *fillWith:*

У розділі 13 йшлося про створення меню для застосунку «*Mail*», зокрема контекстного. Для цього визначили метод *MailClientPresenter >> accountMenu*, який повертає екземпляр меню для *MailAccountPresenter*. З використанням команд меню будують інакше. Його заповнюють командами, отриманими з дерева команд (визначеними в методі *buildAccountMenuWith:*). Демонстратор має доступ до кореня дерева команд через повідомлення *rootCommandsGroup*. Доступ до піддерев можна отримати, надіславши повідомлення / (похила риска). За допомогою команд створити контекстне меню[19] дуже просто, адже назва, підказка, реакція вже інкапсульовані в команді.

```
MailClientPresenter >> accountMenu
   ^ self newMenu
         fillWith: (self rootCommandsGroup / 'AccountMenu');
         yourself
```

Якщо відкрити застосунок за допомогою такого фрагмента коду,

```
(MailClientPresenter on: MailAccount new) open
```

то можна побачити пункти меню, як на рис. 18.2[20]. Згодом продемонструємо, що можна замінити пункт меню на інший, змінивши його назву або піктограму на льоту.

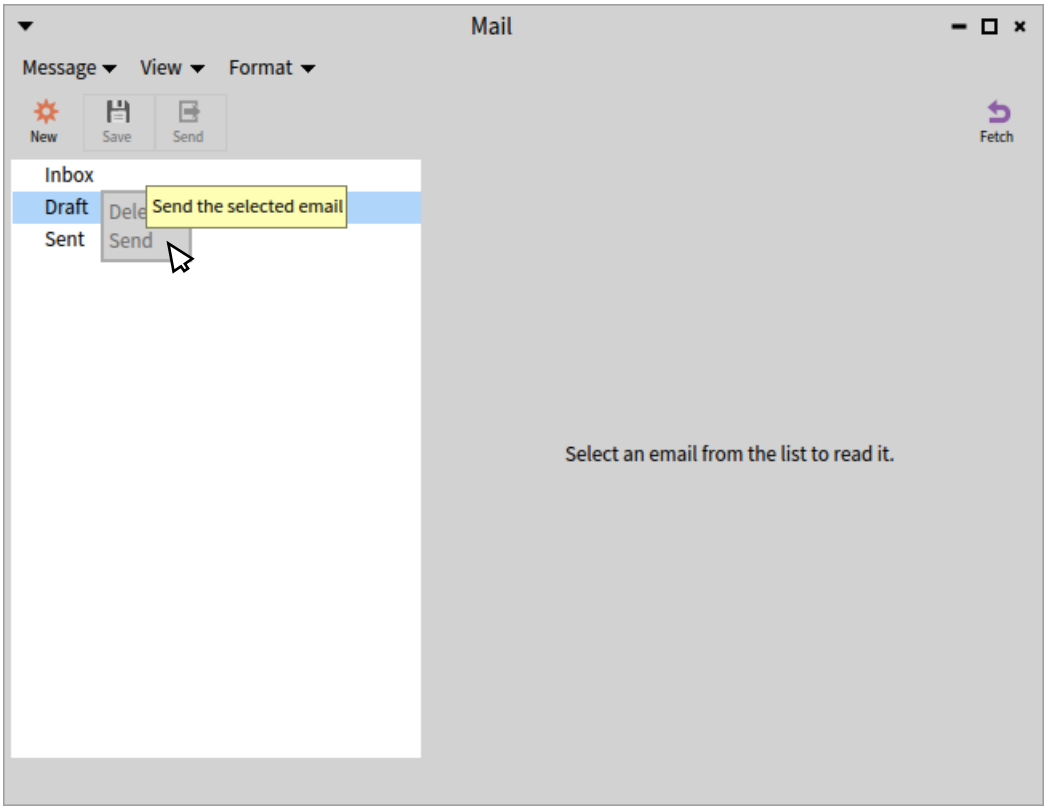


Рис. 18.2. Контекстне меню з двох пунктів, на другий з яких наведено мишку

[19] Якщо читач відтворює написаний тут код у Pharo, то йому доведеться перетворити на коментар попередню версію визначення методу *accountMenu* (прим. – *Ярошко С.*).
[20] На рисунку видно також головне меню та панель інструментів, створені в розділі 13 без використання команд. Згодом їх також буде перебудовано (прим. – *Ярошко С.*).

18.8. Керування піктограмами та клавіатурними скороченнями

Зазвичай команда не знає про особливу для Spec поведінку, бо команда не повинна бути пов'язана з інтерфейсом користувача. Але зрозуміло, що в ході розробки інтерактивного застосунку добре було б мати можливість додавати до пунктів меню піктограми та комбінації гарячих клавіш.

Commander підтримує додавання піктограм і комбінацій клавіш до екземплярів команд. Подивимося, як це працює з погляду користувача. Фреймворк надає два методи для встановлення піктограми та клавіш швидкого доступу відповідно: *iconName:* і *shortcutKey:*. Їх викликають у спеціалізації методу *asSpecCommand*.

```
DeleteMailCommand >> asSpecCommand
   ^ super asSpecCommand
         shortcutKey: $d meta;
         yourself

FetchMailCommand >> asSpecCommand
   ^ super asSpecCommand
         iconName: #refresh;
         shortcutKey: $f meta;
         yourself

NewMailCommand >> asSpecCommand
   ^ super asSpecCommand
         iconName: #smallNew;
         shortcutKey: $n meta;
         yourself

SaveMailCommand >> asSpecCommand
   ^ super asSpecCommand
         iconName: #smallSave;
         shortcutKey: $s meta;
         yourself

SendMailCommand >> asSpecCommand
   ^ super asSpecCommand
         iconName: #smallExport;
         shortcutKey: $l meta;
         yourself
```

Нагадаємо, що екземпляр команди створюють за допомогою повідомлення *forSpec*. Це повідомлення відповідає за виклик *asSpecCommand*.

18.9. Створення головного меню

Commander також підтримує створення рядка меню такого, як на рис. 13.2. Логіка така сама, як і для контекстних меню: потрібно визначити групу та зареєструвати її під заданим коренем, а потім веліти демонстраторові використати її як рядок меню.

Перший крок – визначити рядок меню. Для цього доповнимо визначений раніше метод.

```
MailClientPresenter class >> buildCommandsGroupWith: presenter
                             forRoot: rootCommandGroup
```

```
    rootCommandGroup
        register: (self buildAccountMenuWith: presenter);
        register: (self buildMenuBarGroupWith: presenter)
```

Він делегує методові *MailClientPresenter >> buildMenuBarGroupWith:*.

```
MailClientPresenter class >> buildMenuBarGroupWith: presenter
    ^ (CmCommandGroup named: 'MenuBar') asSpecGroup
        beRoot;
        register: (self buildMessageMenuWith: presenter);
        register: (self buildViewMenuWith: presenter);
        register: (self buildFormatMenuWith: presenter);
        yourself
```

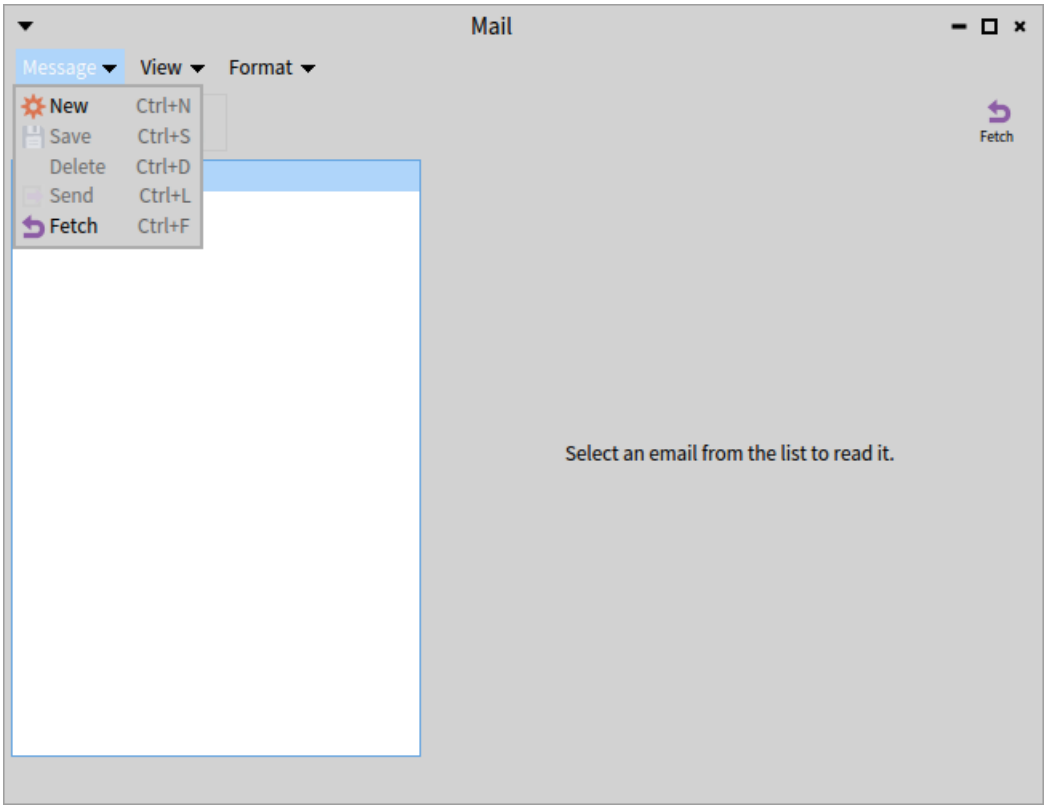


Рис. 18.3. Головне меню застосунку

Зі свого боку цей метод делегує трьом іншим.

```
MailClientPresenter class >> buildMessageMenuWith: presenter
    ^ (CmCommandGroup named: 'Message') asSpecGroup
        register: (NewMailCommand forSpec context: presenter);
        register: (SaveMailCommand forSpec context: presenter);
        register: (DeleteMailCommand forSpec context: presenter);
        register: (SendMailCommand forSpec context: presenter);
        register: (FetchMailCommand forSpec context: presenter);
        yourself

MailClientPresenter class >> buildViewMenuWith: presenter
    ^ (CmCommandGroup named: 'View') asSpecGroup
        register: (ShowCcFieldCommand forSpec context: presenter);
        register: (ShowBccFieldCommand forSpec context: presenter);
        yourself
```

```
MailClientPresenter >> buildFormatMenuWith: presenter
   ^ (CmCommandGroup named: 'Format') asSpecGroup
      register: (FormatPlainTextCommand forSpec context: presenter);
      register: (FormatRichTextCommand forSpec context: presenter);
      yourself
```

Тепер, коли дерево команд для рядка меню визначене, можна його використати для побудови головного меню. У розділі 13 було визначено метод *MailClientPresenter >> initializeMenuBar*. Тепер можна замінити його реалізацію.

```
MailClientPresenter >> initializeMenuBar
   menuBar := self newMenuBar.
   menuBar fillWith: self rootCommandsGroup / 'MenuBar'
```

Рис. 18.3 демонструє результат побудови меню з використанням команд.

18.10. Впровадження груп

Як видно на рис. 18.3, перший розділ головного меню відображає звичайний список пунктів меню. Воно було не таким у першій версії застосунку. У розділі 13 на рис. 13.2 зображено меню з двох частин. Перші чотири пункти меню відокремлені від п'ятого розділювальною лінією.

Можна досягти такого ж поділу за допомогою команд. Команди можна організувати в групи, а такі групи можна перетворити на відповідні частини розділу меню.

Внесімо необхідні зміни у визначений раніше метод *MailClientPresenter class >> buildMessageMenuWith:*.

```
MailClientPresenter class >> buildMessageMenuWith: presenter
   | fetchGroup |
   fetchGroup := CmCommandGroup new asSpecGroup
      register: (FetchMailCommand forSpec context: presenter);
      beDisplayedAsGroup;
      yourself.
   ^ (CmCommandGroup named: 'Message') asSpecGroup
      register: (NewMailCommand forSpec context: presenter);
      register: (SaveMailCommand forSpec context: presenter);
      register: (DeleteMailCommand forSpec context: presenter);
      register: (SendMailCommand forSpec context: presenter);
      register: fetchGroup;
      yourself
```

Ми вже використовували групи для створення розділів головного меню. Наприклад, у цьому методі створено групу з назвою «*Message*». Ця ж назва буде написом в рядку меню, назвою розділу меню. Замість того, щоб реєструвати *FetchMailCommand* як останню команду в групі, зареєстровано нову групу, визначену на початку методу. Нова група містить лише одну команду, *FetchMailCommand*, і не має назви, бо вона не потрібна.

Важливим є повідомлення *beDisplayedAsGroup*. Воно засвідчує те, що нову групу в меню треба відокремити від інших пунктів лінією, а не додати як пункт меню з вкладеним підменю. На рис. 18.4 зображено, як виглядатиме меню, якщо *beDisplayedAsGroup* не

надсилати. Це було б добре, якби було потрібне вкладене меню. Але в такому випадку варто давати групі гарну назву, бо «*Unnamed group*» є назвою за замовчуванням.

