Розділ 13

Morphic

Говорячи про *Morphic*, маємо на увазі графічний інтерфейс *Pharo*. *Morphic* написаний на *Pharo*, тому він повністю підртимується на усіх операційних системах. З цього випливає, що *Pharo* виглядає одинаково на Unix, MacOS та Windows. Чим *Morphic* відрізняється від більшості інших графічних інструментів, так це тим, що в нього немає окремих режимів для *створення* і *запуску* інтерфейсу: усі графічні елементи можуть бути зібрані та розібрані користувачем в любий момент часу. (Ми вдячні Hilaire Fernandes за дозвіл побудувати цей розділ на основі його оригінальної статті на французькій.)

# 13.1 Історія

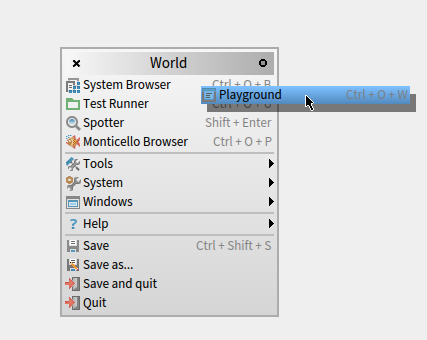
*Morphic* був розроблений Дж. Мелоні(John Maloney) і Р. Смітом(Randy Smith) для мови програмування *Self*, приблизно у 1993р. Пізніше Мелоні написав нову версію *Morphic* для *Squeak*, але оригінальні ідеї версії для Self досі живі і зустрічаються у *Pharo Morphic*: прямота та живучість(*directness* and *liveness*). Прямота означає, що форми на екрані є об’єктами, які можуть бути розглянуті та змінені напряму, тобто клікнувши на них мишкою. Живучість означає, що користувацький інтерфейс завжди має реагувати на дії користувача: інформація на екрані має динамічно змінюватися при зміні стану, який її описує. Простий приклад - ми можемо відділити елемент меню і залишити його як кнопку.

Відкриваємо “*World Menu*” і робимо по ньому мета-клік один раз, щоб появилося його morphic-меню. Потім знову робимо мета-клік на елемент меню, який хочемо відділити, щоб відкрити morphic-меню для цього елементу. Тепер перенесіть цей елемент кудись на екрані, затиснувши відповідну кнопку (малюнок 13.1), як це показано на малюнку 13.2.

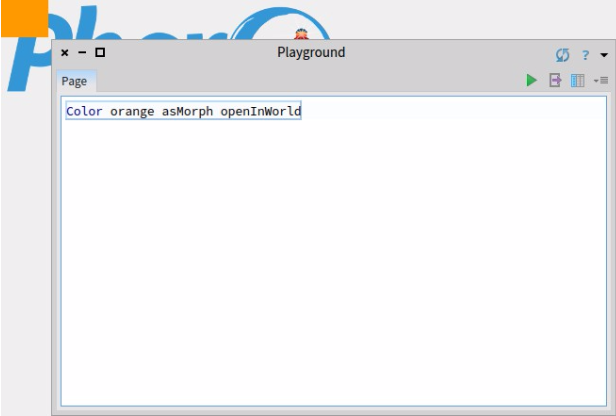
Усі об’єкти, які ви бачите на екрані *Pharo* є “морфами”, тобто вони є об’єктами субкласу *Morph*. Сам по собі, *Morph* - це клас з багатьма методами; це надає змогу похідним класам реалізувати цікаву поведінку малою кількістю коду. Можна створити морф, щоб відобразити довільний об’єкт, хоча отримане відображення залежить від об’єкту!



**Малюнок 13.1**:Обробник захоплення



**Малюнок 13.2**: Відділяємо елемент меню, щоб зробити його окремою кнопкою



**Малюнок 13.3**: Color orange asMorph openInWorld з нашим новим методом.

Щоб створити морф для представлення об’єкту типу “рядок”, виконайте наступний код у Пісочниці:

**'морф' asMorph openInWorld**

Це створює *Morph* для відображення рядка 'морф' і відкриває його (тобто відображує) на головному екрані. Ви маєте отримати графічний елемент ( *Morph* ), яким можна маніпулювати за допомогою натисків мишки.

Звісно ж, можливо визначати морфи з більш цікавим графічним представленням, ніж ми щойно бачили. Метод *asMorph* має реалізацію за замовчуванням у класі *Object class*, який всього лиш створює *StringMorph*. Наприклад, *Color tan asMorph* повертає *StringMorph* з результатом *Color tan printString*. Давайте спробуємо це змінити, щоб натомість отримати кольоровий прямокутник.

Відкрийте у браузері *Color class* і додайте до нього наступний метод:

**Color >> asMorph**

**^ Morph new color: self**

Тепер виконайте *Color orange asMorph openInWorld* у Пісочниці. Замість морфу у вигляді рядка, отримано оранжевий прямокутник (малюнок 13.3)!

## 13.2 Manipulating morphs

Морфи - це об’єкти, тому ми можемо маніпулювати ними як і усіма іншими об’єктами у *Pharo*: надсилаючи повідомлення, ми можемо змінювати їхні властивості, створювати нові підкласи *Morph* тощо.

