КОМП'ЮТЕРНІ НАУКИ

УДК 004.432.2, 004.422.83 *http://dx.doi.org/10.30970/vam.2025.34.13592*

УДОСКОНАЛЕННЯ БІБЛІОТЕКИ ВІЗУАЛЬНИХ КОМПОНЕНТІВ SPEC 2.0 ДЛЯ PHARO

О. Байдала, С. Ярошко

Львівський національний університет імені Івана Франка,
вул. Університетська 1, Львів, 79000, Україна,
e-mail: serhiy.yaroshko@lnu.edu.ua

Описано новий спосіб налаштування властивостей візуальних компонентів бібліотеки Spec у середовищі Pharo. Збудовано новий базовий клас демонстратора *BasePresenter*, який надає декларативний інтерфейс для налаштування кольору, шрифту, рамок, відступів тощо й автоматично генерує та застосовує відповідний стиль оформлення. Новий демонстратор використано для доповнення бібліотеки Spec новими візуальними компонентами: простими та структурованими. Оголошено клас декоратора *PresenterDecorator* для огортання будь-якого стандартного демонстратора Spec і збагачення його новими можливостями стилізації. Оголошено кілька конкретних підкласів: *ButtonPresenter*, *LabelPresenter*, *ListPresenter*, *CheckBoxPresenter*, *RadioButtonPresenter*. Призначення кожного з них легко зрозуміти за назвою.

Також розроблено складніші компоненти. *TextInputPresenter* надає можливість автоматичної перевірки введеного тексту. Одне або кілька значень можна вибрати за допомогою *ComboBoxPresenter*, *MultiSelectComboBoxPresenter*, *RadioButtonGroupPresenter* або *CheckBoxGroupPresenter*. *TimePickerPresenter* підтримує інтерактивний вибір часу. Компонент *DynamicFormBuilder*, керований послідовністю повідомлень *textField:*, *dropList:*, *radioGroup:* тощо може створити будь-яку форму введення. Компоненти *SearchableTablePresenter* та *PaginatedTablePresenter* надають демонстраторам таблиць поле фільтра та розбиття на сторінки, відповідно. *WizardPresenter* може організувати та візуалізувати покроковий сценарій взаємодії з користувачем.

Продемонстровано, що запропонований підхід пришвидшує створення графічного інтерфейсу застосунку.

Ключові слова: Pharo, Spec, графічний інтерфейс користувача, візуальний компонент, стилізація, віконний застосунок, клас, об'єкт.

1. Вступ

Одна з головних вимог до сучасного програмного забезпечення – наявність якісного та зручного графічного інтерфейсу, орієнтованого на користувача. Саме інтерфейс формує перше враження про програмний продукт і може відіграти вирішальну роль у тому, чи користувач продовжить роботу з програмою. Тому проєктування та побудова графічних інтерфейсів (GUI) є надзвичайно важливим етапом розробки застосунку. Багато мов програмування пропонують власні, часто кросплатформні програмні каркаси для створення GUI, наприклад, .Net MAUI в C# чи Tkinter у Python. Ми ж розглянемо бібліотеку Spec 2.0, яку використовують у сучасній багатоплатформній вільно поширюваній системі програмування Pharo [1]. Pharo (Фарó) – це і середовище програмування, орієнтоване на безпосередню взаємодію з "живими" об'єктами, і універсальна динамічно типізована об'єктно-орієнтована мова програмування, яку можна використовувати для побудови класичних і вебзастосунків. Синтаксис Pharo стисло описано в [2]. Там же описано процес

побудови віконного застосунку. Докладно вивчити синтаксис мови і ознайомитися зі способами використання середовища можна в [3].

Віконний інтерфейс класичного застосунку у Pharo будують за допомогою програмного каркасу Spec 2.0, який має певні особливості. Під час використання Spec програміст взаємодіє з демонстратором компонента, а не безпосередньо з графічним об'єктом. Демонстратори можуть використовувати різні графічні бібліотеки. Нині доступні Morphic та GTK, а незабаром стане до ладу Toplo. Завдяки відокремленому шару графіки GUI застосунків Pharo виглядає та функціонує однаково на різних платформах. Головним принципом Spec є повторне використання логіки поведінки GUI та компонування його видимих складових [4] – це інша особливість Spec, яка наділяє його безперечними перевагами над схожими бібліотеками. Програмування зі Spec є компонентно-орієнтованим [2]: будь-яку розроблену частину інтерфейсу можна трактувати як окремий компонент і використовувати повторно в різних проєктах. Компоненти Spec однаково успішно можна застосовувати для побудови вікон довільної складності. Примітно, що значна частина інструментів самого середовища програмування створена з використанням Spec: налагоджувач, менеджер Git-репозиторіїв Iceberg, хранитель змін коду Change Sorter, оглядач якості коду Critics Browser.