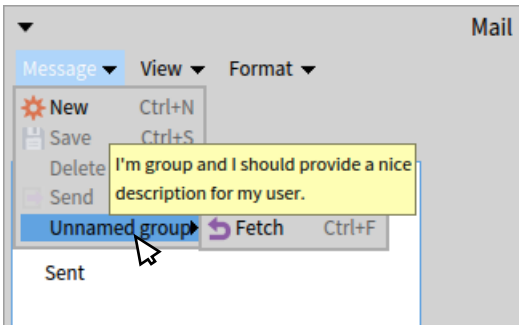


Рис. 18.4. Вкладене меню, на назву якого наведено вказівник мишки

У цьому випадку вкладене меню не потрібне, треба відокремити частину пунктів меню, тому реалізація *MailClientPresenter class >> buildMessageMenuWith:* така, як показано вище. З нею отримаємо меню, зображене на рис. 18.5. Як і в розділі 13, тепер є дві групи команд, розділених лінією.

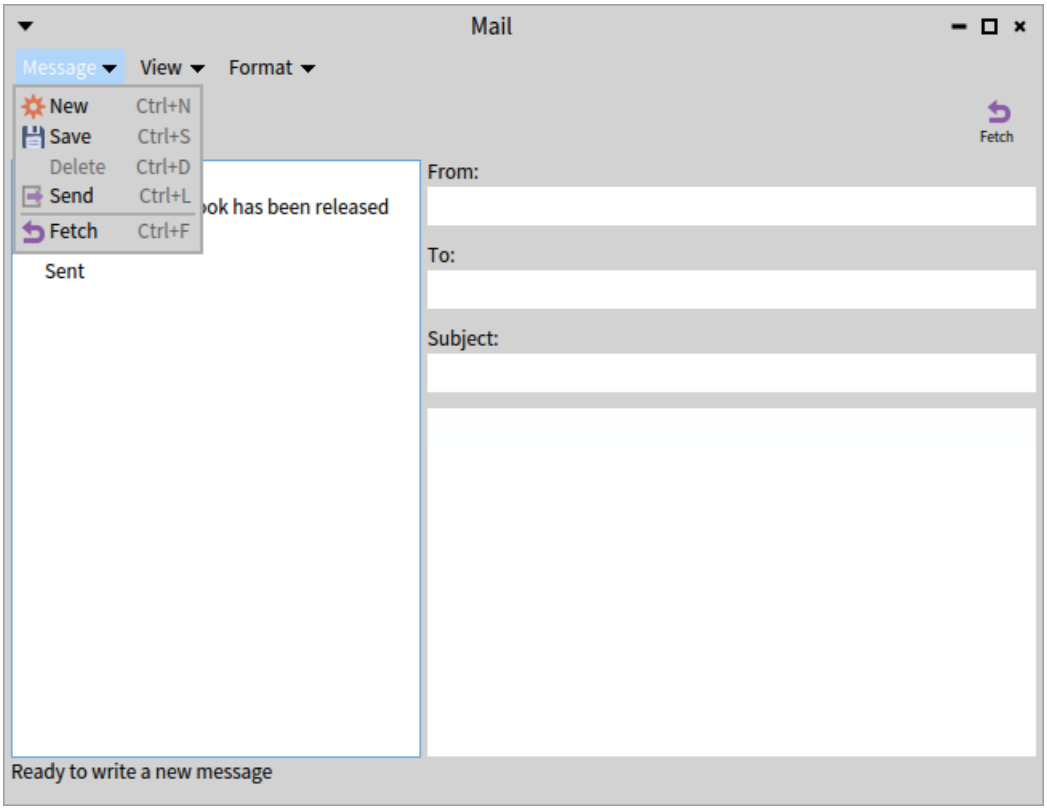


Рис. 18.5. Розділ меню, поділений на частини

18.11. Розширення меню

Створення меню – це добре, але іноді потрібно додати розділ до вже наявного меню. Commander підтримує таку можливість через спеціальну прагму *<extensionCommands>*, яка ідентифікує розширення.

Уявіть, що виникла потреба додати нову функціональність до застосунку «*Mail*», і вона постачається в іншому пакеті, наприклад, *'CodeOfSpec20Book-Extensions'*. Як приклад

такої ситуації, додамо можливість створювати нові листи за шаблоном. Щоб не нагромаджувати додатковий програмний код, зробимо все спрощено. Запровадження повноцінної системи шаблонів не буде метою. Обмежимося створенням шаблона для основної частини листа.

Визначення нової команди

Спочатку визначимо нову команду, а тоді продемонструємо, як можна розширити наявний рядок меню додатковим розділом. Подібно до цього можна додавати нові пункти до готового меню й інструментальні кнопки до наявної панелі інструментів.

```
MailClientCommand << #NewMailTemplateCommand
   slots: { #bodyTemplate };
   package: 'CodeOfSpec20Book-Extensions'
```

Зверніть увагу на те, що новий клас команди помістили в пакет розширення, тоді як увесь інший код залишається в пакеті *'CodeOfSpec20Book'*.

Можна уявити собі, що шаблон постачає значення для всіх атрибутів листа, але, як вже було сказано, робимо все спрощено, тому тільки текст листа шаблонний. Саме тому оголосили лише одну змінну екземпляра. Щоб задати шаблон тексту, знадобиться метод-модифікатор, тому визначимо його.

```
NewMailTemplateCommand >> bodyTemplate: aString
   bodyTemplate := aString
```

Згодом створимо кілька шаблонів, їхні назви стануть назвами команд, а поки що в методі *initialize* задамо шаблону стандартну назву.

```
NewMailTemplateCommand >> initialize
   super initialize.
   self
      name: 'New template';
      description: 'Create a new email from a template'
```

Метод *execute* делегує створення нового листа екземплярові *MailClientPresenter* подібно до того, як реалізовано методи *execute* інших класів команд.

```
NewMailTemplateCommand >> execute
   self mailClientPresenter newFromTemplate: bodyTemplate
```

Така реалізація потребує додавання методу розширення до класу *MailClientPresenter*. Поданий нижче метод зараховано до пакета *'CodeOfSpec20Book-Extensions'*. Його реалізація подібна до *MailClientPresenter >> newMail*. Єдина відмінність полягає в надсиланні новому листу повідомлення *body:*, щоб задати текстові листа шаблонне значення.

```
MailClientPresenter >> newFromTemplate: aString
   editedEmail := Email new.
   editedEmail body: aString.
   reader updateLayoutForEmail: editedEmail.
   self updateToolBarButtons.
   statusBar pushMessage: 'Ready to write a new message from template'
```

18.12. Оголошення розширення

Остання відсутня деталь пазла – оголошення розширення дерева команд із прагмою *<extensionCommands>* методом класу *MailClientPresenter*.

```
MailClientPresenter class >> buildTemplateCommandsGroupWith: presenter
                             forRoot: rootCommandGroup
   <extensionCommands>

   (rootCommandGroup / 'MenuBar')
      register: (self buildTemplateMenuWith: presenter)
```

Цей метод міститься в пакеті *'CodeOfSpec20Book-Extensions*'. Як і попередній схожий, він використовує інший метод з інструкціями для створення дерева команд нового розділу меню. Метод створення також зараховано до пакета *'CodeOfSpec20Book-Extensions*'. У нашому прикладі він створює тільки дві команди. У розширеній реалізації можна було б уявити, що шаблони є об'єктами, і їх створюють деінде.

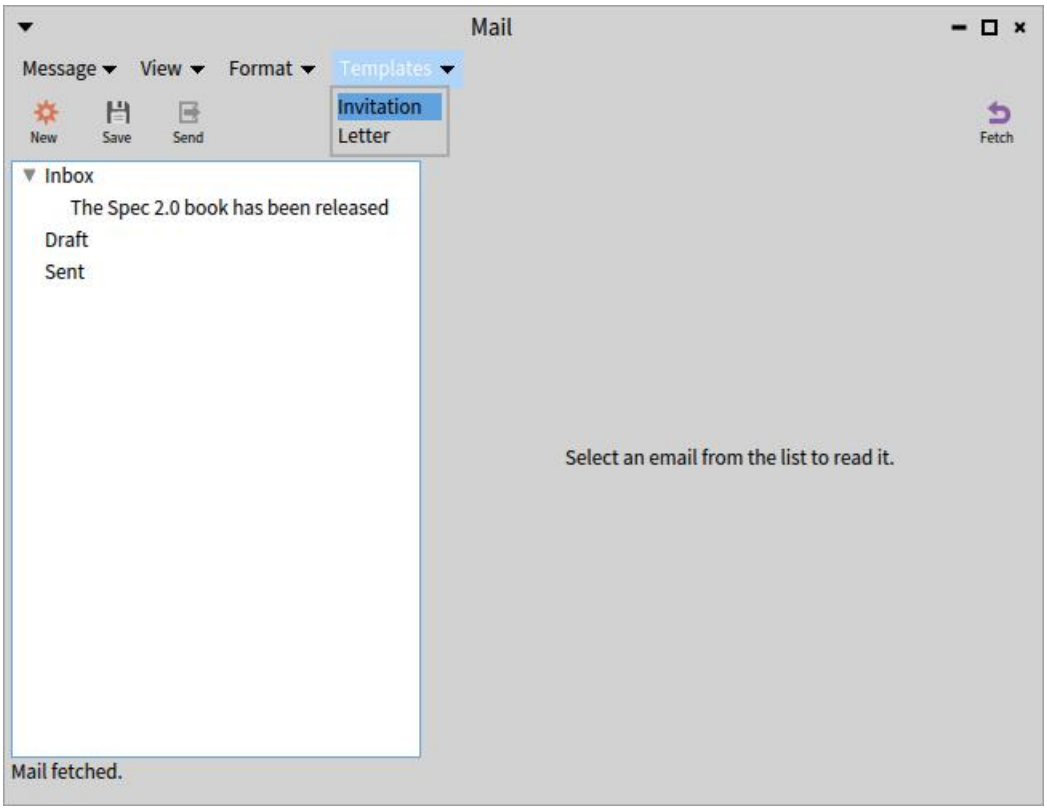


Рис. 18.6. Застосунок з доповненим рядком меню

Зверніть увагу на відмінність способу створення команд від застосованого раніше. Тут не надсилаємо *forSpec* класам команд. Насправді, цього не можна зробити, бо кожну команду потрібно ініціалізувати значенням шаблона тексту листа. Тому спочатку створюємо та налаштовуємо екземпляри команд, а тоді надсилаємо їм *asSpecCommand*. Контекст встановлюємо в момент реєстрації.

```
MailClientPresenter class >> buildTemplateMenuWith: presenter
   | letterTemplateCommand invitationTemplateCommand |
   invitationTemplateCommand := NewMailTemplateCommand new
      name: 'Invitation';
```

```
        bodyTemplate: 'Hi, you are invited to my party on <date>.';
        asSpecCommand.
    letterTemplateCommand := NewMailTemplateCommand new
        name: 'Letter';
        bodyTemplate:
            'Dear <name>, I write you to inform you about <something>.';
        asSpecCommand.
    ^ (CmCommandGroup named: 'Templates') asSpecGroup
        register: (invitationTemplateCommand context: presenter);
        register: (letterTemplateCommand context: presenter);
        yourself
```

Настав час знову відкрити застосунок «*Mail*». На рис. 18.6 показано отриманий результат із відкритим розділом меню ***Templates***. Якщо вибрати в ньому ***Invitation***, то шаблон нового листа з'явиться у вікні праворуч, як показано на рис. 18.7.