Кожен морф, навіть той, що на даний момент не відображається, має позицію і розмір. Для зручності, усі морфи займають прямокутну область екрану; у разі, якщо вони неправильної форми, то їхня позиція і розмір складається з маленьких прямокутних коробок навколо них, так званих обмежуючих коробок. Метод *position* повертає точку(*Point*), яка описує координати верхнього лівого краю(морфу або його обмежуюччої коробки). Початком відліку системи координат монітору є верхній лівий кут. координата y зростає донизу, а х - вправо. Метод *extent*  також повертає точку, але ця точка визначається шириною та висотою мору, а не координатами.

Введіть даний код у Пісочницю і виконайте його:

**petro := Morph new color: Color blue.**

**petro openInWorld.**

**pavlo := Morph new color: Color red.**

**pavlo openInWorld**

Потім введіть *petro position* і потім *Print it*. Щоб посунути Петра, циклічно виконайте *petro position: (petro position + (10@4))* (Малюнок 13.4).

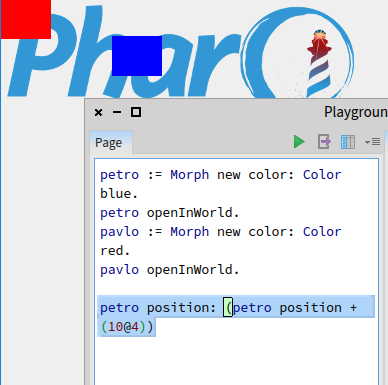
Аналогічні речі можна зробити і з розміром. *petro extend* відповідає за розмір Петра; щоб заставити його вирости, виконайте *petro extent: (petro extent \* 1.1)*. Щоб змінити колір морфа, надішліть йому *color: message* з потрібним об’єктом *Color* як аргумент, наприклад petro color: Color orange. Щоб додати прозорість, попробуйте *petro color: (Color orange alpha: 0.5).*

Для того, щоб Павло наслідував Павла, ви можете циклічно виконати даний код:

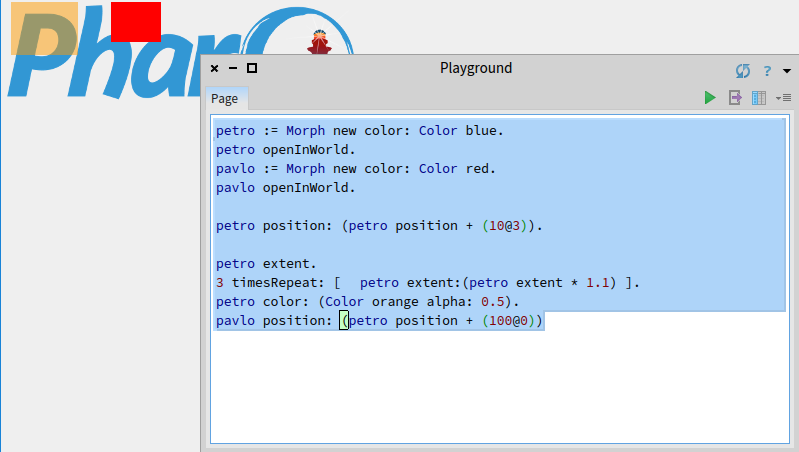
pavlo position: (petro position + (100@0))

Якщо ви виконаєте цей код після того, як порухаєте Петра мишкою, то Павло теж посунеться на 100пікселів вправо від Павла.

Результат видно на малюнку 13.5



**Малюнок 13.4**: Петро і Павло після 10ти ітерацій



**Малюнок 13.5**: Петро і Павло після 10ти ітерацій

## 13.3 Поєднання морфів

Одим із способів створення нових графічних відображень є розсташування одних морфів всередині інших. Це називається *композиція*; морфи можуть бути скомпоновані з любим рівнем вкладеності.

Можна розсташовувати один морф всередині іншого надсиланням повідомлення *addMorph:* до контейнера морфу.

Попробуйте додати один морф до іншого:

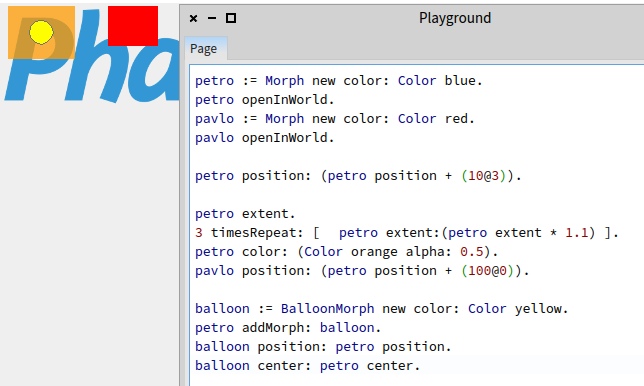
**balloon := BalloonMorph new color: Color yellow.**

**petro addMorph: balloon.**

**balloon position: petro position.**

Останній рядок розсташовує кульку за тими ж самими координатами, що й Петро. Зауважте, що координати внутрішнього морфа все одно відносно екрану, а не зовнішнього морфу. Доступно багато методів для розсташування морфа; ознайомтеся з документацією класу *Morph*(протокол *geometry*) самостійно. Наприклад, щоб розсташувати кульку всередині Петра, виконайте *balloon center: petro center.*

Якщо спробувати зараз пересунути мишкою кульку, то легко бачити, що ви пересуваєте Петра і два морфи загалом: кулька вбудована всередині Петра. Можна вбудувати більше морфів всередині Петра. Це можна зробити як за допомогою коду, так і вручну.