Усі візуальні компоненти Spec мають стандартний вигляд з дотриманням єдиного стилю. Щоб налаштувати особливий шрифт чи колір для окремих частин GUI, застосовують таблицю стилів, визначену на рівні екземпляра застосунку. Такий підхід збирає все керування зовнішнім виглядом застосунку в один метод, що полегшує налаштування застосунку тоді, коли інтерфейс містить багато схожих елементів. Але така централізація може зумовити певні труднощі, адже демонстратор компонента створюють в одному місці, а налаштовувати параметри його вигляду доводиться в іншому. Ми з'ясуємо, як можна усунути такий недолік Spec. Продемонструємо також, як доповнити наявний перелік візуальних компонент новими корисними демонстраторами, зокрема відомими з інших бібліотек. У статті використано результати, які отримали під час написання бакалаврської роботи [5].

2. Особливості стилізації у Spec

У Spec можна стилізувати властивості п'яти типів: *Container, Draw, Font, Text, Geometry*, доступ до яких надають відповідні підкласи *SpPropertyStyle*: *SpContainerStyle, SpDrawStyle, SpFontStyle, SpTextStyle* і *SpGeometryStyle* [4]. Таблицю стилів для застосунку Spec, який використовує графічну бібліотеку Morphic, описують рядком у форматі STON (*Smalltalk Object Notation*). Такий рядок розпізнають і перетворюють на елементи стилю.

Продемонструємо на прикладі зумисно спрощеного застосунку, які кроки потрібно виконати, щоб стилізувати елементи GUI. Демонстратор застосунку міститиме тільки дві кнопки (*firstButton* і *secondButton*).

```
SpPresenter << #MyPresenter
   slots: { #firstButton . #secondButton }
   package: 'MyExample'

MyPresenter >> defaultLayout
" Макет вікна розташовує кнопки в ряд "
   ^ SpBoxLayout newLeftToRight
```

```
      add: firstButton; add: secondButton; yourself
```

Щоб задати таблицю стилів, доведеться визначити клас застосунку й оголосити в ньому відповідний метод.

```
SpApplication << #MyApplication
   package: 'MyExample'

MyApplication >> styleSheet
" Стандартну таблицю стилів доповнено стилем fontSize14 "
   ^ super styleSheet, (SpStyleVariableSTONReader fromString:
      '.application [
         .fontSize14 [ Font { #size: 14 } ] ]')
```

Тепер стиль *fontSize14* можна використати під час налаштування кнопок.

```
MyPresenter >> initializePresenters
" Створення і налаштування вкладених демонстраторів.
  Другу кнопку налаштовано із застосуванням стилю "
   firstButton := self newButton.
   secondButton := self newButton.
   firstButton label: 'First'.
   secondButton
      label: 'Second';
      addStyle: 'fontSize14'
```

Залишилося пов'язати екземпляри демонстратора і застосунку. Зазвичай це роблять у методі *start* застосунку.

```
MyApplication >> start
   MyPresenter new application: self; open
```

Готово. Тепер застосунок можна запустити за допомогою *MyApplication new start* і переконатися, що друга кнопка містить напис шрифтом кеглю 14.

Видно, що заради одного простого налаштування нам довелося виконати кілька додаткових кроків: оголосити клас застосунку і два методи в ньому. Для правильного визначення стилю довелося також пильнувати дотримання формату STON. Використання стилю відбулося далеко від місця оголошення, що також ускладнює правильне його застосування. Запропонуємо декларативний спосіб налаштування вигляду такий, за якого користувач використовує повідомлення звичного у Pharo вигляду. Наприклад, *secondButton fontSize: 14*, щоб задати потрібний розмір шрифту. Оголосимо для цього новий базовий клас демонстратора, який огортатиме стандартний демонстратор і виконуватиме рутинні дії щодо налаштування стилю.

3. Клас демонстратора, придатний для налаштування

Централізовані конфігурації на основі STON досить незручні, бо не надають зворотного зв'язку, потребують ручного редагування довгих рядків. Суттєвим поліпшенням стане можливість задавати значення візуальних властивостей безпосередньо елементові, навіть без створення *SpApplication*. Нові можливості будуть

корисним доповненням бібліотеки Spec і не шкодитимуть усталеним підходам. Щоб забезпечити повну сумісність з наявною архітектурою Spec 2.0, новий клас демонстратора *BasePresenter* наслідуємо від стандартного *SpPresenter*. Усі компоненти, до яких можна застосувати хоча б один із стилів Draw, Container, Geometry, Font або Text, мусять наслідувати від *BasePresenter*.