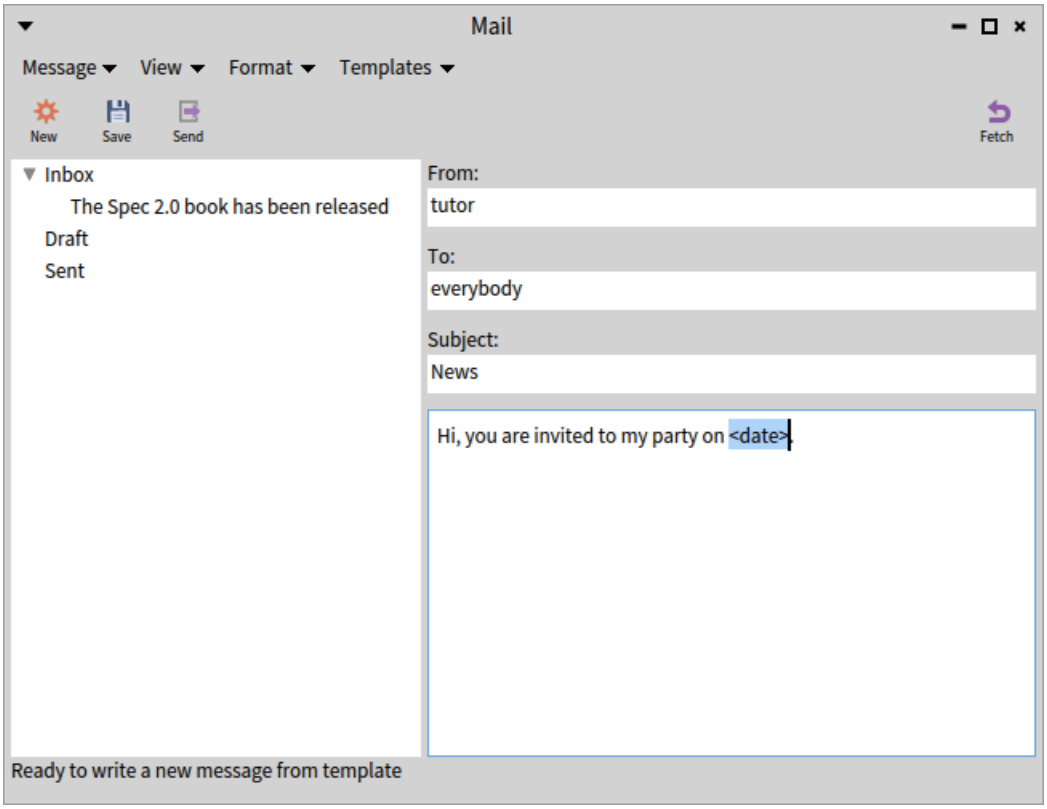


Рис. 18.7. Створення листа за шаблоном запрошення

18.13. Створення панелі інструментів

У попередніх параграфах описано, як використовувати команди для побудови контекстного та головного меню. Ті самі команди можна використовувати для формування панелі інструментів. Велика різниця між меню та панелями інструментів полягає в тому, що пункти меню відображаються лише на вимогу. Вони з'являються після того, як користувач розгорне меню. На противагу їм, кнопки панелі інструментів завжди видимі. Постійна видимість впливає на спосіб керування активністю кнопок. Яким має бути стан пунктів меню, увімкнений або вимкнений, визначається на момент їхнього відображення. А от кнопки панелі інструментів доведеться вмикати та вимикати в момент зміни стану програми. Обговоримо це незабаром. Зберімо спочатку

панель інструментів на основі команд. Уже відома схема дій для побудови меню. Така сама для панелей інструментів.

Першим кроком є визначення команд, які будуть доступні на панелі інструментів. Доповнимо метод побудови дерева команд востаннє.

```
MailClientPresenter class >> buildCommandsGroupWith: presenter
                             forRoot: rootCommandGroup
   rootCommandGroup
      register: (self buildAccountMenuWith: presenter);
      register: (self buildMenuBarGroupWith: presenter);
      register: (self buildToolBarGroupWith: presenter)
```

Остання реєстрація додає до дерева команди для панелі інструментів. Ось реалізація відповідного методу

```
MailClientPresenter class >> buildToolBarGroupWith: presenter
   ^ (CmCommandGroup named: 'ToolBar') asSpecGroup
      beRoot;
      register: (NewMailCommand forSpec context: presenter);
      register: (SaveMailCommand forSpec context: presenter);
      register: (SendMailCommand forSpec context: presenter);
      register: (FetchMailCommand forSpec context: presenter);
      yourself
```

Це дуже схоже на те, як раніше визначили команди для меню. Тут додано чотири команди, які хотілося би винести на панель інструментів.

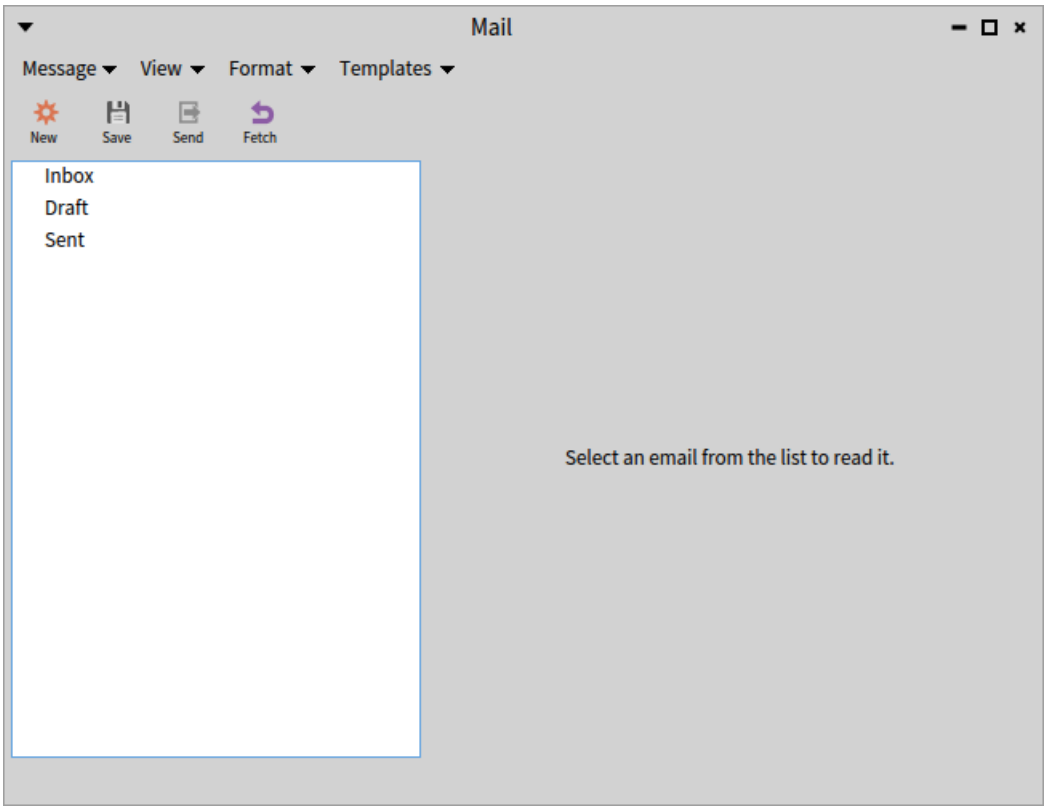


Рис. 18.8. Панель інструментів, збудована на основі команд

Наступним кроком буде заповнення панелі інструментів цими командами. Щоб досягти бажаного, змінимо метод, визначений раніше.

```
MailClientPresenter >> initializeToolBar
   toolBar := self newToolbar.
   toolBar fillWith: self rootCommandsGroup / 'ToolBar'
```

Як бачите, тут використано такий самий шаблон, що й у методах *MailClientPresenter >> accountMenu* та *MailClientPresenter >> initializeMenuBar*. Створюємо нову панель інструментів, а потім надсилаємо повідомлення *fillWith:*, щоб заповнити її командами, отриманими з дерева команд.

Якщо знову відкрити *MailClientPresenter*, то побачимо панель інструментів, як на рис. 18.8. Усі кнопки інструментів розташовані на ній біля лівого краю. Така панель відрізняється від зображеної на рис. 13.3 у розділі 13, де кнопка для отримання пошти розташована біля правого краю.

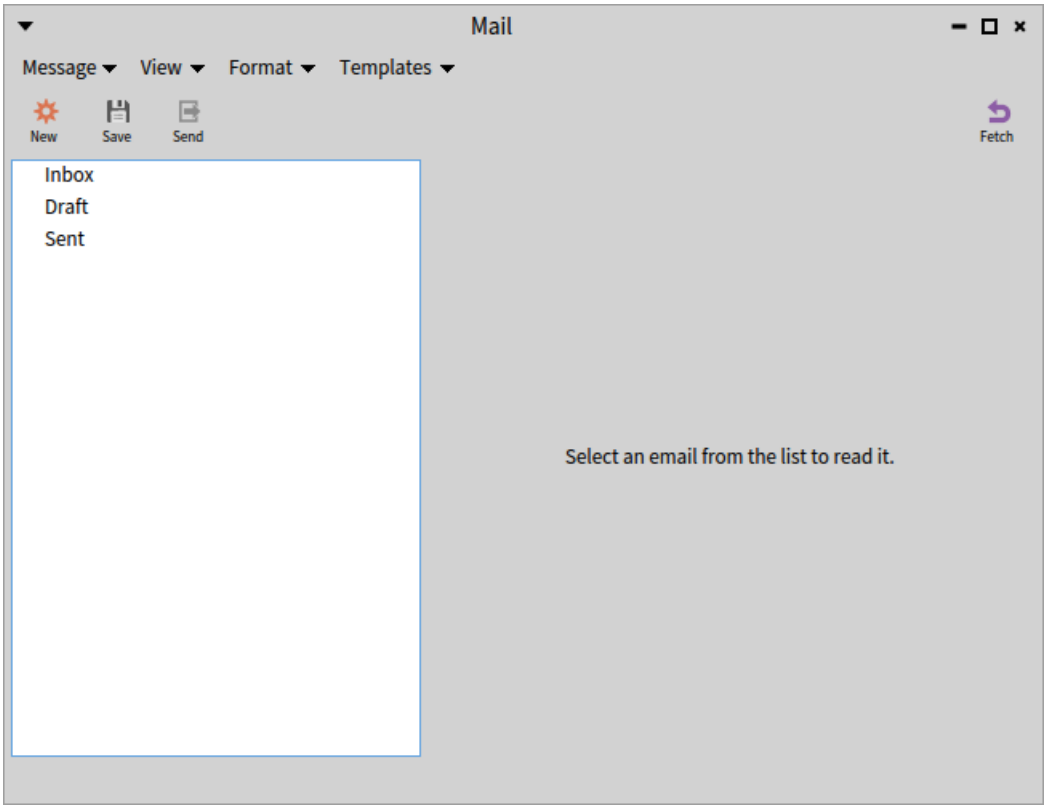


Рис. 18.9. Виправлена панель інструментів

Щоб розмістити кнопку ***Fetch*** праворуч, потрібна додаткова зміна в *FetchMailCommand >> asSpecCommand*. Команді надсилаємо повідомлення *beDisplayedOnRightSide*.

```
FetchMailCommand >> asSpecCommand
   ^ super asSpecCommand
         iconName: #refresh;
         shortcutKey: $f meta;
         beDisplayedOnRightSide;
         yourself
```

Коли знову відкрити демонстратор, то побачимо панель інструментів, як на рис. 18.9. Кнопка отримання пошти розташована біля правого краю.

Керування активністю кнопок

На початку цього параграфа ми пояснили, чому вмикати кнопки панелі інструментів потрібно інакше, ніж пункти меню. У поточному стані застосунку кнопки панелі інструментів постійно увімкнені. Це неправильно, бо натискання «не тієї» кнопки на панелі може призвести до аварійного переривання роботи застосунку.

Активність кнопок панелі інструментів треба оновлювати щоразу, коли змінюється стан застосунку. Наприклад, надіслати листа можна лише тоді, коли листа вибрано. У розділі 13 вже було описано метод *MailClientPresenter >> updateToolBarButtons* для оновлення стану кнопок панелі інструментів. Можна адаптувати його для взаємодії з командами. Трудність полягає в тому, що Spec не надає методу оновлення кнопок панелі інструментів. Єдиний спосіб – це повторно наповнити панель. Для цього можна надіслати повідомлення *fillWith:*, бо відповідний метод очищає панель інструментів, перш ніж наповнювати її знову.

```
MailClientPresenter >> updateToolBarButtons
   toolBar fillWith: self rootCommandsGroup / 'ToolBar'
```

Тепер, коли є такий метод, варто видалити дублювання коду в *MailClientPresenter >> initializeToolBar*.

```
MailClientPresenter >> initializeToolBar
   toolBar := self newToolbar.
   self updateToolBarButtons
```

18.14. Підсумки розділу

У цьому розділі описано, як можна визначити дерево команд і наповнити його піддеревами команд для певних контекстів. На основі цих піддерев можна створювати контекстні меню, головне меню та панель інструментів за допомогою лише кількох рядків коду. Ви дізналися, як команди можна налаштовувати та використовувати повторно. Групи команд надають спосіб структурування меню. Створення класів команд вартує потрачених зусиль, бо збирає в одному місці визначення функціональних можливостей застосунку, узгоджує поведінку відповідних пунктів різних меню та кнопок інструментів.

Розділ 19
Модель світлофора

Збудований у цьому розділі застосунок має дещо розважальний характер: мабуть не варто надто серйозно сприймати програму, яка імітує роботу вуличного світлофора. Проте наша мета цілком серйозна – продемонструвати використання описаних раніше засобів Spec у нових умовах і ознайомити читача з кількома новими можливостями. Ми інтегруємо за допомогою *SpMorphPresenter* морфи в застосунок Spec, використаємо новий різновид кнопки, тимчасово приховаємо частину інтерфейсу користувача, залучимо на допомогу таймер, щоб виконанням застосунку керував плин часу, а не тільки користувач.

19.1. Вимоги до програми

Потрібно створити віконний застосунок, який моделює роботу вуличного світлофора. Він зображає ліхтарі червоного, жовтого й зеленого кольорів, дає змогу перемикати їх у ручному й автоматичному режимах. Тривалість світіння кожного ліхтаря в автоматичному режимі налаштовує користувач. Зазвичай засоби налаштування приховані. Схематично вигляд застосунку зображено на рис. 19.1.