**Малюнок13.6**: Кулька розсташована всередині Петра, напівпрозорого оранджевого морфа.

## 13.4 Створення та відображення власного морфу



**Малюнок13.7**: *CrossMorph*; його розмір можна змінювати за бажанням.

Хоча можливо зробити багато цікавих і корисних графічних відображень, компнуючи морфи, інколи потрібно створити щось кардинально інакше.

Щоб це зробити, визначаємо підклас від *Morph* і перевантажуємо метод *drawOn:* таким чином, щоб змінити появу елемента.

Морф фреймворк надсилає повідомлення *drawOn:* морфу, коли потрібно перевідобразити морф на екрані. Параметром виступає різновид *Canvas*; очікувана поведінка - морф намалює себе на цьому полотні, у його межах. Давайте використаємо ці знання, щоб створити хрестоподібний морф.

Використовуючи браузер, визначимо новий клас *CrossMorph*, що унаслідується від *Morph*:

**Morph subclass: #CrossMorph**

**instanceVariableNames: ''**

**classVariableNames: ''**

**package: 'PBE-Morphic'**

Ми можемо визначити *drawOn:* метод наступним чином:

**CrossMorph >> drawOn: aCanvas**

**| crossHeight crossWidth horizontalBar verticalBar |**

**crossHeight := self height / 3.0.**

**crossWidth := self width / 3.0.**

**horizontalBar := self bounds insetBy: 0 @ crossHeight.**

**verticalBar := self bounds insetBy: crossWidth @ 0.**

**aCanvas fillRectangle: horizontalBar color: self color.**

**aCanvas fillRectangle: verticalBar color: self color**

Надіславши повідомлення *bounds,* морф відповідає, що це обмежуюча коробка, яка є екземпляром класу *прямокутник(Rectangle)*. Прямокутники розуміють велику кількість повідомлень, які створюють інші прямокутники з пов’язаної з ними геометрії. Тут ми використали повідомлення *insetBy:* з точкою як аргументом, щоб створити перше прямокутник зі зменшеною висотою, а потім прямокутник із зменшеною шириною.

Щоб протестувати новий морф, виконайте *CrossMorph new openInWorld*.

Резульат має бути схожий на те, що можна бачити на малюнку 13.7. Проте, залиається “чутлива зона” — де ми досі можемо клікнути на морф — це досі обмежуюча коробка. Давайте це виправимо.

Коли фреймворк намагається зрозуміти, які морфи знаходяться під курсором, він надсилає повідомлення *containsPoint*: до усіх морфів, чиї обмежуючі коробки знаходяться під вказівником мишки. Тому, щоб зменшити “чутливу зону” нашого морфу у формі хреста, нам потрібно переписати метод *containsPoint*:

Визначимо цей метод у класі *CrossMorph*:

**CrossMorph >> containsPoint: aPoint**

**| crossHeight crossWidth horizontalBar verticalBar |**

**crossHeight := self height / 3.0.**

**crossWidth := self width / 3.0.**

**horizontalBar := self bounds insetBy: 0 @ crossHeight.**

**verticalBar := self bounds insetBy: crossWidth @ 0.**

**^ (horizontalBar containsPoint: aPoint) or: [ verticalBar containsPoint: aPoint ]**

Цей метод аикористовує ту ж логіку, що й *drawOn*:, тому ми можемо бути впевнені, що точки, для яких *containsPoint*: дає позитивну відповідь, є ті ж самі, що ми замалювали у *drawOn*. Зверніть увагу на те, як ми використали метод *containsPoint*: класу *Rectangle*, щоб зробити важку роботу.

З вищеописаними методами є дві проблеми. Найбільш очевидним є те, що у нас повторюється код. Це легко призводить до помилок: якщо ми вирішили змінити розразхунок *horizontalBar* чи *verticalBar*, то ми легко можемо забути змінити ще декілька їхніх появ. Вихід - винести ці обрахунки у два окремих методи у приватному протоколі:

**CrossMorph >> horizontalBar**

**| crossHeight |**

**crossHeight := self height / 3.0.**

**^ self bounds insetBy: 0 @ crossHeight**

**CrossMorph >> verticalBar**

**| crossWidth |**

**crossWidth := self width / 3.0.**

**^ self bounds insetBy: crossWidth @ 0**

Ми можемо визначити *drawOn*: і *containsPoint*: використовуючи ці методи:

**CrossMorph >> drawOn: aCanvas**

**aCanvas fillRectangle: self horizontalBar color: self color.**

**aCanvas fillRectangle: self verticalBar color: self color**

**CrossMorph >> containsPoint: aPoint**

**^ (self horizontalBar containsPoint: aPoint) or: [ self verticalBar containsPoint: aPoint ]**

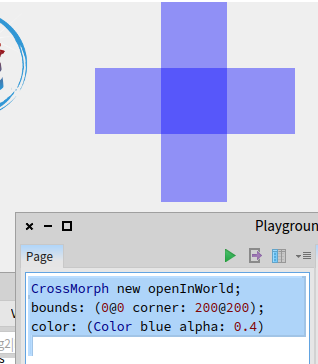
Такий код значно простіше зрозуміти, бо ми дали зрозумілі назви приватним методам. Це все так просто, що ви навіть могли не зауважити 2гу проблему: зона в центрі хреста, яка належить обом прямокутникам, замальовується двічі. Це не є важливо, коли колір повністю непрозорий, але ця вада стає помітною, коли ми використаємо навпівпрозорий колір для замальовки(малюнок 13.8)

Виконайте наступний код у пісочниці:

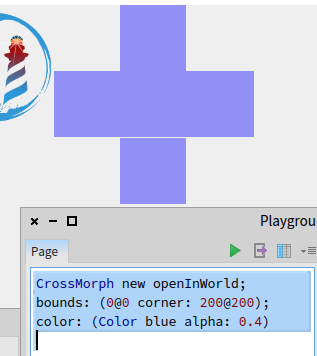
**CrossMorph new openInWorld;**

**bounds: (0@0 corner: 200@200);**

**color: (Color blue alpha: 0.4)**



**Малюнок 13.8**: Фігура в центрі двічі замальовується.



**Малюнок 13.9**: Рядок з незамальованими пікселями.

Вирішенням є поділити вертикальний прямокутник на три частини і замальовувати лише верх та низ.Знову знайдемо метод у класі *Rectangle*, який робить тяжку роботу за нас: *r1 areasOutside: r2* повертає масив прямокутників, які належать r1 і не належать r2. Ось приклад коду:

**CrossMorph >> drawOn: aCanvas**

**| topAndBottom |**

**aCanvas fillRectangle: self horizontalBar color: self color.**

**topAndBottom := self verticalBar areasOutside: self horizontalBar.**

**topAndBottom do: [ :each | aCanvas fillRectangle: each color: self color ]**

Цей код виглядає робочим, хоча якщо ви спробуєте змінити розмір, то можете зауважити, що у деяких випадках з’являється біла лінія товщиною в 1піксель (малюнок 13.9). Це через заокруглення: коли розмір для замальовки - не ціле число, *fillRectangle:*

*color*: заокруглює його нерівномірно, залишаючи один рядок пікселів незамальованим. Ми можемо виправити це, заокруглюючи самостійно, коли обраховуємо розміри прямокутників.

**CrossMorph >> horizontalBar**

**| crossHeight |**

**crossHeight := (self height / 3.0) rounded.**

**^ self bounds insetBy: 0 @ crossHeight**

**CrossMorph >> verticalBar**

**| crossWidth |**

**crossWidth := (self width / 3.0) rounded.**

**^ self bounds insetBy: crossWidth @ 0**

## 13.5 Взаємодія та анімації

Щоб побудувати живий користувацький інтерфейс за допомогою морфів, нам потрібно взаємодіяти з ними за допомогою мишки та клавіатури. Більше того, морфи мають вміти реагувати на дії користувача, змінюючи свій вигляд та місце розсташування — це, власне і є анімація.

### Події мишки

Після натиску кнопки мишки, Morphic надсилає кожному морфу під вказівником мишки повідомлення *handlesMouseDown*:. Якщо морф відповідає true, тоді Morphic миттєво надсилає йому повідомлення *mouseDown*:; він також надсилає *mouseUp*:, коли юзер відпускає кнопку. Якщо всі морфи відповіли *false*, то Morphic ініціює “чіпляй-і-тягни”(drag-and-drop) операцію. Як ми далі будемо бачити, повідомлення *mouseDown*: і *mouseUp*: надсилаються з аргументом — об’єкт MouseEvent — який містить деталі події мишки.

Давайте розширимо *CrossMorph* так, щоб він зміг обробляти події мишки. Спочатку впевнимося, що усі об’єкти класу *CrossMorph* відповідають *true* на повідомлення *handlesMouseDown*:.

Додайте наступний метод до *CrossMorph*:

**CrossMorph >> handlesMouseDown: anEvent**

**^ true**

Нехай ми хочемо при натисканні лівої кнопки мишки змінити колір хреста на червоний, а після натиску правої - на жовтий. Це можна реалізувати через метод *mouseDown*: наступним чином:

**CrossMorph >> mouseDown: anEvent**

**anEvent redButtonPressed**

**ifTrue: [ self color: Color red ]. "click"**

**anEvent yellowButtonPressed**

**ifTrue: [ self color: Color yellow ]. "action-click"**

**self changed**

Зауважте, що окрім зміни кольору, цей метод також повертає “*self changed*”. Це робиться для того, щоб упевнитись, що морф своєчасно надсилає *drawOn*:.

Також слід зауважити, що як тільки морф реагує на події мишки, ви вже не можете його пересувати. Натомість, вам треба використовувати його гало: мета клік на морф змусить гало з’явитися. Далі використайте коричневий обробник руху(малюнок 13.10) або чорний обробник(13.11) згори над морфом.

Аргумент *anEvent* методу *mouseDown*: являє собою об’єкт класу *MouseEvent*, який у свою чергу є підкласом *MorphicEvent*. *MouseEvent* визначає методи *redButtonPressed* і

*yellowButtonPressed*. Ви можете детально ознайомитись з цим класом, заглянувши до його документації.



**Малюнок 13.10**: Обробник руху.



**Малюнок 13.11**: Обробник “захоплення”.

### Події клавіатури

Щоб захопити події клавіатури, нам потрібно проробити три кроки.

1. Сфокусувати клавіатуру на певному морфі.Наприклад, ми можемо зробити це, коли над морфом знаходиться вказівник мишки.
2. Обробити подію клавіатури за допомогою методу *handleKeystroke*:. Цей метод надсилається морфу, на якому наведений фокус клавіатури, коли користувач натискає клавішу.
3. Зняти фокус клавіатури з морфа, якщо курсор більше не знаходиться над морфом.

Давайте розширимо CrossMorph таким чином, щоб він реагував на натискання клавіш. Спочатку, нам треба організувати повідомлення, коли курсор над морфом. Це відбувається, коли морф відповідає *true* на повідомлення *handlesMouseOver*:

Оголосимо, що CrossMorph реагуватиме, коли він знаходеться під вказівником мишки.