Spec влаштована так, що уникнути використання STON та *SpConfiguration* не вдасться. Натомість приховаємо ці дії від користувача в методах *BasePresenter*. Замість написання стилів у STON, розробник задаватиме потрібні властивості безпосередньо в коді. Наприклад, компоненту можна буде надсилати повідомлення *fontSize:*, *color:*, *borderWidth:* тощо. Екземпляр *BasePresenter* міститиме візуальний компонент і словник заданих для нього налаштувань. Екземпляр самостійно сформує зі словника стиль і застосує його до свого вкладеного компонента. Вкладений компонент буде конкретизовано в майбутніх підкласах.

Щоб дати змогу задавати властивості безпосередньо кожному компоненту, треба мати окремий механізм, який би відповідав за генерацію STON-рядка на підставі заданих властивостей і додавання сформованого стилю до таблиці стилів аплікації. З цією метою оголошено окремий службовий клас *StyleManager*. Його головна відповідальність – перетворення простого набору пар "ключ → значення" (наприклад, fontSize → 14, color → #black) на валідний STON-рядок. Крім генерування рядків стилю, *StyleManager* також виконує роль реєстратора стилів. Зрештою, компоненти передаватимуть *StyleManager*'у словник властивостей та отримуватимуть готовий до застосування стиль. З погляду користувача всі стилістичні властивості задаватимуться безпосередньо об'єкту.

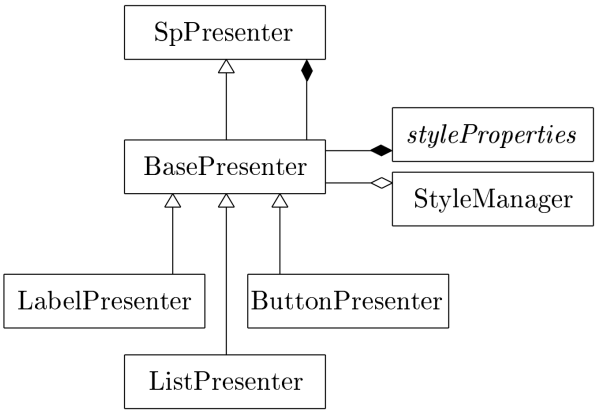

Рис. 1. Схема застосунку

Діаграму згаданих класів зображено на рис. 1.

Продемонструємо найважливіші частини програмного коду та пояснимо їх. Розпочнемо з оголошення *BasePresenter*.

```
SpPresenter << #BasePresenter
   slots: { #styleProperties . #mainPresenter . #appliedStyleName };
   tag: 'BaseComponents';
   package : 'SpecComponentsLibrary'
```

Тут *styleProperties* – словник заданих налаштувань, *mainPresenter* – демонстратор, до якого застосовують налаштування, *appliedStyleName* – автоматично згенероване ім'я стилю.

"Робочими конячками" цього класу є такі методи.

```
BasePresenter >> setGeneratedStyleProperty: key value: val
   self styleProperties at: key put: val

BasePresenter >> generateAndApplyStyle
   | allStyles |
   appliedStyleName ifNotNil: [
      mainPresenter removeStyle: appliedStyleName.
      allStyles := self application configuration styleSheet styles.
      allStyles := allStyles reject: [ :each |
                     (each isKindOf: SpClassStyle) and: [
                     each name = appliedStyleName ] ].
      self application configuration styleSheet styles: allStyles ].
   appliedStyleName := StyleManager
                     installGeneratedStyleFrom: self styleProperties
                     in: self application.
   mainPresenter addStyle: appliedStyleName
```

Перший з методів тільки зберігає до словника чергове налаштування, а другий відповідає за створення і застосування стилів. Перед додаванням нового стилю видаляємо старий, якщо він існує. Також, щоб уникнути накопичення зайвих стилів, попередній видаляємо з таблиці стилів аплікації. Новий стиль генерує клас *StyleManager* на основі властивостей зі словника *styleProperties*. (Щоб довідатися, як влаштовано *StyleManager*, перегляньте репозиторій [6].)

Тепер легко оголосити методи налаштування візуальних властивостей. Наприклад, колір тла компонента задає метод

```
BasePresenter >> backgroundColor: aColor
   self setGeneratedStyleProperty: #backgroundColor value: aColor.
   self generateAndApplyStyle
```

Подібно влаштовано інші методи: *bold*, *borderColor:*, *borderWidth:*, *color:*, *fontFamily:* тощо. Екземпляр *BasePresenter* у змінній *mainPresenter* містить деякий стандартний демонстратор, наприклад, напис *SpLabelPresenter* або список *SpListPresenter*. Як задати текст напису чи елементи списку? Для цього користувач мав би надіслати відповідне повідомлення вкладеному демонстратору. Щоб надати таку можливість, можна оголосити селектор для доступу до *mainPresenter*. Тоді налаштування напису виглядало б так:

```
   styledLabel mainPresenter label: 'Hello, World!'
```