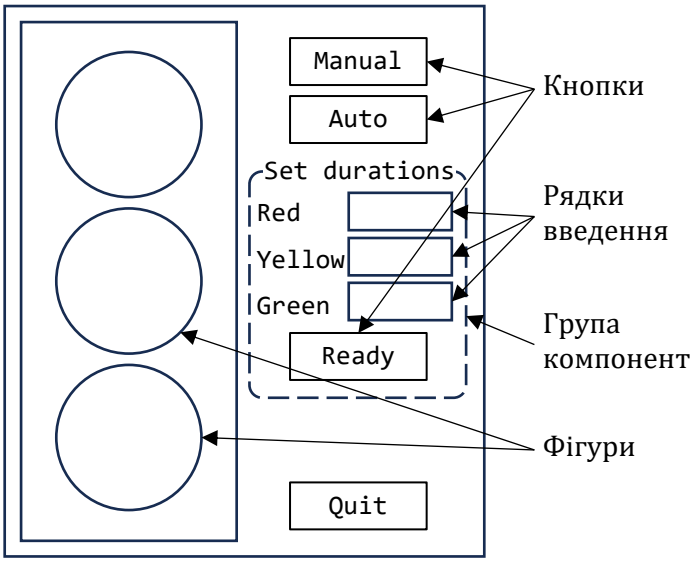


Рис. 19.1. Схема вікна застосунку «Світлофор»

Кнопка ***Manual*** виконує ручне перемикання світла, ***Auto*** відкриває панель *Set durations*, ***Ready*** приховує панель і переводить світлофор у автоматичний режим, а ***Quit*** завершує роботу застосунку. Для введення тривалостей можна використати поля введення тексту або чисел. Корпус світлофора та його ліхтарі можна зобразити спрощено за допомогою звичайних морф.

Щоб виконати це завдання, спроєктуємо класи моделі даних, демонстраторів і застосунку.

19.2. Графічні елементи

Ми не маємо на меті будувати реалістичне зображення світлофора, тому використаємо найпростіші графічні об'єкти – морфи. Наприклад, корпус можна зобразити прямокутником, екземпляром *BorderedMorph*, а ліхтарі – різноколірними кругами, екземплярами *CircleMorph*. Морфи, хоч і прості, мають чудову властивість: вони можуть володіти іншими, вкладеними морфами. Тому зображення світлофора можна сконструювати з прямокутника, який містить три вкладені круги.

Поекспериментуймо з фрагментами коду в Пісочниці, щоб з'ясувати, чи вдасться реалізувати задумане. Доведеться задати розміри та розташування фігур.

```
view := BorderedMorph new color: Color lightGray; extent: 70 @ 180.
view addAllMorphs: {
   (CircleMorph new color: Color red; borderColor: Color darkGray;
      extent: 50 @ 50; position: body position + (10 @ 10)).
   (CircleMorph new color: Color yellow; borderColor: Color darkGray;
      extent: 50 @ 50; position: body position + (10 @ 65)).
   (CircleMorph new color: Color green; borderColor: Color darkGray;
      extent: 50 @ 50; position: body position + (10 @ 120)) }.
view openInWorld.
```

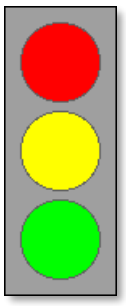


Рис. 19.2. Морфа, відображена командою *openInWorld*

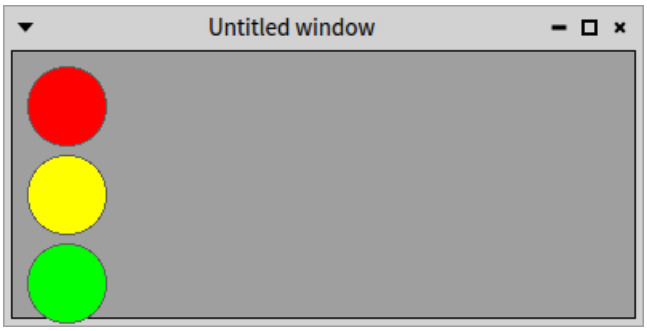


Рис. 19.3. Морфа, відкрита за допомогою *SpMorphPresenter*

Так отримаємо зображення, показане на рис. 19.2. Схоже на те, що треба, тільки розміри задамо більші, і придумаємо, як вимикати «ліхтарі». Тут *view* – це «самостійна» морфа, яка не належить ніяким вікнам середовища. Її можна легко перетягти мишкою, а закрити – відповідною командою меню-ореола, яке розгортають метаклацанням на морфі ([*Shift* + *Alt*] + *Click* у Windows), або програмно, виконавши *view delete*.

Проте нам потрібно зробити *view* частиною вікна. Використаємо для цього відповідний демонстратор – *SpMorphPresenter*.

```
smp := SpMorphPresenter new morph: view.
smp open.
```

Отримаємо вікно, як на рис. 19.3. Видно, що демонстратор підлаштував розміри морфи до усталених розмірів вікна, причому розтягається тільки батьківська морфа, а розміри і розташування вкладених залишаються незмінними. Отож, щоб зберегти потрібний розмір зображення, використаємо згодом два вкладені макети *SpBoxLayout*. Пригадуємо, що за допомогою методу *add:withConstraints:* можна задати висоту у вертикальному макеті та ширину в горизонтальному.

19.3. Модель світлофора

Для початку оголосимо класи, які моделюватимуть дані, з якими доведеться мати справу застосункові.

Ліхтар – самостійний об'єкт

Кожен ліхтар має низку властивостей: розміри, координати, колір, тривалість світіння. Він може вмикатися та вимикатися. Усе це свідчить про те, що ми маємо справу з об'єктом реального світу, стан і поведінку якого варто моделювати окремим класом, наприклад, *Lamp*. Очевидно, що розміри та координати належать до властивостей графічного компонента, відповідального за зображення ліхтаря, а екземплярові *Lamp* достатньо зберігати посилання на нього. Вмикання та вимикання ліхтаря можна зображати зміною кольору його морфи: увімкнений ліхтар має «свій» колір, а вимкнений – деякий стандартний, наприклад, світло-сірий.

Розпочнімо роботу над застосунком з оголошення класу *Lamp*. Константні величини у Pharo прийнято задавати методами класу – так і зробимо, щоб визначити колір у вимкненому стані. Оголосимо також метод створення ліхтаря заданого кольору і тривалості світіння. Новостворений ліхтар вимкнений.

```
Object << #Lamp
   slots: { #color . #view . #duration . #switchedOn };
   tag: 'Model';
   package: 'TrafficLightsProject'

Lamp class >> passiveColor
   "color of an off lamp"
   ^ Color lightGray lighter

Lamp class >> new: aColor for: time on: aMorph
   "create an instance with the color, duration and view"
   ^ self basicNew   color: aColor;
      duration: time; view: aMorph;
      switchOff
```

Оголосимо методи-модифікатори, використані в *new:for:on:*, і методи увімкнення.

```
Lamp >> color: aColor
   Color := aColor

Lamp >> duration: aNumber
   duration := aNumber

Lamp >> view: aMorph
   view := aMorph

Lamp >> switchOff
   switchedOn := false.
   view color: self class passiveColor

Lamp >> switchOn
   switchedOn := true.
   view color: color

Lamp >> isSwitchedOn
   ^ switchedOn
```

Тестування ліхтарів

Оце й усе, але перш ніж рухатися далі, варто перевірити, чи правильно функціонує клас *Lamp*. Напишемо для цього модульні тести. Клас тестів розташуємо в тому ж пакеті, але в іншій категорії: не в *Model*, а в *Test*.

```
TestCase << #LampTest
   tag: 'Tests';
   package: 'TrafficLightsProject'
```

Достатньо перевірити налаштування ліхтаря відразу після створення та правильність перемикання. Морфу створимо безпосередньо в тілі тесту, щоб мати змогу контролювати її колір.

```
TestCase >> testCreation
     | morph color lamp |
   morph := CircleMorph new.
   color := Color yellow.
   lamp := Lamp new: color for: 5 on: morph.
   self deny: lamp isSwitchedOn.
   self assert: lamp duration equals: 5.
   lamp switchOn.
   self assert: morph color equals: color

TestCase >> testSwitchOnOff
     | morph color lamp |
   morph := CircleMorph new.
   color := Color green.
   lamp := (Lamp new: color for: 10 on: morph) switchOn.
   self assert: lamp isSwitchedOn.
   self assert: morph color equals: color.
   lamp switchOff.
   self deny: lamp isSwitchedOn.
   self assert: morph color equals: Lamp passiveColor.
   lamp switchOn.
   self assert: lamp isSwitchedOn.
   self assert: morph color equals: color
```

Світлофор керує роботою ліхтарів

Продовжимо роботу над моделлю. Роботою ліхтарів керує окремий клас – світлофор. Він повинен зберігати колекцію ліхтарів, пам'ятати номер увімкнутого, вміти перемикати ліхтарі по одному або автоматично. Створимо для цього клас *TrafficLights*.

```
Object << #TrafficLights
   slots: { #lamps . #activeLamp . #view . #mustRun . #stopwatch };
   tag: 'Model';
   package: 'TrafficLightsProject'
```

Тут оголошено такі змінні екземпляра: масив ліхтарів; номер увімкнутого ліхтаря; посилання на екземпляр *BorderedMorph*, який зображає світлофор на екрані та містить вкладені *CircleMorhp* ліхтарів; ознака того, чи увімкнуто автоматичний режим; секундомір. Важливо, щоб екземпляр класу можна було створити тільки з наявною морфою, тому визначимо такі методи класу.

```
TrafficLights class >> new
   ^ self error:
      'Should use newWith: aBorderedMorph to create an instance'

TrafficLights class >> newWith: aMorph
   ^ self basicNew
      view: aMorph; initialize
```

Метод-модифікатор *view:* – тривіальний, а метод ініціалізації заслуговує на увагу. Нам доведеться реалізувати алгоритм перемикання ліхтарів, але який колір потрібно увімкнути після жовтого: зелений чи червоний? Щоб не шукати відповідь на це запитання, збережемо в масиві *два* посилання на жовтий ліхтар. Тоді зможемо завжди перемикати на наступний (у припущенні, що після останнього знову йде перший).

```
TrafficLights >> initialize
   lamps := {
         (Lamp new: Color red for: 5 on: (view submorphs at: 1)).
         (Lamp new: Color yellow for: 1 on: (view submorphs at: 2)).
         (Lamp new: Color green for: 5 on: (view submorphs at: 3)).
         nil }.
   lamps at: 4 put: (lamps at: 2).
   activeLamp := 2.
   (lamps at: activeLamp) switchOn.
   mustRun := false.
   stopwatch := Stopwatch new
```

Новий світлофор розпочинає роботу з жовтого ліхтаря в ручному режимі перемикання. Тепер легко записати метод перемикання.

```
TrafficLights >> changeLamp
   (lamps at: activeLamp) switchOff.
   activeLamp := activeLamp % 4 + 1.
   (lamps at: activeLamp) switchOn
```

Його можна викликати в ручному або автоматичному режимі перемикання. У першому випадку виклик *changeLamp* стане відповіддю на клацання на відповідній кнопці у вікні застосунку, а в другому викликом мали б керувати зміни показів системного годинника. Важливо, щоб відслідковування часового інтервалу не заблокувало роботу всього застосунку. Тому запустимо його в окремому потоці. У Pharo це легко зробити за допомогою повідомлення *fork*, яке надсилають блокові. Автоматичним режимом керуватимуть такі методи.

```
TrafficLights >> run
   mustRun := true.
   [ [ mustRun ] whileTrue: [ stopwatch activate.
      [ stopwatch duration seconds < (lamps at: activeLamp) duration ]
         whileTrue: [  ].
      stopwatch reset.
      self changeLamp ] ] fork

TrafficLights >> stop
   mustRun := false
```

Вкладений цикл у методі *run* затримує виконання на час світіння активного ліхтаря, а зовнішній – пильнує за прапорцем *mustRun*: чи потрібно продовжувати перемикання.