**CrossMorph >> handlesMouseOver: anEvent**

**^true**

Це повідомлення еквівалентне до handlesMouseDown: для позиції мишки. Коли вказівник мишки заходить або покидає територію морфа, то надсилаються повідомлення *mouseEnter*: і *mouseLeave*:.

Визначимо два методи так, щоб CrossMorph захоплював і звільняв фокус кавіатури, а третій медот, власне, обробляв натисканя клавіш.

**CrossMorph >> mouseEnter: anEvent**

**anEvent hand newKeyboardFocus: self**

**CrossMorph >> mouseLeave: anEvent**

**anEvent hand newKeyboardFocus: nil**

**CrossMorph >> handleKeystroke: anEvent**

**| keyValue |**

**keyValue := anEvent keyValue.**

**keyValue = 30 "up arrow"**

**ifTrue: [self position: self position - (0 @ 1)].**

**keyValue = 31 "down arrow"**

**ifTrue: [self position: self position + (0 @ 1)].**

**keyValue = 29 "right arrow"**

**ifTrue: [self position: self position + (1 @ 0)].**

**keyValue = 28 "left arrow"**

**ifTrue: [self position: self position - (1 @ 0)]**

Ми написали цей метод таким чином, щоб мати змогу пересувати морф за допомогою клавіш зі стрілками. Зауважте, що коли мишка не знаходиться над морфом, то повідомлення *handleKeystroke*: не надсилається, отже морф не реагує на стрілки. Щоб побачити ключі значень, відкрийте вікно *Transcript*  і додайте *Transcript show: anEvent keyValue* до повідомлення *handleKeystroke*:.

Аргумент *anEvent* методу *handleKeystroke*: являє собою об’єкт класу *KeyboardEvent*, який у свою чергу є підкласом *MorphicEvent*. Загляньте у його документацію, щоб дізнатися більше про події клавіатури.

### Morphic animations

Morphic надає просту систему анімацій з двома основними методами: *step* надсилається до морфа у регулярні проміжки часу, а *stepTime* визначає час в млісекундах між двома кроками. Власне, *stepTime* визначає мінімальний час між кроками. Якщо ви встановите *stepTime* 1мс, то не дивуйтеся, що *Pharo* буде надто зайнятий кроками вашого морфу так часто. На додаток, *startStepping* включає “крокування”, а *stopStepping* його виключає.

Заставимо *CrossMorph* миготіти, визначивши методи наступним чином:

**CrossMorph >> stepTime**

**^ 100**

**CrossMorph >> step**

**(self color diff: Color black) < 0.1**

**ifTrue: [ self color: Color red ]**

**ifFalse: [ self color: self color darker ]**

Щоб побачити це, відкрийте інспектор морфу, використовуючи обробник налагодження (малюнок 13.12) на гало, введіть *self startStepping* у маленькому вікні пісочниці знизу і потім *“Do it”.*

Також можна модифікувати метод *handleKeystroke*: таким чином, щоб можна було викоистовувати клавіші + та - що включати і виключати крокування. Добавте наступний код до методу *handleKeystroke*: **:**

**keyValue = $+ asciiValue**

**ifTrue: [ self startStepping ].**

**keyValue = $- asciiValue**

**ifTrue: [ self stopStepping ]**

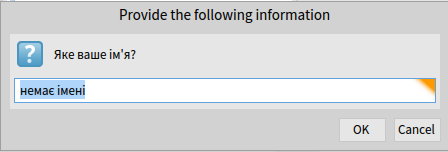


**Малюнок 13.12**: Обробник налагодження.

## 13.6 Взаємодіячі(Interactors)

Для того, щоб підказати користувачу ввід, клас *UIManager* надає велику кількість готових діалогових коробок. Наприклад, метод *request:initialAnswer*: повертає рядок, введений користувачем (Малюнок 13.13).

**UIManager default request: 'Яке ваше ім''я?' initialAnswer: 'немає імені'**



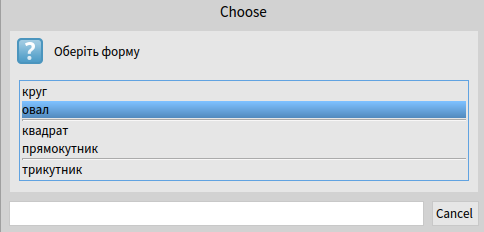
**Малюнок 13.13**: Вікно введення.

Щоб відобразити cпливаюче меню, використайте один з методів *chooseFrom*: (малюнок 13.14):

**UIManager default**

**chooseFrom: #('круг' 'овал' 'квадрат' 'прямокутник' 'трикутни')**

**lines: #(2 4) message: 'Оберіть форму'**



**Малюнок 13.14**: Спливаюче меню.

Огляньте клас *UIManager* і попробуйте інші запропоновані способи взаємодії.

## 13.7 “Перетягни та впусти”(Drag-and-drop)

Morphic також підтримує “перетягни та впусти”. Давайте розглянемо простий приклад з двома морфами, одержувач та морф, який кидають. Одержувач прийматиме морф лише в тому випадку, якщо він синій. Якщо ж його відторгнули, то морф, який кидали, вирішує, що йому робити.