Використання селектора ускладнює спілкування з демонстратором, тому запропонуємо інший спосіб. Ми могли б визначити в класі *BasePresenter* метод *label:* так, щоб делегувати вкладеному демонстраторові. Але тоді довелося б визначити більше десяти таких методів. На щастя, Pharo надає легкий спосіб визначити власний спосіб реагування на незрозумілі повідомлення: якщо об'єкт не знаходить методу для

опрацювання деякого отриманого повідомлення, то він отримує інше повідомлення – *doesNotUnderstand:*, аргументом якого є те саме незрозуміле повідомлення. Якщо оголосити метод *doesNotUnderstand:*, то можна переадресувати повідомлення тому, хто міг би його опрацювати. Так і зробимо.

```
BasePresenter >> doesNotUnderstand: aMessage
   ^ aMessage sendTo: mainPresenter
```

Тепер екземпляр *BasePresenter* передаватиме вкладеному демонстраторові всі повідомлення, які не зможе опрацювати сам. Задати текст напису можна зовсім просто.

```
   styledLabel label: 'Hello, World!'
```

4. Використання класу *BasePresenter*

Абстрактний клас *BasePresenter* визначає спільну поведінку всіх демонстраторів, придатних до налаштування, але не конкретизує значення змінної *mainPresenter*. Зробимо це в підкласах. Наприклад, демонстратор списку, який можна налаштовувати звичайними повідомленнями, оголошено так.

```
BasePresenter << ListPresenter
   tag: 'BaseComponents';
   package : 'SpecComponentsLibrary'

ListPresenter >> initializePresenters
 super initializePresenters.
 self mainPresenter: SpListPresenter new.
 self focusOrder add: self mainPresenter
```

Подібно оголошено класи інших часто вживаних компонентів: *ButtonPresenter*, *LabelPresenter*, *CheckBoxPresenter*, *RadioButtonPresenter* [6]. Звичайно, цей перелік не містить усі наявні компоненти бібліотеки Spec. За потреби можна оголосити відповідний підклас і визначити в ньому єдиний метод *initializePresenter*. Щоб не змушувати користувача оголошувати такі підкласи, було визначено допоміжний клас-декоратор.

```
Object << PresenterDecorator
   tag: 'Style';
   package : 'SpecComponentsLibrary'

PresenterDecorator class >> wrap: aPresenterOrClass
   | instance |
   instance := aPresenterOrClass isBehavior
      ifTrue: [ aPresenterOrClass new ] ifFalse: [ aPresenterOrClass ].
   ^ BasePresenter new
      mainPresenter: instance; yourself
```

Продемонструємо, як з його допомогою створити і налаштувати поле для введення тексту.

```
styled := PresenterDecorator wrap: SpTextInputFieldPresenter.
styled fontSize: 14; backgroundColor: #lightYellow;
   borderWidth: 1; borderColor: #black.
styled text: 'Hello, World!'
```

Так отримаємо обведене тонкою чорною рамкою поле з світло-жовтим тлом, у полі виведено текст *'Hello, World!'* шрифтом кеглю 14 пунктів.

5. Додаткові компоненти

Бібліотека Spec 2.0, безперечно, є потужним інструментом для створення користувацьких інтерфейсів [4]. Вона дає змогу працювати зі складними компонентами та розміткою, ізолювати створення GUI від основної логіки. Проте відчутна нестача деяких важливих базових компонентів. Наприклад, у стандартній бібліотеці є простий компонент для вибору дати, але немає жодного аналога для вибору часу. Є клас *SpFilteringListPresenter*, який реалізує список із фільтруванням через текстове поле, але немає таблиці з подібною функцією. Також відчувається нестача комбінованих спадних списків, таблиць з розбиттям на сторінки тощо. Такі компоненти досить поширені у багатьох прикладних інтерфейсах, тому доволі ймовірно, що користувачеві доведеться реалізовувати їх самостійно. Заповнимо ці прогалини компонентами власного авторства [6]. Далі наведено їхній стислий опис.

Введення даних і вибір варіантів. Один із найпоширеніших сценаріїв взаємодії з користувацьким інтерфейсом – введення значень і вибір із набору запропонованих варіантів. Тому на етапі проєктування одразу було виділено окрему групу компонентів, відповідальних за такі сценарії.

Текстове поле з підтримкою перевірки введення реалізує *TextInputPresenter*, підклас *BasePresenter*. Компонент має два режими роботи: з перевіркою та без неї. Режим задають повідомленням *isValidationField: aBoolean*. Компонент пам'ятає словник правил, ключем якого є блок з описом умови, значенням – відповідне діагностичне повідомлення. Введений текст перевіряється на коректність після кожної зміни. Якщо якась з умов не виконується, то під полем буде виведено повідомлення про це, а поле буде обведено червоною рамкою. Користувач може конструювати свої правила перевірки або використати готові з класу *ValidationRules*. Програмно дізнатися про результат перевірки можна через властивість *isValid*.