Світлофор майже готовий. Залишилося тільки оголосити метод, який призначатиме ліхтарям тривалості світіння, отримані від користувача.

```
TrafficLights >> setDurations: anArray
   1 to: 3 do: [ :i | (lamps at: i) duration: (anArray at: i) ]
```

Тестування світлофора

Правильність функціонування світлофора перевіримо модульними тестами.

```
TestCase << #TrafficLightsTest
   slots: { #morph . #trafficLights };
   tag: 'Tests';
   package: 'TrafficLightsProject'
```

У цьому класі тестів не випадково оголошені змінні екземпляра *morph* і *trafficLights*. Використаємо їх, щоб створити екземпляр світлофора перед виконанням тестових методів. Таке налаштування контексту виконання тестів у системі SUnit середовища Pharo виконує метод *TestCase >> setUp*.

```
TrafficLightsTest >> setUp
   super setUp.
   morph := BorderedMorph new.
   3 timesRepeat: [ morph addMorph: CircleMorph new ].
   trafficLights := TrafficLights newWith: morph
```

Тепер перевіримо правильність створення світлофора та перемикання ліхтарів. У класі *TrafficLights* не визначено селектор, який би повертав номер активного ліхтаря, але ми зможемо перевірити кольори ліхтарів.

```
TrafficLightsTest >> testChangeLamp
   trafficLights changeLamp.
   self assert: (morph submorphs at: 2) color equals: Lamp passiveColor.
   self assert: (morph submorphs at: 3) color equals: Color green.
   trafficLights changeLamp; changeLamp.
   self assert: (morph submorphs at: 1) color equals: Color red.
   self assert: (morph submorphs at: 2) color equals: Lamp passiveColor.
   self assert: (morph submorphs at: 3) color equals: Lamp passiveColor

TrafficLightsTest >> testNew
   self should: [ TrafficLights new ] raise: Error

TrafficLightsTest >> testNewWith
   self assert: (morph submorphs at: 2) color equals: Color yellow
```

Можна було б додати ще тестів, але залишимо це як вправу для читача і перейдемо до проєктування вікна застосунку.

19.4. Панель керування світлофором

Введення тривалостей світіння

Найскладнішою частиною інтерфейсу, схематично зображеного на рис. 19.1, є панель *Set durations*. Збудуємо її за допомогою окремого класу демонстратора. Він міститиме змінні екземпляра для зберігання компонент введення чисел і кнопки. Як відомо з попередніх розділів, для створення написів змінні екземпляра не обов'язкові.

```
SpPresenter << #InputPanelPresenter
   slots: { #redInput . #yellowInput . #greenInput . #readyButton };
   tag: 'View';
   package: 'TrafficLightsProject'
```

Пам'ятаємо, що за створення вкладених демонстраторів відповідає метод *initializePresenters*, а за їхнє взаємне розташування – *defaultLayout*.

```
InputPanelPresenter >> initializePresenters
   redInput := self newNumberInput
      rangeMinimum: 1 maximum: 10; number: 5; yourself.
   yellowInput := self newNumberInput
      rangeMinimum: 1 maximum: 10; number: 2; yourself.
   greenInput := self newNumberInput
      rangeMinimum: 1 maximum: 10; number: 5; yourself.
   readyButton := self newButton label: 'Ready';
      icon: (self iconNamed: #smallOk); yourself

InputPanelPresenter >> defaultLayout
   ^ SpGridLayout new
      add: 'Set durations of ligths' at: 1 @ 1 span: 2 @ 1;
      add: 'red' at: 1 @ 2; add: redInput at: 2 @ 2;
      add: 'yellow' at: 1 @ 3; add: yellowInput at: 2 @ 3;
      add: 'green' at: 1 @ 4; add: greenInput at: 2 @ 4;
      add: readyButton at: 1 @ 5 span: 2 @ 1;
      yourself
```

Екземпляр класу *InputPanelPresenter* призначено для повторного використання у складі демонстратора цілого застосунку, тому потрібні методи, які дали б змогу налаштувати реакцію кнопки й отримати від демонстратора введені користувачем числа.

```
InputPanelPresenter >> action: aBlock
   readyButton action: aBlock

InputPanelPresenter >> redDuration
   ^ redInput number

InputPanelPresenter >> yellowDuration
   ^ yellowInput number

InputPanelPresenter >> greenDuration
   ^ greenInput number

InputPanelPresenter >> allDurations
   "returns three numbers - durations of all lamps"
   ^ { self redDuration . self yellowDuration . self greenDuration }
```

Перевірка роботи *InputPanelPresenter*

Щоб переконатися, що демонстратор *InputPanelPresenter* працює як треба, відкриємо його в окремому вікні. Для цього оголосимо ще два методи: *initializeWindow:* задасть початковий розмір і заголовок вікна, а *connectPresenters* налаштує реакцію кнопки і відобразить в консолі отримані від демонстратора дані.

```
InputPanelPresenter >> initializeWindow: aWindowPresenter
   aWindowPresenter
      title: 'Set durations of ligths';
      initialExtent: 270 @ 200

InputPanelPresenter >> connectPresenters
    self action: [
 Transcript show: '   Red - '; show: self redDuration printString; cr.
 Transcript show: 'Yellow - '; show: self yellowDuration printString; cr.
 Transcript show: ' Green - '; show: self greenDuration printString; cr.
 self hide ]
```

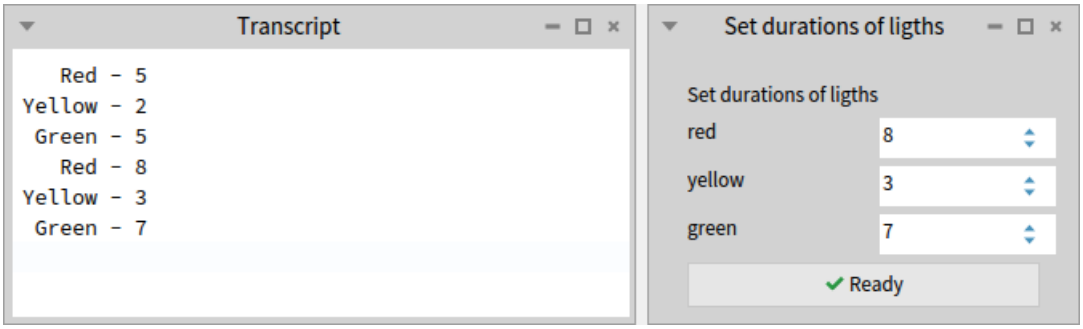


Рис. 19.4. Випробування *InputPanelPresenter*

Тепер можна відкрити вікно Transcript і виконати в Пісочниці фрагмент коду *InputPanelPresenter new open*. Мало б з'явитися вікно, як на рис. 19.4 праворуч. Натискання на кнопку ***Ready*** відобразить задані користувачем числові значення в консолі та приховає демонстратор, після чого порожнє вікно можна закрити, як звичайно. Ліворуч на рис. 19.4 зображено результати двох таких натискань.

19.5. Вигляд світлофора та застосунок

Демонстратор застосунку

Настав час об'єднати всі розроблені складові. Оголосимо для цього клас демонстратора застосунку. Він міститиме демонстратор морфи, три кнопки, екземпляр *InputPanelPresenter* і модель – екземпляр *TrafficLights*.

```
SpPresenter << #TrafficPresenter
   slots: { #view . #manualButton . #autoButton . #quitButton .
            #durationPanel . #model };
   tag: 'View';
   package: 'TrafficLightsProject'
```

Як завжди, вкладені демонстратори ініціалізує метод *initializePresenters*. Побудова графічного зображення світлофора трохи громіздка, тому її описано в окремому методі, *initializeMorph*. Отриману морфу використано і для побудови моделі, і для налаштуван-

ня *SpMorphPresenter*. Зверніть увагу на те, що екземпляри *SpMorphPresenter* і *InputPanel-Presenter* створюють загальним методом *instantiate:*. Панель введення тривалостей світіння спочатку прихована.

```
TrafficPresenter >> initializeMorph
   | body |
   body := BorderedMorph new color: Color lightGray; extent: 170 @ 440.
   body addAllMorphs: {
      (CircleMorph new borderColor: Color darkGray;
         extent: 120 @ 120; position: body position + (25 @ 25)).
      (CircleMorph new borderColor: Color darkGray;
         extent: 120 @ 120; position: body position + (25 @ 155)).
      (CircleMorph new borderColor: Color darkGray;
         extent: 120 @ 120; position: body position + (25 @ 285)) }.
   ^ body

TrafficPresenter >> initializePresenters
   | body |
   view := self instantiate: SpMorphPresenter.
   manualButton := self newButton.
   autoButton := self newButton.
   quitButton := self newButton.
   durationPanel := self instantiate: InputPanelPresenter.
   body := self initializeMorph.
   model := TrafficLights newWith: body.
   view morph: body.
   manualButton label: 'Manual switch';
      icon: (self iconNamed: #smallDoIt).
   autoButton label: 'Auto switch';
      icon: (self iconNamed: #tools).
   quitButton label: 'Quit';
      icon: (self iconNamed: #smallQuit).
   durationPanel hide
```

Тепер – макет! На рис. 19.1 видно, що компоненти керування розташовані праворуч від зображення світлофора. Таку розмітку забезпечить горизонтальний *SpBoxLayout*. Він же допоможе фіксувати ширину морфи. Щоб розташувати кнопки і панель у стовпець, використаємо вертикальний *SpBoxLayout*. І ще один вертикальний *SpBoxLayout*, щоб обмежити висоту морфи.

```
TrafficPresenter >> defaultLayout
   ^ SpBoxLayout newHorizontal spacing: 5;
      add: (SpBoxLayout newVertical
         add: view
         withConstraints: [ :c | c expand:false; height:440; padding:5 ];
         yourself )
      withConstraints: [ :constr | constr width: 180; padding: 5 ];
      add: (SpBoxLayout newVertical spacing: 5;
         add: manualButton expand: false;
         add: autoButton expand: false;
         add: durationPanel;
         addLast: quitButton expand: false;
         yourself);
      yourself
```

Розташувати кнопку ***Quit*** внизу вікна легко за допомогою повідомлення *addLast:*, або *addLast:expand:*.

Тепер, щоб все «ожило», налаштуємо поведінку кнопок і панелі введення.

```
TrafficPresenter >> connectPresenters
   manualButton action: [ model stop; changeLamp ].
   autoButton action: [ model stop. durationPanel show ].
   durationPanel action: [
      durationPanel hide.
      model setDurations: durationPanel allDurations; run ].
   quitButton action: [ self delete ]
```

Останній штрих – налаштування вікна застосунку.

```
initializeWindow: aWindowPresenter
   super initializeWindow: aWindowPresenter.
   aWindowPresenter
      title: 'Traffic Ligths Model by Pharo';
      initialExtent: 360 @ 475
```

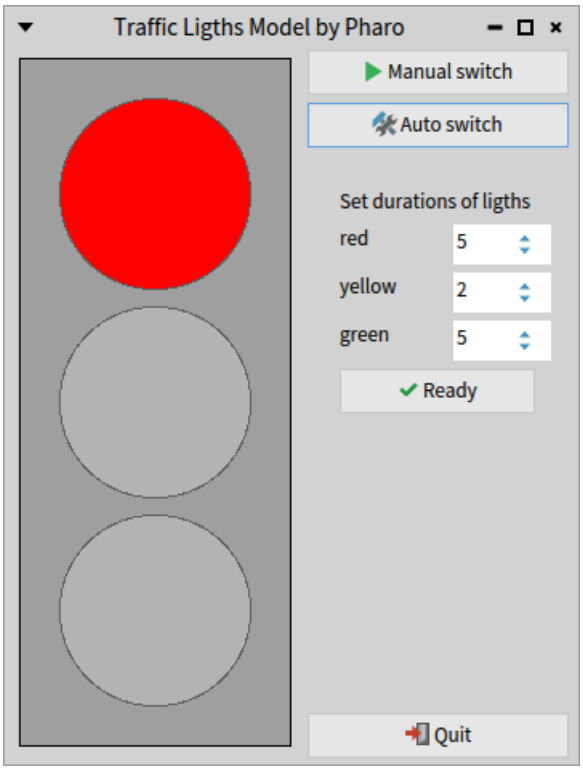


Рис. 19.5. Модель світлофора з відкритою панеллю налаштувань

Тепер достатньо виконати в Пісочниці код *TrafficPresenter new open*, і з'явиться вікно як на рис. 19.5. Легко переконатися, що всі компоненти працюють, як задумано.

Клас застосунку

Ми могли б полегшити використання *TrafficPresenter*, оголосивши для нього клас застосунку. То зробімо це! Так ми ще й зможемо зробити розміри вікна незмінними.

```
SpApplication << #TrafficApplication
   tag: 'View';   package: 'TrafficLightsProject'
```

```
TrafficApplication >> start
   | window |
   window := (self newPresenter: TrafficPresenter) open.
   window window beUnresizeable; removeExpandBox.
```

Щоб зробити запуск застосунку якнайзручнішим, оголосимо ще один метод класу.

```
TrafficApplication class >> example
   <script>
   self new run
```

Завдяки прагмі *script*, біля імені методу *example* в Системному оглядачі з'явиться активна піктограма . Клацніть на ній, і застосунок розпочне роботу.

19.6. Розширення можливостей застосунку

Мабуть кожен з нас спостерігав, що іноді світлофори працюють у блимаючому режимі. Найчастіше таке трапляється уночі: з постійною частотою вмикається і вимикається жовтий ліхтар, а інші не працюють. Чи змогли б ми доповнити розроблений застосунок можливістю перемикання в нічний режим роботи? І зробити це так, щоб не зруйнувати і не ускладнити понад міру написаний код. Спробуймо!