Давайте перше визначимо морфа-одержувача:

**Morph subclass: #ReceiverMorph**

**instanceVariableNames: ''**

**classVariableNames: ''**

**package: 'PBE-Morphic'**

Тепер пропишемо метод ініціалізації:

**ReceiverMorph >> initialize**

**super initialize.**

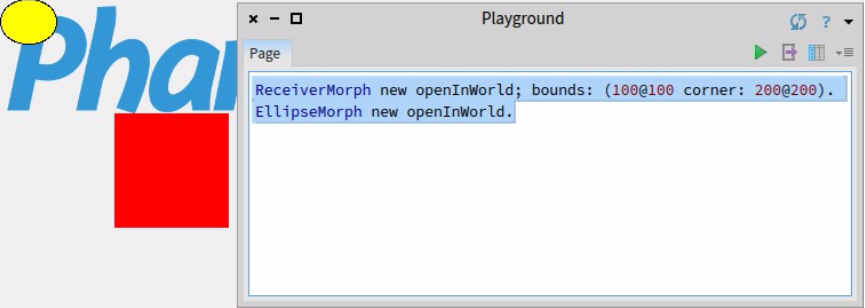
**color := Color red.**

**bounds := 0 @ 0 extent: 200 @ 200**

Як ми вирішуємо, чи морф-одержувач прийме, або ж відторгне кинутий морф? У загальному випадку, оба морфи мають вміти взаємодіяти

How do we decide if the receiver morph will accept or reject the dropped

morph? In general, both of the morphs will have to agree to the interaction.



**Малюнок 13.15**: *ReceiverMorph* та *EllipseMorph.*

Одержувач робить це, відповідаючи на *wantsDroppedMorph:event*:. Перший аргумент - морф, який кидають, а другий - подія мишки. Таким чином одержувач може, наприклад, побачити, чи були натисненні якісь клавіші на момент опускання. Опущений морф також надає можливість перевірити і бачити, чи його влаштовує морф, на якого його кидають, відповідаючи на повідомлення *wantsToBeDroppedInto*:.

Реалізація даного методу (у класі *Morph*) за замовчуванням повертає *true*.

**ReceiverMorph >> wantsDroppedMorph: aMorph event: anEvent**

**^ aMorph color = Color blue**

Що ж відбудеться з морфом, який кидають, якщо морф-одержувач не хоче його прийняти? Поведінка за замовчуванням - нічого не робити, тобто знаходитись зверху морфа-одержувача, але не взаємодіяти з ним. Більш інтуїтивно зрозумілою поведінкою буде повертати його на первинну позицію Цього можна домогтися, отримавши від одержувача ствердну відповідь на повідомлення *repelsMorph:event*:, коли він не хоче впущеного морфа:

**ReceiverMorph >> repelsMorph: aMorph event: anEvent**

**^ (self wantsDroppedMorph: aMorph event: anEvent) not**

Створюємо у пісочниці об’єкти *ReceiverMorph* та *EllipseMorph*:

**ReceiverMorph new openInWorld;**

**bounds: (100@100 corner: 200@200).**

**EllipseMorph new openInWorld.**

Попробуйте перетягнути та впустити жовтий *EllipseMorph* на одержувача. Його буде відторгнена і відіслано на його початкову позицію.

Щоб змінити таку поведінку, змінимо колір еліпса на синій(надіславши повідомлення *color: Color blue*; одразу після *new*). Синій морф має бути сприйнятий об’єктом класу *ReceiverMorph*.

Давайте створимо підклас класу *Morph*, назвемо його *DroppedMorph*, щоб мати змогу дещо більше експериментувати:

**Morph subclass: #DroppedMorph**

**instanceVariableNames: ''**

**classVariableNames: ''**

**package: 'PBE-Morphic'**

**DroppedMorph >> initialize**

**super initialize.**

**color := Color blue.**

**self position: 250 @ 100**

Тепер ми можемо задати, що кинутий морф має робити, якщо його відторгнуто одержувачем; в даному випадку він залишатиметься прикріпленим до вказівника мишки:

**DroppedMorph >> rejectDropMorphEvent: anEvent**

**| h |**

**h := anEvent hand.**

**WorldState addDeferredUIMessage: [ h grabMorph: self ].**

**anEvent wasHandled: true**

Надсилаючи повідомлення *hand* до *anEvent* дає відповідь *hand*, об’єкт класу *Hand-Morph*, що представляє вказівник мишки та все, що він тримає. Тут ми кажемо *World*, що *hand* повинна захоплювати *self*, відторгнутий морф.

Створимо два об’єкти класу *DroppedMorph*, і потім перетягнемо та впустимо їх на одержувача.

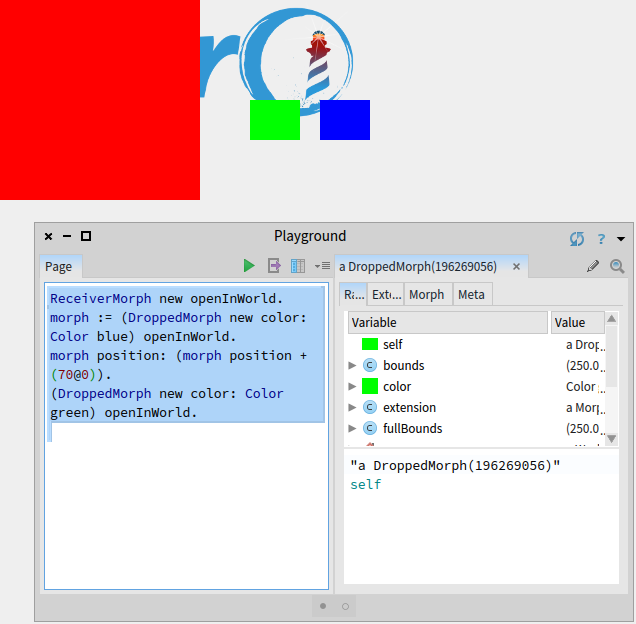
**ReceiverMorph new openInWorld.**

**morph := (DroppedMorph new color: Color blue) openInWorld.**

**morph position: (morph position + (70@0)).**

**(DroppedMorph new color: Color green) openInWorld.**

Зелений морф відтрогнений, як результат залишається приклеєний до вказівника мишки.