Окрему підгрупу становлять компоненти, пов'язані з вибором одного або кількох значень. У Spec 2.0 реалізовано клас *SpDropListPresenter*, який надає спадний список. Якщо елементів багато, то користувач змушений вручну гортати список, щоб знайти потрібне значення. Тому було розроблено комбінований список *ComboBoxPresenter* – компонент, який поєднує поле введення для фільтра зі спадним переліком варіантів. Поле фільтра значно полегшує пошук і робить використання такого компонента зручнішим для кінцевого користувача. API компонента складається з методів *items:*, *selectedIndex*, *selectedItem* і події *whenSelectionChangedDo:*. Для випадків, коли потрібно вибрати кілька значень зі списку одночасно, пропонуємо *MultiSelectComboBoxPresenter* – комбінований список з підтримкою множинного вибору. Його можна було б реалізувати на основі звичайного комбінованого списку, дозволивши множинне виділення. Однак такий підхід змушує користувача затискати клавішу [Ctrl] для вибору кількох елементів. Саме тому варіанти було подано як список прапорців (checkboxes). Для вибору значення достатньо одного клацання, що суттєво спрощує взаємодію з компонентом. Його API – *items:*, *selectedIn-*

dexes, *whenSelectionChangedDo:*. Для відображення спадних списків поверх інших компонентів застосунку було використано спеціальні графічні сутності: морфи *PluggableListMorph* і *SpFTTableMorph*.

Ще однією важливою складової цієї категорії є групи перемикачів: залежних або незалежних. Їх реалізовано класами *RadioButtonGroupPresenter* і *CheckBoxGroupPresenter*, відповідно. Група залежних перемикачів дає змогу вибрати лише один варіант, а група прапорців – кілька, усі або жодного. Для груп обох типів можна додати рамку та заголовок, розмістити елементи в довільній кількості стовпців. Групи відрізняються тільки типом перемикачів, тому їхню спільну частину визначає клас *AbstractSelectableGroupPresenter*. Його API складається з методів для налаштування колекції об'єктів, які відображатимуться в перемикачах (*items:*, *addItem:*, *removeItemAt:*) і методів налаштування зовнішнього вигляду (*showFrame:*, *showTitle:*, *title:*, *columnCount:*). Підкласи додають свої можливості: *RadioButtonGroupPresenter* – *itemSelected*, *indexSelected*; *CheckBoxGroupPresenter* – *checkedIndexes*, *checkedItems*, *clearSelection* та *isCheckedAt:*.

Для організації вибору дати в Spec 2.0 є відповідний компонент – *SpDatePresenter*, але немає компонента для вибору часу. Тому було розроблено *TimePickerPresenter*. Компонент вибору дати складається з поля та кнопки, яка відкриває інтерактивний календар. Щоб зберегти стилістичну єдність, *TimePickerPresenter* має схожу будову. Він складається з текстового поля, яке заборонено редагувати, та кнопки, що відкриває модальне вікно з інтерактивним годинником. Годинник має мати два режими: вибір годин (від 0 до 23) і вибір хвилини (від 0 до 59). Після завершення діалогу в компонента можна довідатися про зроблений вибір за допомогою повідомлень *hour* і *minutes*.

Побудова форм. Одним із головних складників майже кожного користувацького інтерфейсу є введення та редагування даних про якийсь об'єкт. Для цього використовуються форми. Процес їх створення доволі рутинний, оскільки щоразу доводиться вручну створювати поля, пов'язувати їх з підписами, додавати кнопки підтвердження, перевірку введених значень тощо. Тому доцільним є впровадження компонента, який візьме на себе формування повноцінної форми та налаштування її властивостей і поведінки. Таким компонентом став *DynamicFormBuilder*. Тепер форму можна побудувати декларативно, тобто шляхом послідовного надсилання повідомлень для додавання частин форми. Кожну частину описують через ключ, підпис ті додаткові параметри, залежно від її типу. Частинами можуть бути текстові поля, групи прапорців і перемикачів, спадні й комбіновані списки тощо. Компоненти і підписи до них автоматично розташовуються в сітці. Для зручності форма має дві кнопки, поведінку яких можна налаштувати через обробники подій. За бажанням і ці кнопки, і заголовок можна приховати. Додатково передбачена можливість вставки власного елемента у верхню частину форми (логотип, індикатор кроків тощо). Кожний компонент форми має унікальний ключ, за яким можна зчитати введене значення після заповнення форми. Також передбачено метод для автоматичного збирання даних зі всіх компонентів. Такий підхід значно пришвидшує створення форм введення інформації та дає змогу уникнути дублювання коду.