Алгоритм перемикання

Перше, що спадає на гадку, це доповнити клас *TrafficLights* змінною *mode* і перевіряти її значення в методі *TrafficLights >> changeLamp*. Але тоді в одному методі буде описано два різні алгоритми, і перевіряти *mode* доведеться перед кожним перемиканням. До того ж така архітектура коду не сприятиме майбутнім змінам: якщо виникне потреба описати ще один режим роботи, то легко може виникнути плутанина.

Підемо іншим шляхом. Кожен алгоритм перемикання треба описати окремим методом. Селектор актуального методу можна зберігати в змінній екземпляра, наприклад, у змінній *method*. Тоді все, що потрібно зробити в *TrafficLights >> changeLamp* – це виконати *self perform: method*, а зміна режиму роботи полягатиме в зміні значення *method*.

Вирішено. Доповнимо оголошення класу *TrafficLights*.

```
Object << #TrafficLights
   slots: { #lamps . #activeLamp . #view . #mustRun . #stopwatch .
            #method };
   tag: 'Model';
   package: 'TrafficLightsProject'
```

Змінимо також *changeLamp*: його вміст перенесемо в інший метод, а його зробимо загальним і надалі незмінним.

```
TrafficLights >> nextLamp
   (lamps at: activeLamp) switchOff.
   activeLamp := activeLamp % 4 + 1.
   (lamps at: activeLamp) switchOn

TrafficLights >> blinkLamp
   | lamp |
   lamp := lamps at: activeLamp.
```

```
        lamp isSwitchedOn
           ifTrue: [ lamp switchOff ]
           ifFalse: [ lamp switchOn ]

     TrafficLights >> changeLamp
        self perform: method
```

Щоб задати початковий режим роботи світлофора, доведеться доповнити *TrafficLights >> initialize* одним рядочком: *method := #nextLamp*. А для перемикання режимів оголосимо нові методи.

```
     TrafficLights >> switchToDayMode
        method := #nextLamp

     TrafficLights >> switchToNightMode
        (lamps at: activeLamp) switchOff.
        activeLamp := 2.
        method := #blinkLamp
```

Саме час перевірити, чи внесені зміни не зіпсували модель. На щастя, ми оголосили класи тестів. Запустимо їх і переконаємося, що все працює, як і раніше. Звичайно, варто було б доповнити *TrafficLightsTest* тестом нічного режиму роботи.

Доповнення інтерфейсу користувача

Модель готова до перемикань. Тепер треба додати до вікна застосунку відповідний засіб. Ми могли б використати групу залежних перемикачів *SpRadioButtonPresenter*, або один незалежний *SpCheckBoxPresenter*. Як на два режими, то група перемикачів – занадто складний вибір, а один незалежний перемикач – не дуже наочно. У Spec є ще один різновид кнопки – *SpToggleButtonPresenter*, його і використаємо. Така кнопка перебуває в одному з двох станів: натиснута (активована), або вільна (деактивована).

Доповнимо оголошення класу *TrafficPresenter* змінною для зберігання перемикача.

```
     SpPresenter << #TrafficPresenter
        slots: { #view . #manualButton . #autoButton . #quitButton .
                 #durationPanel . #model . #switchButton };
        tag: 'View';
        package: 'TrafficLightsProject'
```

Новий візуальний компонент хотілося б оздобити належними піктограмами. Нагадаємо, що переглянути перелік доступних піктограм можна в Інспекторі, якщо виконати такий код: *Smalltalk ui icons inspect*.

Доповнити доведеться оголошені раніше методи. Заради економії місця покажемо тут тільки доповнення.

```
     TrafficPresenter >> initializePresenters
        . . . . .
        switchButton := self newToggleButton.
        switchButton
           label: 'Switch to night mode';
           icon: (self iconNamed: #glamorousCloud)

     TrafficPresenter >> defaultLayout
        . . . . .
```

```
            add: (SpBoxLayout newVertical spacing: 5;
               . . . . .
               addLast: switchButton expand: false;
               addLast: quitButton expand: false;
         . . . . .

      TrafficPresenter >> connectPresenters
         . . . . .
         switchButton
            whenActivatedDo: [
               switchButton
                  label: 'Switch to day mode';
                  icon: (Object iconNamed: #smallNew).
               model switchToNightMode ];
            whenDeactivatedDo: [
               switchButton
                  label: 'Switch to night mode';
                  icon: (Object iconNamed: #glamorousCloud).
               model switchToDayMode ]
```

Наявність двох різних методів конфігурування для різних станів кнопки полегшує програмування. Тепер перемикання кнопки впливатиме не тільки на модель, а й на вигляд самої кнопки.

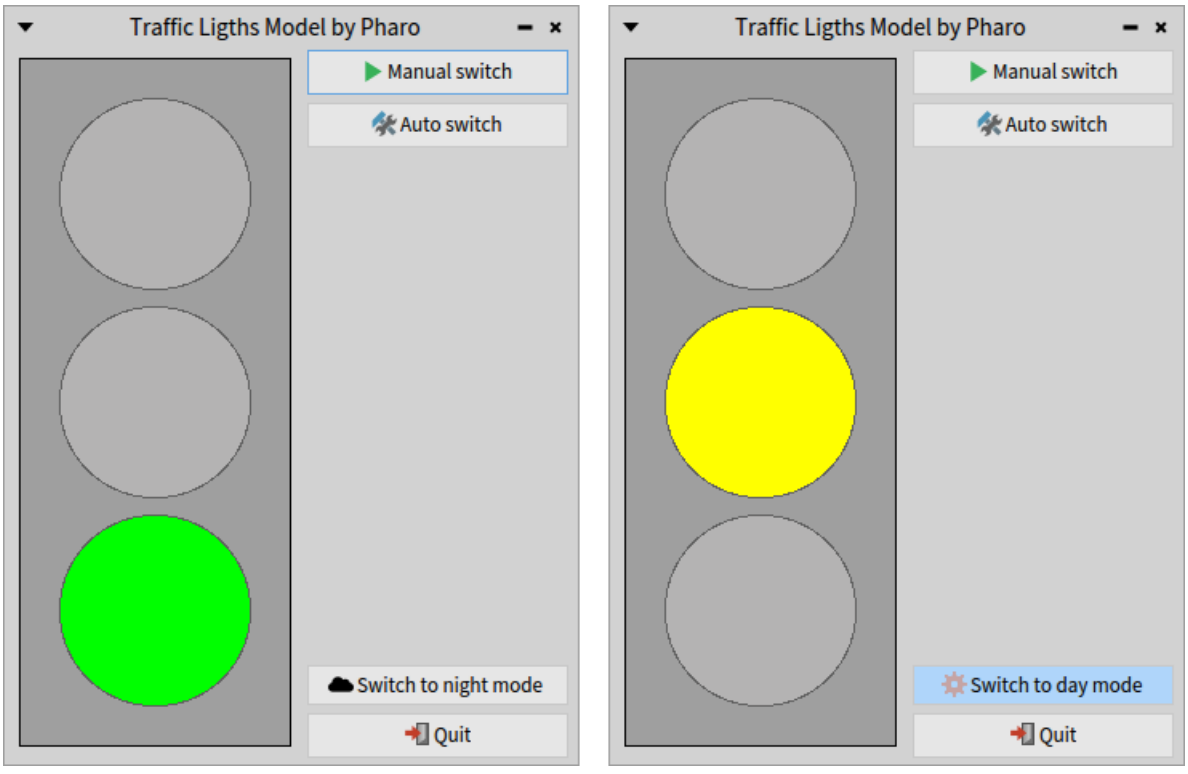


Рис. 19.6. Денний і нічний режими роботи світлофора

Відшукайте в Оглядачі класів *TrafficApplication* і запустіть на виконання метод класу *example*. Мало б з'явитися вікно застосунку, як на рис. 19.6 ліворуч, доповнене перемикачем. Клацання на ньому переводить світлофор у інший режим роботи (див. рис. 19.6 праворуч). Випробуйте, як тепер працює доповнений застосунок.

Текст «Про програму …»

Розробку завершено. Залишилося хіба що прикрасити її коротким текстом «Про програму …». Для цього доведеться доповнити метод ініціалізації вікна.

```
initializeWindow: aWindowPresenter
   super initializeWindow: aWindowPresenter.
   aWindowPresenter
      title: 'Traffic Ligths Model by Pharo';
      initialExtent: 360 @ 475;
      aboutText: 'Model of Traffic Lights built with Spec.\\Can run under
   manual or automatic control.\Day and night modes are available.\\ (C)
   Serhiy Yaroshko, IFNUL, 2025' withCRs
```

19.7. Підсумки до розділу

У розділі описано, як інтегрувати *SpMorphPresenter* у застосунок Spec. Щоб задати його розмір, довелося помістити його у два вкладені *SpBoxLayout*. Створений застосунок використовує деякі не описані раніше у цій книзі засоби і можливості. Наприклад, додавати демонстратор до макета можна повідомленням *addLast:*, тимчасово приховати і показати знову повідомленнями *hide* та *show*, відповідно. Spec заслуговує на глибоке вивчення, бо містить багато корисних класів, наприклад, цей розділ відкрив читачеві *SpToggleButtonPresenter*.

У розробленому застосунку реалізовано два способи керування: і діями користувача, і плином часу. Для стеження за перебігом часу в окремому потоці виконання використано екземпляр *Stopwatch*.

Можливо, читач вирішить, що доречно було б ліпше відокремити модель даних світлофора від його вигляду і наслідувати демонстратор від *SpPresenterWithModel*, як описано в розділі 6. Залишимо цю можливість як вправу для читача.

Розділ 20

Група залежних перемикачів – демонстратор для повторного використання

Доволі часто в інтерфейсі користувача застосунку використовують залежні перемикачі. Відповідний візуальний компонент у Spec представляє *SpRadioButtonPresenter*. У нього є певна особливість: перемикач ніколи не використовують окремо від інших екземплярів *SpRadioButtonPresenter*, вони завжди працюють у групі. Методи класу *SpRadioButtonExample* добре пояснюють, як це зробити. Спочатку до демонстратора застосунку за допомогою *self newRadioButton* додають перемикачі й першому з них повідомляють, що всі інші становлять колекцію його залежних перемикачів. Без такої вказівки група не працюватиме злагоджено. Потім задають видимі назви і налаштовують реакції перемикачів на зміну стану. Зазвичай всі перемикачі реагують однаково, бо ж працюють у групі.

Навіть цей стислий опис кроків, які потрібно виконати, щоб створити групу кнопок-перемикачів наштовхує на думку, що добре було б автоматизувати процес створення. У цьому розділі описано побудову класу *SpRadioGroupPresenter* – демонстратора, якому достатньо дати колекцію об'єктів, щоб отримати готову групу відповідних *SpRadioButtonPresenter*'ів (подібно до того, як *SpListPresenter* вибудовує список).

Пропонований *SpRadioGroupPresenter* призначено для використання в інтерфейсі інших застосунків. Таке використання буде проілюстровано невеликою програмою з міні-опитуванням.

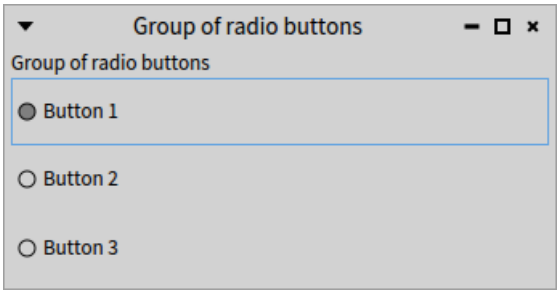


Рис. 20.1. Зразок групи залежних перемикачів

20.1. Формулювання завдання

Мета – оголосити клас, який автоматизує створення групи залежних перемикачів. Назви перемикачів задають колекцією рядків або інших об'єктів, для яких визначено спосіб отримання рядка-напису. Група повинна містити відповідні піддемонстратори *SpRadioButtonPresenter*, задавати колекцію залежних перемикачів першого з них, розташовувати піддемонстратори в одному або кількох стовпцях. Зміна стану одного з перемикачів групи повинна спричиняти подію групи. Безпосередньо над перемикачами група може відображати напис, загальну назву, як зображено на рис. 20.1.

Клас групи перемикачів мав би містити методи для налаштування колекції об'єктів групи, способу відображення об'єктів у назви перемикачів, загальної назви, кількості стовпців і реакції на події. Додатково група могла б повідомляти номер вибраного перемикача та відповідний йому об'єкт колекції.