**Малюнок 13.16**: Створення *DroppedMorph* та *ReceiverMorph.*

## 13.8 Повноцінний приклад

Давайте розробимо морф, який кидає гральний кубик. Перший натиск - запускатиме почергове відображення значення усіх сторін кубика, а наступний натиск - зупинятиме анімацію. визначимо кубик як підклас *BorderedMorph* замість *Morph*, бо ми використовуватимемо краї.

**BorderedMorph subclass: #DieMorph**

**instanceVariableNames: 'faces dieValue isStopped'**

**classVariableNames: ''**

**package: 'PBE-Morphic**

Змінна *faces* записуватиме кількість крапок на кубику; ми будемо відображати до 9ти сторін! *dieValue* записує значення сторони, яка відображається на даний момент, і *isStopped* має значення *true*, якщо анімація призупинена. Щоб створити кубик, ми визначимо метод *faces: n* у *DieMorph class*, щоб створити кубик з *n* сторонами.

**DieMorph class >> faces: aNumber**

**^ self new faces: aNumber**



**Малюнок 13.17**: Кубик зі своїм морф-меню*.*

Визначимо метод *initialize* на стороні об’єкту звичним способом; пам’ятаємо, що *new* автоматично надсилає повідомлення *initialize* новоствореному об’єкту.

**DieMorph >> initialize**

**super initialize.**

**self extent: 50 @ 50.**

**self**

**useGradientFill;**

**borderWidth: 2;**

**useRoundedCorners.**

**self setBorderStyle: #complexRaised.**

**self fillStyle direction: self extent.**

**self color: Color green.**

**dieValue := 1.**

**faces := 6.**

**isStopped := false**

Ми використовуємо декілька методів *BorderedMorph* щоб надати кубику при появі гарного вигляду: грубий край з ефектом “підняття”, заокруглені кути, і градієнт кольору на видимій частині. Ми визначаємо об’єкту метод *faces*: для перевірки вхідних параметрів наступним чином:

**DieMorph >> faces: aNumber**

**"Вкажіть кількість сторін"**

**((aNumber isInteger and: [ aNumber > 0 ]) and: [ aNumber <= 9 ])**

**ifTrue: [ faces := aNumber ]**

Було би добре пригадати порядок, у якому повідомлення надсилаються, коли створюється кубик. Наприклад, якщо ми починаємо обраховувати *DieMorph faces: 9*:

* Метод класу *DieMorph class >> faces*: надсилає *new* до *DieMorph class*.
* Метод для *new* (успадкований *DieMorph class* від *Behavior*) створює новий об’єкт і надсилає його повідомленню *initialize*.
* Метод *initialize* у *DieMorph* задає *faces* початкове значення 6.
* *DieMorph class >> new* повертається до методу класу *DieMorph class >> faces*:, який надсилає повідомлення *faces*: 9 новому об’єкту.
* Метод об’єкта *DieMorph >> faces*: виконується, задаючи змінній *faces* значення 9.

Перед визначенням *drawOn*:, нам потрібно декілька методів, щоб розсташовувати крапки на стороні, яка відображається:

**DieMorph >> face1**

**^ {(0.5 @ 0.5)}**

**DieMorph >> face2**

**^{0.25@0.25 . 0.75@0.75}**

**DieMorph >> face3**

**^{0.25@0.25 . 0.75@0.75 . 0.5@0.5}**

**DieMorph >> face4**

**^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75}**

**DieMorph >> face5**

**^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.5@0.5}**

**DieMorph >> face6**

**^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 . 0.75@0.5}**

**DieMorph >> face7**

**^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 . 0.75@0.5 . 0.5@0.5}**

**DieMorph >> face8**

**^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 . 0.75@0.5 . 0.5@0.5 . 0.5@0.25}**

**DieMorph >> face9**

**^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 . 0.75@0.5 . 0.5@0.5 . 0.5@0.25 . 0.5@0.75}**

Ці методи визначають колекцію координат точок для кожної сторони. Координати знаходяться у квадраті розміром 1х1; нам лишень потрібно првильно вирахувати розсташування крапок.

Метод *drawOn*: робить дві речі: малює фон кубика через *super*, а потім малює крапки.

**DieMorph >> drawOn: aCanvas**

**super drawOn: aCanvas.**

**(self perform: ('face', dieValue asString) asSymbol)**

**do: [:aPoint | self drawDotOn: aCanvas at: aPoint]**

Друга частина цього методу використовує рефлексію *Pharo*. Малювання крапок здійснюється проходом по колекції, яку надає метод *faceX*, посилаючи повідомлення *drawDotOn:at:* для кожної координати. Щоб викликати коректний метод *faceX*, ми використовуємо метод *perform*:, який надсилає повідомлення, побудоване з рядка, *('face', dieValue asString) asSymbol*. Ви зустрічатимете таке використання *perform*: доволі часто.