Як уже було зазначено, компоненти форми створюють декларативно, тобто через спеціалізовані методи екземпляра класу *DynamicFormBuilder*. Наразі доступні такі види демонстраторів:

- текстове поле з перевіркою правильності введення даних (метод *textField: la-*

bel: placeholder: rules:);
- спадний список (метод *dropList: label: items:*);
- комбінований список (метод *comboBox: label: items:*);
- комбінований список з множинним вибором (метод *multiComboBox: label: items:*);
- окремий прапорець (метод *checkbox: label: text:*);
- група прапорців (метод *checkboxGroup: label: items: columns: frame:*);
- група перемикачів (метод *radioGroup: label: items: columns: frame:*);
- компонент для вибору дати (метод *datePicker: label:*);
- компонент для вибору часу (метод *timePicker: label:*).

У кожен з цих методів потрібно передати ключ компонента, підпис (який буде відображено ліворуч) та параметри, характерні для відповідного компонента.

Користувач може відстежувати натискання кнопок через події *onSubmit:* та *onBack:*. Для збирання значень із полів визначено метод *collectValues*.

Відображення колекцій даних. У багатьох інтерфейсах важливим аспектом є не лише отримання даних від користувача, а й зручне їх відображення у вигляді списків і таблиць. Списки у Spec 2.0 реалізовані достатньо повно, адже є варіанти з фільтруванням, використанням компонентів замість простих елементів, редагуванням тощо. А от із таблицями не все так добре, зокрема одразу помітно, що немає таблиці з підтримкою фільтрування. Саме тому було реалізовано *SearchableTablePresenter* – таблицю з пошуком. Крім самої таблиці, компонент містить поле для введення запиту та комбінований список для вибору стовпців, у яких відбуватиметься фільтрування. Після кожної зміни пошукового запиту таблиця автоматично оновлюється, залишаючи лише ті елементи, які відповідають критеріям пошуку. Щоб налаштувати екземпляр таблиці, йому надсилають повідомлення *columns: aCollection*, аргумент якого містить пари "назва стовпця → ключ". Видимістю поля фільтра керують методи *showSearchContainer* і *hideSearchContainer*.

Якщо кількість рядків у таблиці достатньо велика, то неефективно показувати всі записи одразу, доцільно реалізувати таблицю з розбиттям на сторінки та навігацією між ними. Такий підхід дасть змогу працювати з великими обсягами даних, не перевантажуючи інтерфейс. Тому було створено компонент *PaginatedTablePresenter*, підклас *SearchableTablePresenter*. Він додатково містить навігаційну панель, на якій розміщено кнопки *Вперед*, *Назад*, кнопки з номерами окремих сторінок, текстове поле для ручного введення номера сторінки. Переходами між сторінками таблиці можна керувати програмно за допомогою методів *nextPage*, *previousPage*, *goToPage:*, *currentPage*, *totalPages*. Для налаштування розміру сторінки таблиці застосовують повідомлення *itemsPerPage: anInt*.

Візуалізація покрокового виконання. У випадках, коли користувач має пройти певну послідовність етапів взаємодії, зручним рішенням буде покроковий інтерфейс. Наприклад, у випадку, коли потрібно зібрати велику кількість даних, але немає бажання перевантажувати користувача великою формою, поділ на кроки полегшуватиме сприйняття та надаватиме можливість зберігати проміжні значення або перевіряти дані частинами. Для такого сценарію було реалізовано компонент *WizardPresenter*. Він складається з заголовка для відображення кроків і поточного прогресу, контейнера, в який користувач може вставити відповідний крокові демонстратор, і кнопок для переміщення між етапами. Проєктна модель передбачає, що кроки незалежні, кожен з них описують окремо, а *WizardPresenter* відповідає тільки за загальну навігацію та інтеграцію. Екземпляр *WizardPresenter* налаштовують

повідомленням *steps:*, в аргументі якого передають колекцію словників спеціального вигляду. Кожен такий словник містить точно два записи: "*#title –> назва кроку*" і "*#content –> демонстратор*". Логіку створення та оновлення заголовка винесено у клас *WizardStepHeaderPresenter*. Для побудови цього компонента використано бібліотеку Roassal [7]. Приклад заголовка зображено на рис. 2.

Рис. 2. Вигляд готового заголовка у *WizardPresenter*

Сповіщення. Під час роботи програм часто виникає потреба показати користувачу коротке повідомлення. Для цього було реалізовано компонент *NotificationPresenter*. У його основі – морф, який відображається поверх інших вікон. Його головне завдання – стисло інформувати користувача про успішне збереження, помилку, попередження тощо. Повідомлення не потребує ніяких дій від користувача і зникає через заданий проміжок часу. На відміну від повідомлень стандартного *UIManager*, зовнішній вигляд нового можна повністю налаштувати. Зокрема, надано можливість задавати колір, розмір, тривалість відображення тощо. Текст повідомлення можна стилізувати, щоб пристосувати компонент до стилю свого інтерфейсу. Також передбачено механізм позиціонування повідомлення стосовно заданого вікна. Для налаштування повідомлення можна використати методи *title:*, *message:*, *position:*, *duration:*, *window:*, *type:* і один з визначених типів *#success*, *#warning*, *#error*. Якщо не задати тип, то повідомлення відобразиться на сірому тлі.