20.2. Оголошення класу *SpRadioGroupPresenter*

До складу демонстратора групи входитиме чимало об'єктів, тому саме час задуматися про перелік змінних екземпляра. Колекцію об'єктів, джерело даних для перемикачів, назвемо *items*, а колекцію кнопок-перемикачів – *buttons*. Загальну назву, напис над групою помістимо в *title*, а кількість стовпців – у *columnCount*. Спосіб перетворення об'єктів на назви зазвичай задають блоком, зберігатимемо його в змінній *display*. Для відстеження стану групи стане в пригоді змінна *index*, яка міститиме номер увімкненого перемикача, а змінна *whenIndexChangedDo* зберігатиме блок, який задає реакцію на зміну індексу. Отримаємо таке оголошення.

```
SpPresenter << #SpRadioGroupPresenter
   slots: { #title . #items . #buttons . #display . #columnCount .
            #index . #whenIndexChangedDo };
   tag: 'Presenter';
   package: 'RadioGroupProject'
```

20.3. Створення вкладених демонстраторів

Демонстратори створюють у методі *initializePresenters*. Для створення перемикачів потрібно використати об'єкти колекції *items*, тому її значення потрібно задати *перед* викликом цього методу. Зробити це можна двома способами. У методі *SpRadioGroup-Presenter>>initiflize* колекцію можна ініціалізувати деяким усталеним значенням, наприклад таким, що його повертає відповідний метод класу. Тоді екземпляр демонстратора можна буде створити методом *instatiate:* і налаштувати *items* згодом. Але є ліпший спосіб. Як описано у параграфі 6.2, щоб задати модель даних демонстратора ще перед ініціалізацією, оголошують метод *setModelBeforeInitialization:*, а демонстратор створюють методом *instatiate:on:*.

Запрограмуємо обидва способи. Разом з *items* ініціалізуємо й інші змінні.

```
SpRadioGroupPresenter >> setModelBeforeInitialization: aCollection
   items := aCollection asOrderedCollection

SpRadioGroupPresenter class >> defaultItems
   ^ #( 'Button 1' 'Button 2' 'Button 3' )

SpRadioGroupPresenter >> initialize
   index := 1.
   items ifNil: [ items := self class defaultItems ].
   display := [ :object | object asStringOrText ].
   columnCount := 1.
   super initialize
```

Тут важливо, щоб повідомлення *super initialize* було надіслане *після* того, як задано всі інші значення, адже саме воно спричиняє виклик *initializePresenters*.

```
SpRadioGroupPresenter >> initializePresenters
   title := self newLabel.
   title label: self class title.
   self updateButtons

SpRadioGroupPresenter class >> title
   ^ 'Group of radio buttons'
```

Створення перемикачів описано окремим методом. Він стане нам в нагоді ще не раз.

```
SpRadioGroupPresenter >> updateButtons
   | firstButton |
   firstButton := self newRadioButton.
   buttons := OrderedCollection with: firstButton.
   items allButFirstDo: [ :object | buttons add: self newRadioButton ].
   firstButton associatedRadioButtons: buttons allButFirst.
   self updateLabels

SpRadioGroupPresenter >> updateLabels
   buttons withIndexDo: [ :btn :ind |
      btn label: (display value: (items at: ind)) ]
```

20.4. Налаштування поведінки

Як і годиться, поведінку кнопок-перемикачів налаштуємо в методі *connectPresenters*.

```
SpRadioGroupPresenter >> connectPresenters
   | block |
   block := [ self buttonStateChanged ].
   buttons do: [ :btn | btn whenChangedDo: block ]
```

Тут усі кнопки реагують однаково – викликають *SpRadioGroupPresenter >> buttonStateChanged*. Обговоримо докладніше, що мав би робити такий метод. Для зручного використання групи він мав би з'ясовувати номер увімкнутого перемикача і спричиняти подію групи. Виявилося, що при кожному перемиканні *buttonStateChanged* спрацьовує двічі: спочатку, коли вимикається один перемикач, а потім, коли вмикається інший. Щоб не запускати подію групи двічі, довелося перевіряти, який раз виконується *buttonStateChanged*, і реагувати тільки на друге спрацювання. Для цього до оголошення класу додали змінну *isEvenEvent*. Її початкове значення – *false*.

```
SpRadioGroupPresenter >> buttonStateChanged
   buttons withIndexDo: [ :btn :ind |
      btn state ifTrue: [ index := ind ] ].
   isEvenEvent ifTrue: [
      whenIndexChangedDo ifNotNil: [ whenIndexChangedDo value: index ] ].
   isEvenEvent := isEvenEvent not
```

20.5. Макет групи

Почнемо з простішого випадку, коли всі кнопки розташовано в один стовпчик, але опишемо такий макет окремим методом, щоб бути готовими вносити зміни і розташовувати їх у кілька стовпців.

```
SpRadioGroupPresenter >> defaultLayout
   ^ SpBoxLayout newTopToBottom
        add: title expand: false;
        add: self singleColumnLayout;
        yourself

SpRadioGroupPresenter >> singleColumnLayout
   | boxLayout |
   boxLayout := SpBoxLayout newTopToBottom.
```

```
        buttons do: [ :btn | boxLayout add: btn ].
        ^ boxLayout
```

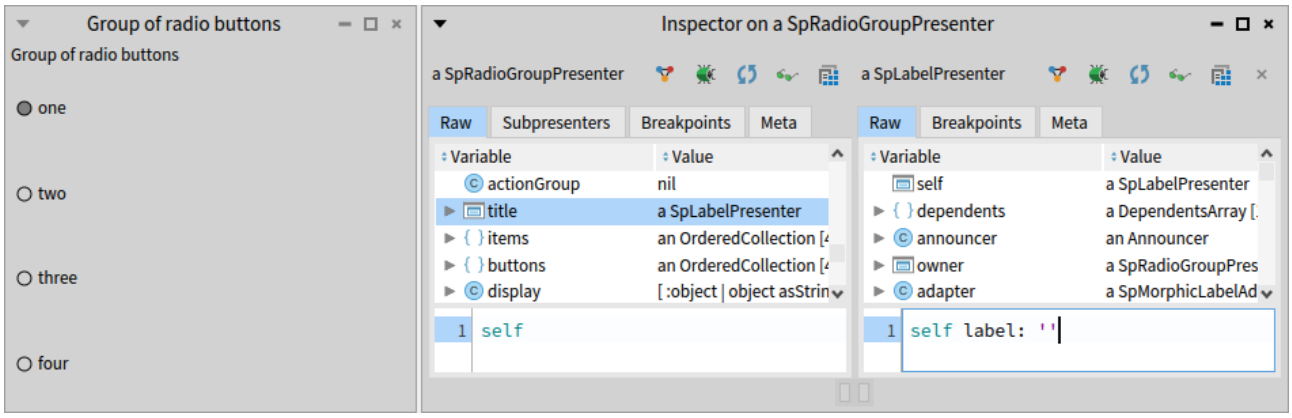


Рис. 20.2. Інспектування екземпляра *SpRadioGroupPresenter* із заданим вмістом

Перше випробування

Ми оголосили всі обов'язкові методи класу демонстратора і можемо тепер подивитися, що вийшло. Легко переконатися, що код *SpRadioGroupPresenter new open* відкриває вікно, як на рис. 20.1. Якщо ж відкрити Пісочницю і виконати в ній такий фрагмент,

```
group := SpRadioGroupPresenter on: #('one' 'two' 'three' 'four').
group open; inspect.
```

то відкриються вікна, як на рис. 20.2. У Інспекторі можна відстежувати, чи правильно змінюється значення *index* після перемикання, які значення мають інші змінні демонстратора. Одразу стає зрозуміло, що класові бракує методів доступу. Немає змоги інакше, як через Інспектор, змінити, наприклад, напис над групою. Якщо ж виконати в Інспекторі *self label: ''* для *title*, то виявиться ще один недолік: напис зникне з вікна, але все ще займатиме місце, і над кнопками виникне помітна прогалина.

20.6. Доповнення інтерфейсу *SpRadioGroupPresenter*

Оголошення методів-селекторів не спричинить ніяких труднощів. З методами-модифікаторами доведеться попрацювати більше, адже вони змінюватимуть зовнішній вигляд демонстратора.

```
SpRadioGroupPresenter >> columnCount
    ^ columnCount

SpRadioGroupPresenter >> indexSelected
    ^ index

SpRadioGroupPresenter >> itemSelected
    ^ items at: index

SpRadioGroupPresenter >> title
    ^ title label

SpRadioGroupPresenter >> whenIndexChangedDo: aBlock
    whenIndexChangedDo := aBlock
```

```
SpRadioGroupPresenter >> display: aBlock
   display := aBlock.
   self updateLabels
```

Найбільше клопоту спричиняє проблема порожнього напису. Щоб не виникала прогалина доведеться перевіряти, чи напис містить порожній рядок, і не додавати *title* до макета в цьому випадку. Створення макета дещо ускладниться.

```
SpRadioGroupPresenter >> title: aString
   title label: aString.
   self updateCurrentLayout

SpRadioGroupPresenter >> updateCurrentLayout
   self layout: (self conditionalLayout: [ self singleColumnLayout ])

SpRadioGroupPresenter >> conditionalLayout: aBlock
   self title = String empty
      ifTrue: [ ^ aBlock value ]
      ifFalse: [
         ^ SpBoxLayout newTopToBottom
              add: title expand: false;
              add: aBlock value;
              yourself ]
```

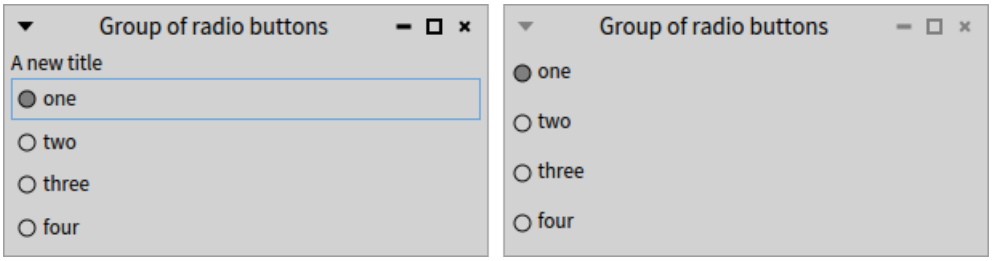


Рис. 20.3. Результати налаштування напису над групою перемикачів

Тепер можна повернутися до Пісочниці, повторно створити *group* і переконатися, що *group title: 'A new title'* змінює напис над групою перемикачів, а *group title: ''* спричиняє перебудову макета, і напис зникає повністю. Ви мали б отримати демонстратори, як на рис. 20.3.

Об'єкт *String empty* відіграє особливу роль у керуванні написом. Щоб виокремити цю обставину і полегшити налаштування групи, оголосимо такий метод.

```
SpRadioGroupPresenter >> hideTitle
   self title: String empty
```

20.7. Динамічна зміна макета та інші удосконалення

Розташування перемикачів у кількох стовпцях

Перше, що спадає на гадку, аби розташувати перемикачі у кількох стовпцях, тобто у вигляді таблиці, то це використати *SpGridLayout*. На жаль, цей макет не розтягається по висоті до розмірів вікна, а займає стільки місця, скільки мінімально потрібно вкладеним демонстраторам. Тому імітуватимемо таблицю за допомогою кількох вертикальних *SpBoxLayout*, вкладених у горизонтальний *SpBoxLayout*. Для того, щоб така таблиця

мала горизонтальні рядки, кожен стовпець повинен містити однакову кількість вкладених демонстраторів. Може так трапитися, що кількість перемикачів у групі не ділитиметься на задану кількість стовпців. Тоді останньому стовпцю бракуватиме компонентів. Щоб вирівняти рядки, доповнимо його порожніми *SpLabelPresenter*. Так отримаємо метод побудови макета з *columnCount* стовпцями.

```
SpRadioGroupPresenter >> multipleColumnLayout
   | size hBoxLayout vBoxLayout high begin end |
   hBoxLayout := SpBoxLayout newLeftToRight.
   size := buttons size.
   high := size + columnCount - 1 // columnCount.
   begin := 1 - high.
   end := 0.
   columnCount timesRepeat: [
      vBoxLayout := SpBoxLayout newTopToBottom.
      begin := begin + high.
      end := end + high.
      begin to: (end min: size) do: [ :i |
           vBoxLayout add: (buttons at: i) ].
      hBoxLayout add: vBoxLayout ].
   end - size timesRepeat: [ vBoxLayout add: SpLabelPresenter new ].
   ^ hBoxLayout
```

Значення *columnCount* не мало б бути довільним. Обмежимо його значеннями від 1 до 4. Зробимо це так, щоб кількість стовпців можна було змінити лише такими методами.

```
SpRadioGroupPresenter >> beSingleColumn
   columnCount = 1 ifTrue: [ ^ self ].
   columnCount := 1.
   self layout: (self conditionalLayout: [ self singleColumnLayout ])

SpRadioGroupPresenter >> beDoubleColumn
   columnCount = 2 ifTrue: [ ^ self ].
   columnCount := 2.
   self layout: (self conditionalLayout: [ self multipleColumnLayout ])
```

Ще два методи *beTripleColumn* і *beQuatroColumn* відрізняються від *beDoubleColumn* тільки тим, що замість 2 у них всюди написано 3 і 4, відповідно.