**DieMorph >> drawDotOn: aCanvas at: aPoint**

**aCanvas**

**fillOval: (Rectangle**

**center: self position + (self extent \* aPoint)**

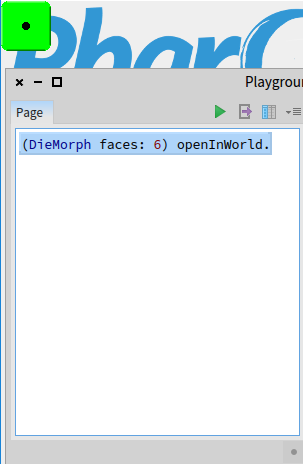
**extent: self extent / 6)**

**color: Color black**

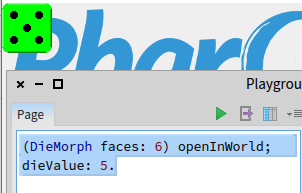
Оскільки коодринати нормовані до інтервалу [0:1], ми масштабуємо їх відповідно до розміру кубика: *self extent \* aPoint*.

Ми тепер можемо створити об’єкт кубика з пісочниці (малюнок 13.18):

**(DieMorph faces: 6) openInWorld.**



**Малюнок 13.18**: Новий кубик, створений після виконання “*(DieMorph faces: 6) openInWorld.”*



**Малюнок 13.19**: Результат виконання “*(DieMorph faces: 6) openInWorld; dieValue: 5.”*

Щоб змінити сторону, яка відображається, ми створюємо відповідний метод, щоб його можна було використовувати як *myDie dieValue: 5*:

**DieMorph >> dieValue: aNumber**

**((aNumber isInteger and: [ aNumber > 0 ]) and: [ aNumber <= faces ])**

**ifTrue: [**

**dieValue := aNumber.**

**self changed ]**

Тепер використаємо анімацію, щоб швидко показати усі сторони:

**DieMorph >> stepTime**

**^ 100**

**DieMorph >> step**

**isStopped ifFalse: [self dieValue: (1 to: faces) atRandom]**

Тепер кубик крутиться!

Щоб почати або призупинити анімацію натиском кнопки мишки, ми використовуватимемо попередні знання про події мишки. Спочатку, активуємо прийом подій мишки:

**DieMorph >> handlesMouseDown: anEvent**

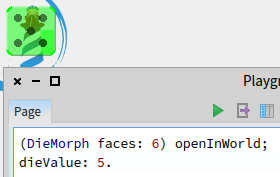
**^ true**

**DieMorph >> mouseDown: anEvent**

**anEvent redButtonPressed**

**ifTrue: [isStopped := isStopped not]**

Тепер кубик буде починати або переставати крутитися, коли ми на нього натискатимемо.



**Малюнок 13.20**: Кубик, з коефіцієнтом прозорості 0.5.

## 13.9 Більше про *canvas*

Метод *drawOn*: єдиним аргументом має об’єкт класу *Canvas*; полотно(canvas) - це територія, на якій морф відображає себе. Використовуючи графічні методи полотна, ви можете задати морфу такого вигляду, якого забажаєте. Якщо ви глянете на ієрархію наслідування класу *Canvas,* то побачите, що там є декілька варіантів. Варіантом за замовчуванням є *FormCanvas*, і ви знайдете більшість ключових графічних методів у *Canvas* та *FormCanvas*. Ці методи можуть малювати точки, лінії, полігони, прямокутники, еліпси, тексти та картинки, також крутити їх та масштабувати.

Також можливо використовувати інші видиполотна, наприклад, щоб отримати прозорі морфи, більше графічних методів, згалджування тощо. Щоб використовувати ці особливості, потрібно *AlphaBlendingCanvas* або *BalloonCanvas*. Але як ми можемо отримати таке полотно у методі *drawOn*:, якщо *drawOn*: отримує своїм аргументом об’єкт класу *FormCanvas*? На щастя, ми можемо перетворити один тип полотна у інший.

Щоб використати полотно з коефіцієнтом прозорості 0.5 у *DieMorph*, перевизначимо *drawOn*: наступним чином:

**DieMorph >> drawOn: aCanvas**

**| theCanvas |**

**theCanvas := aCanvas asAlphaBlendingCanvas: 0.5.**

**super drawOn: theCanvas.**

**(self perform: ('face', dieValue asString) asSymbol)**

**do: [:aPoint | self drawDotOn: theCanvas at: aPoint]**

Це все, що потрібно зробити!

## 13.10 Підсумок

*Morphic* - це графічний фреймворк, у якому можна динамічно компонувати графічні елементи.

* Можна перетворювати об’єкт у морф і відображати його на екрані, надсилаючи йому повідомлення *asMorph openInWorld.*
* Морфом можна керувати, здійснюючи мета-кліки на них і використовуючи обробники, що з’являються. (Орбробники мають вспливаючі підказки, які пояснюють, що вони роблять.)
* Можна компонувати морфи, вставляючи їх один у одного, перетягуючи та впускаючи їх чи надсилаючи повідомлення *addMorph*:.
* Можна створити підклас існуючого класу та перевизначити ключові методи, такі як *initialize* and *drawOn*:.
* Можна контролювати взаємодію морфа з мишкою та клавіатрою, перевизначивши такі методи як *handlesMouseDown*:, *handlesMouseOver*: тощо.
* Морф можна анімувати, перевизначивши методи *step* (що робити) and *stepTime* (кількість мілісекунд між кроками).