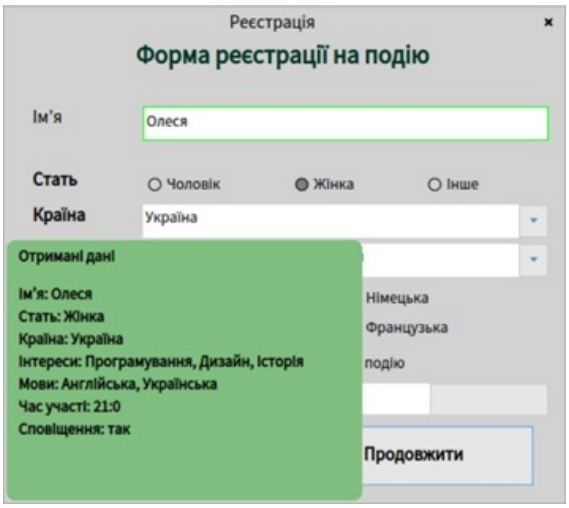

Рис. 3. Заповнена форма та сповіщення з зібраними даними

6. Тестування та оцінка ефективності

Щоб підтвердити працездатність розробленої бібліотеки, її було протестовано за допомогою стандартного засобу тестування в середовищі Pharo – SUnit. Для кожного компонента створено відповідний клас тестів, який містить модульні тести для більшості методів. Середовище програмування вміє пов'язувати тести з відповідними методами, що значно спрощує перевірку і окремих методів, і бібліотеки в цілому. У тестах перевірено і позитивні, і негативні сценарії. Негативними вважаємо ситуації пов'язані з обробкою некоректних даних, наприклад, порожніх списків, недопустимих індексів, значень, яких немає, тощо.

Для перевірки зручності використання розробленої бібліотеки було випробувано в ділі окремі компоненти та розроблено кілька застосунків різної складності, серед яких форма реєстрації на подію та читацький щоденник. На початку цієї статті наведено приклад стилізації кнопки *SpButtonPresenter* стандартними засобами Spec. Довелося оголосити додатковий клас і два методи. За допомогою розробленого *ButtonPresenter* такі ж налаштування можна виконати за допомогою одного додаткового повідомлення екземплярові кнопки.

Серед розроблених елементів треба виділити *DynamicFormBuilder* і *WizardPresenter*. Ці компоненти дають змогу автоматично створювати доволі складні шаблонні інтерфейси, які часто використовують у застосунках. Їх застосування зменшує загальний обсяг коду в кілька разів. Як приклад можна навести форми реєстрації на подію. Для ручного створення форми довелося оголосити методи *initializePresenters* і *defaultLayout* загального обсягу близько 60 рядків. З використанням *DynamicFormBuilder* уся реалізація зайняла лише 18 рядків, до того ж не довелося турбуватися про створення написів і макета форми. Вікно застосунку реєстрації участі в деякій події зображено на рис. 3.

Використання розробленої бібліотеки для написання застосунків у середовищі Pharo значно пришвидшило створення графічних інтерфейсів. Одніа з головних причин цього – реалізація декларативної стилізації. Завдяки такому підходу значно спрощується задання властивостей і підвищується читабельність коду. Розробник одразу бачить параметри, які застосовують до компонента, а не просто назву стилю, як у стандартному підході. Її використання у типових сценаріях взаємодії з користувачем допомагає в рази зменшити обсяг коду. Це, безумовно, позитивно впливає на швидкість розробки. Навіть під час реалізації самої бібліотеки використовувалися раніше створені компоненти.

7. Висновки

Описано розроблену бібліотеку графічних компонентів [5], яка розширює стандартні можливості бібліотеки Spec 2.0 [4]. Було проведено аналіз архітектури Spec 2.0 та виявлено типові проблеми, які виникають під час її використання, а саме: обмежений набір компонентів і складна система стилізації. На підставі цього спроєктовано та реалізовано бібліотеку, яка усуває зазначені недоліки. Додано часто вживані, але не наявні у стандартній реалізації компоненти (поля з перевіркою правильності введення, комбіновані списки з фільтрацією, таблиці з пошуком і розбиттям на сторінки, групи перемикачів, форми, покрокові інтерфейси тощо). Крім того, впроваджено зручний механізм стилізації, який дає підстави задавати властивості безпосереднього у коді – без потреби ручного редагування STON-

конфігурацій. Забезпечено повну сумісність з архітектурою Spec 2.0. Усі компоненти бібліотеки наслідують *SpPresenter*, перевизначають методи *initializePresenters*, *defaultLayout*, підтримують події та інші стандартні механізми. Вони вільно поєднуються зі стандартними елементами Spec 2.0 без необхідності додаткової адаптації.