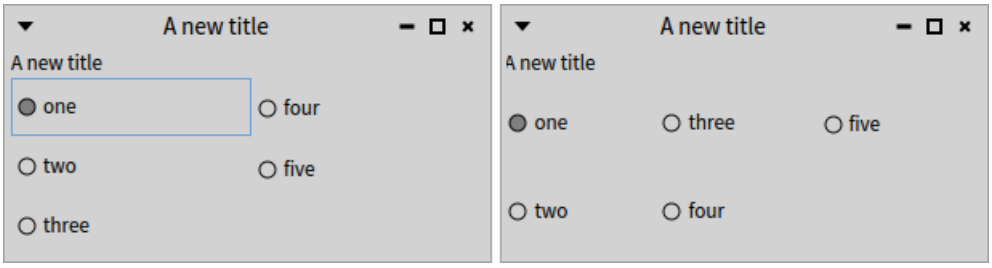


Рис. 20.4. Зміна кількості стовпців

Щоб врахувати нові можливості побудови макета, доведеться дещо змінити метод *updateCurrentLayout*.

```
SpRadioGroupPresenter >> updateCurrentLayout
   columnCount = 1
      ifTrue: [ self layout:
```

```
            (self conditionalLayout: [ self singleColumnLayout ]) ]
         ifFalse: [ self layout:
            (self conditionalLayout: [ self multipleColumnLayout ]) ]
```

Щоб випробувати нові методи в дії, виконаємо в Пісочниці такий фрагмент.

```
group := SpRadioGroupPresenter on: #('one' 'two' 'three' 'four' 'five').
group title: 'A new title';
   beDoubleColumn; open.
```

Отримаємо вікно, як на рис. 20.4 ліворуч. А після виконання *group beTripleColumn* демонстратор набуде вигляду, як на рис. 20.4 праворуч.

Зміна колекції об'єктів

Чим ширший функціонал компонента, тим більші його шанси на успіх у користувачів. Доповнимо *SpRadioGroupPresenter* можливістю змінювати його колекцію об'єктів. Для цього оголосимо три методи: *items: newCollection* замінятиме колекцію повністю, а *addItem: anObject* і *removeItem: anObject* змінюватимуть наявну.

Найпростіше виконати заміну в тому випадку, коли розмір нової колекції збігається з розміром наявної. Тоді достатньо оновити назви перемикачів. У протилежному ж випадку доведеться повністю перебудувати компонент і відповідно налаштувати значення *index*, адже старий вибір для нової колекції стане не актуальним.

```
SpRadioGroupPresenter >> items: aCollection
   | size |
   size := items size.
   items := aCollection asOrderedCollection.
   size = aCollection size
      ifTrue: [ self updateLabels ]
      ifFalse: [ self updateButtons; connectPresenters;
                  updateCurrentLayout; correctIndexToFirst ]

SpRadioGroupPresenter >> correctIndexToFirst
   index := 1.
   whenIndexChangedDo ifNotNil: [ whenIndexChangedDo value: index ]
```

Додавання нового елемента до колекції додасть перемикач до групи. Для цього доведеться перебудувати демонстратор, але попередній вибір мав би зберегтися. У новозбудованій групі увімкнено перший перемикач, тому синхронізуємо стан перемикачів з поточним значенням *index*. Зробимо це ще перед викликом *connectPresenters*, щоб не ініціювати подію групи, адже зміни вибраного перемикача не відбулося.

```
SpRadioGroupPresenter >> addItem: anObject
   items add: anObject.
   self updateButtons.
   (buttons at: index) state: true.
   self connectPresenters; updateCurrentLayout
```

Вилучити елемент з колекції дещо складніше, бо поведінка залежить від того, який саме елемент вилучають: чи його перемикач активовано, чи ні. Якщо ні, то стан перемикачів мав би залишитися без змін.

```
SpRadioGroupPresenter >> removeItem: anObject
```

```
    | keepIndex |
    keepIndex := anObject ~= self itemSelected.
    items remove: anObject.
    self updateButtons.
    keepIndex ifTrue: [ (buttons at: index) state: true.
          self connectPresenters; updateCurrentLayout ]
       ifFalse: [ self connectPresenters;
          updateCurrentLayout; correctIndexToFirst ]
```

20.8. Приклад використання *SpRadioGroupPresenter*

Продемонструємо використання *SpRadioGroupPresenter* на прикладі побудови невеликого застосунку, який задає користувачеві три запитання з можливими варіантами відповідей. Застосунок протоколюватиме кожен вибір, зроблений користувачем, і фіксуватиме його остаточну відповідь.

Для відображення запитання з відповідями використаємо екземпляр *SpRadioGroupPresenter*. Його *title* міститиме текст запитання, а *items* – варіанти відповідей. Підписка на подію *whenIndexChangedDo* дасть змогу записувати кожне клацання користувача по перемикачах групи. Для створення груп використаємо колекції об'єктів різних типів і налаштуємо групи на різну кількість стовпців.

Роботою застосунку керуватимемо за допомогою кількох кнопок, а протокол опитування й остаточну відповідь відображатимемо в текстовому вікні, налаштованому тільки для читання.

```
SpPresenter << #RGHostApplication
   slots: { #question1 . #question2 . #question3 . #buttonCheck .
            #buttonClear . #buttonQuit . #output };
   tag: 'Application';
   package: 'RadioGroupProject'

RGHostApplication >> initializePresenters
   question1 := self instantiate: SpRadioGroupPresenter
                    on: #( 10 14 16 18 20 22 ).
   question2 := self instantiate: SpRadioGroupPresenter
                    on: #( 'Most of Spec in one example'
         'A 10 min small example' 'A Mail application'
         'A working example for managing windows' ).
   question3 := self instantiate: SpRadioGroupPresenter
                    on: { #useful. 'very interesting'.
                         'too complicated'. #MustHaveAndRead }.
   buttonCheck := self newButton.
   buttonClear := self newButton.
   buttonQuit := self newButton.
   output := self newText.
   question1 title: 'How many chapters are in the "Application Building
 with Spec 2.0" book?';
      beTripleColumn.
   question2 title: 'Check the first working example in the book.'.
   question3 title: 'What do you think about the book? It is ...';
      beDoubleColumn.
   buttonCheck label: 'Check'.
   buttonClear label: 'Clear'.
```

```
        buttonQuit label: 'Quit'.
        output beNotEditable

    RGHostApplication >> defaultLayout
        ^ SpBoxLayout newTopToBottom spacing: 20;
            add: question1;
            add: question2;
            add: question3;
            add: (SpBoxLayout newLeftToRight spacing: 5;
                   add: (SpBoxLayout newTopToBottom spacing: 5;
                          add: buttonCheck; add: buttonClear;
                          add: buttonQuit; yourself)
                   withConstraints: [ :constr | constr width: 100 ];
                   add: output);
            yourself

    RGHostApplication >> connectPresenters
        question1 whenIndexChangedDo: [ :index |
           output text: (self buildLineWith: 'first - ' and: index) ].
        question2 whenIndexChangedDo: [ :index |
           output text: (self buildLineWith: 'second - ' and: index) ].
        question3 whenIndexChangedDo: [ :index |
           output text: (self buildLineWith: 'third - ' and: index) ].
        buttonCheck action: [ output text: (String streamContents: [ :stream |
               stream nextPutAll: output text;
                      nextPutAll: '-----------------------'; cr;
                      nextPutAll: (self buidAnswerFor: question1);
                      nextPutAll: (self buidAnswerFor: question2);
                      nextPutAll: (self buidAnswerFor: question3) ] ) ].
        buttonClear action: [ output clearContent ].
        buttonQuit action: [ self delete ]

    RGHostApplication >> buildLineWith: name and: number
        ^ String streamContents: [ :stream | stream nextPutAll: output text;
            nextPutAll: name; print: number; cr ]

    RGHostApplication >> buidAnswerFor: aRadioGroup
        ^ String streamContents: [ :stream | stream
            nextPutAll: aRadioGroup title; nextPutAll: ': '.
            aRadioGroup itemSelected printOn: stream. stream cr ]
```

Це все. Тепер переваги використання *SpRadioGroupPresenter* очевидні: без нього довелося б створювати втричі більше візуальних компонент і втричі більше методів для налаштування їхньої поведінки.

Завершальний крок – оголошення методу, що налаштовує вигляд вікна застосунку.

```
    RGHostApplication >> initializeWindow: aWindowPresenter
        super initializeWindow: aWindowPresenter.
        aWindowPresenter
           title: 'Three questions only';
           initialExtent: 560 @ 445
```

Тепер можна запускати застосунок за допомогою *RGHostApplication new open* і випробовувати його в дії. Ви мали б отримати вікно, як на рис. 20.5.

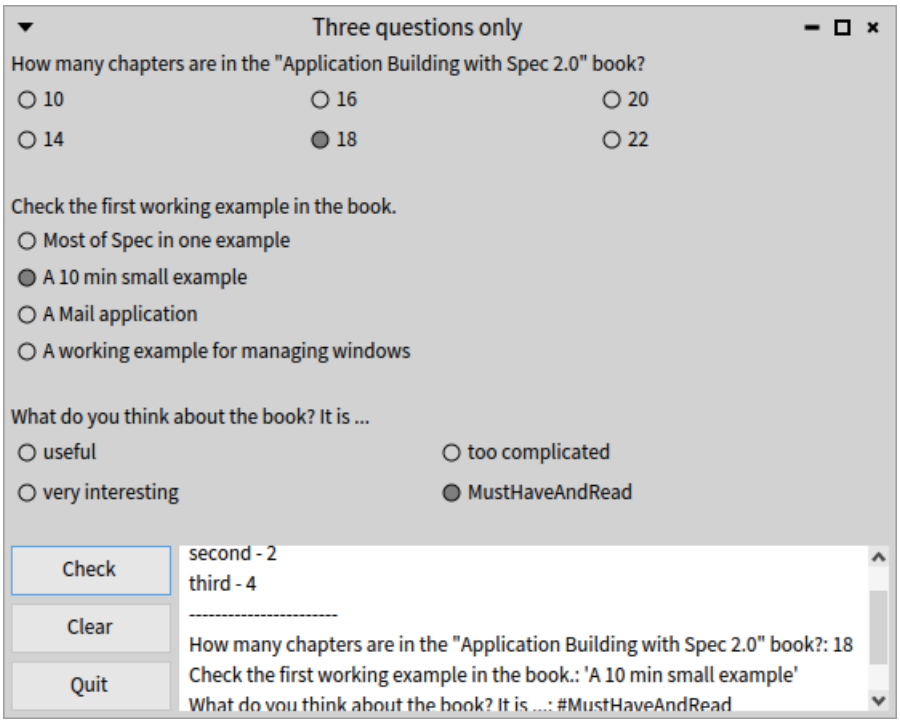


Рис. 20.5. Протокол і результати опитування

20.9. Підсумки до розділу

Усі класи демонстраторів, підкласи *SpPresenter* можна використовувати двояко: і як самостійні графічні інтерфейси користувача, і як візуальні компоненти в складі інших демонстраторів. Щоб успішно виконувати роль компонента, клас повинен володіти розвиненими засобами для налаштування свого складу, вигляду та для організації взаємодії. У *SpRadioGroupPresenter* таких достатньо. Користувач може задати і згодом змінити склад колекції об'єктів і, як наслідок, перемикачів, задати заголовок, керувати зовнішнім виглядом групи, довідатися номер вибраного перемикача і сам вибраний об'єкт, налаштувати реакцію на зміну вибраного перемикача.

У розробці *SpRadioGroupPresenter* довелося приділити достатньо уваги послідовності ініціалізації змінних екземпляра та послідовності налаштувань під час перебудови. Тут суттєво використано динамічну природу демонстраторів і макетів Spec, яка дає змогу змінювати зовнішній вигляд і склад демонстратора «на льоту».

Використання демонстраторів як готових будівельних блоків суттєво спрощує розробку розвинених графічних інтерфейсів застосунків.

На завершення посібника зазначимо, що це далеко не завершення вивчення Spec. Багато нових його можливостей можна довідатися з реалізації класів демонстраторів Pharo. Читачі могли б також розширити описані в посібнику застосунки. Зокрема, можна було б удосконалити *SpRadioGroupPresenter*:

- розглянути можливість використання *nil* замість *String empty* для позначення відсутнього заголовка групи;
- застосувати відповідні стилі, щоб зобразити рамку навколо групи перемикачів, або
- використати *SpFrameLayout*, щоб зобразити рамку і поєднати її з заголовком.

Електронне навчальне видання

К. Де Хондт, С. Дюкас разом з
С. Джордан-Монтаньо та Е. Лоренцано

Побудова застосунків засобами Spec 2.0

Переклад українською з доповненнями
Сергія ЯРОШКА

Редактор *Н. Плиса*
Комп'ютерне верстання *С. Ярошко*

Формат 60×84/8. Ум. друк. арк. 27,7. Зам. 21Е.

Видавець та виготовлювач:
Львівський національний університет імені Івана Франка,
вул. Університетська, 1, Львів, 79000
Свідоцтво про внесення суб'єкта
видавничої справи до Державного реєстру видавців, виготівників
і розповсюджувачів видавничої продукції.
Серія ДК № 3059 від 13.12.2007 р.