За допомогою модульного тестування було підтверджено коректність роботи основного функціонала бібліотеки. Аналіз підтвердив, що використання цієї бібліотеки під час розробки інтерфейсів суттєво зменшує обсяг коду та поліпшує його читабельність. За потреби її можна легко доповнювати новими компонентами. Код бібліотеки доступний за посиланням [6].

Список використаної літератури

1. Pharo [Електронний ресурс] / Pharo Consortium // Official Page of Pharo, 2025. – Режим доступу: https://pharo.org/ – Назва з екрану.
2. Ярошко С. Побудова віконних застосунків у середовищі Pharo за допомогою бібліотеки Spec / Сергій Ярошко, Оксана Ярошко // Вісник Львівського університету. – Серія прикладна математика та інформатика. – 2023. – Вип. 31. – С. 160–172.
3. Дюкас С. Pharo 9 на прикладах / С. Дюкас, Дж. Ракіч [та ін.]; пер. з англ. С. Ярошко. – Львів: ЛНУ ім. Івана Франка, 2022. – 270 с. [Електронне видання] – Режим доступу: http://books.pharo.org/pharo-by-example9/ – Назва з екрану.
4. De Hondt K. Application Building with Spec 2.0 / K. De Hondt, S. Ducasse, S. Jordan-Montano, E. Lorenzano. – Books on Demand, 2024. – 242 p. [Електронне видання] – Режим доступу: http://books.pharo.org/BuildingApplicationsWithSpec/ – Назва з екрану.
5. Байдала О.Т. Розробка бібліотеки графічних компонентів на основі Spec 2 для Pharo / Олеся Байдала, наук. кер. Сергій Ярошко // Рукопис кваліфікаційної (бакалаврської) роботи. – ЛНУ ім. Івана Франка, 2025. – 75 с.
6. Spec Components Library for Pharo [Електронний ресурс] / Олеся Байдала // GitHub, 2025. – Режим доступу: https://github.com/Olesia32/pharo-spec-components – Назва з екрану.
7. Bergel A. Agile Visualization with Pharo / A. Bergel. – Berkeley, CA: Apress, 2024. – 266 c.

Стаття: надійшла до редколегії 09.07.2025
доопрацьована 24.09.2025
прийнята до друку 08.10.2025

IMPROVEMENT OF THE VISUAL COMPONENT LIBRARY SPEC 2.0 FOR PHARO

O. Baidala, S. Yaroshko

Ivan Franko National University of Lviv,
1, Universytetska str., 79000, Lviv, Ukraine,
e-mail: serhiy.yaroshko@lnu.edu.ua

Pharo is a modern object-oriented, dynamically typed, multi-purpose programming language. Spec is a framework in Pharo for developing user interfaces of desktop applications. Using Spec looks like a component-oriented programming. Every visual component in Spec can act as a finished GUI or as a part of a larger one. The fundamental principle behind Spec is the reuse of user interface logic and its visual composition [4]. However, despite its rich functionality, Spec 2.0 has shortcomings that complicate and slow down the developer's work. In particular, it is about the absence of some frequently used components and the complexity of their styling. The above reasons encourage the creation of a separate library that will complement and expand the capabilities of Spec.

The article describes a new way to configure the properties of the visual components of the Spec library in the Pharo environment. The new base demonstrator class *BasePresenter* is built, which provides a declarative interface for configuring color, font, border, size, etc., and automatically generates and applies the appropriate design style. We supplement the Spec library with new visual components using the new demonstrator. The decorator class *PresenterDecorator* is declared to wrap any standard Spec's presenter and enrich it with new styling capabilities. Several concrete subclasses are declared: *ButtonPresenter*, *LabelPresenter*, *ListPresenter*, *CheckBoxPresenter*, *RadioButtonPresenter*. It is easy to understand the purpose of each of them.

More complex components are also developed.*TextInputPresenter* provides the ability to validate the input text automatically. We can select one or more values with the help of *ComboBoxPresenter*, *MultiSelectComboBoxPresenter*, *RadioButtonGroupPresenter*, or *CheckBoxGroupPresenter*. *TimePickerPresenter* gives us an interactive time picker. *DynamicFormBuilder* is a potent component. It can build any input form, driven by a sequence of messages like *textField:*, *dropList:*, *radioGroup:*, etc. *SearchableTablePresenter* and *PaginatedTablePresenter* provide table demonstrators with a filter field and pagination, respectively. *WizardPresenter* can organize and visualize a step-by-step scenario of interaction with a user.

The examples show that the proposed approach accelerates the application's graphical interface creation.

Key words: Pharo, Spec, graphical user interface, visual component, styling, window application, class, object.