Obsługa zdarzeń w Dolphinie

Zdarzenia można generować i można zgłaszać chęć ich odbierania.

Object>>when: anEventSymbol **send:** aSelector **to:** anObject
```
    self when: anEventSymbol perform:
        (EventMessageSend receiver: anObject selector: aSelector)
```

Object>>when: anEventSymbol **send:** aSelector **to:** anObject **withArguments:** anArray

Object>>when: anEventSymbol **perform:** aMessageSend
```
    | events |
    (events := self getEvents) isNil
        ifTrue: [events := EventsCollection new. self setEvents: events].
    events addEvent: anEventSymbol message: aMessageSend.
```

Uwaga: jeśli podamy selektor nieistniejącej metody, to w momencie zajścia zdarzenia wykonywanie programu zakończy się z komunikatem:
<obiekt> dose not undestand <selektor>.

I generowanie zdarzeń:

```
Object>>trigger: anEventSymbol
    self events triggerEvent: anEventSymbol.

Object>>trigger: anEventSymbol with: aParameter
Object>>trigger: anEventSymbol with: aParameter
with: anotherParameter
Object>>trigger: anEventSymbol withArguments: args

EventsCollection>>triggerEvent: anEventSymbol
    | messages |
    messages := self at: anEventSymbol ifAbsent: [^nil].
    messages value
```

Model – View – Presenter (MVP)

- Sposób tworzenia GUI w Dolphinie.
- Oparty na komponentach.
- Elastyczny (łatwość ponownego wykorzystywania komponentów).
- Wykorzystany we wszystkich narzędziach Dolphina.
- Można tworzyć aplikacje w Dolphinie nie używające MVP.
- Każdy komponent (podstawowy lub złożony) jak również cała aplikacja składa się z trójki elementów:
  - Prezenter.
    Łączy widok z modelem. Akcje użytkownika są przekazywane przez widok do prezentera, a ten powoduje odpowiednie zmiany w modelu.
  - Widok.
    Okno bądź pojedynczy element graficzny. Służy do reprezentacji danych i pobierania poleceń użytkownika. Zna model oraz prezenter (zmienne obiektowe). Rejestruje swoje zainteresowanie zmianami stanu modelu (#when:send:to:).
  - Model.
    Zwykle jest to obiekt z poziomu dziedziny aplikacji. Nie wie nic o pozostałych elementach tej trójki! Potrafi generować zdarzenia (np. #valueChanged) używając metody #trigger. Uwaga: wszystkie dane pamiętane są w modelach (nawet wartości pól tekstowych).

Możliwe jest (w pewnym stopniu) wymienianie elementów trójek MVP.

Schemat tworzenia złożonego komponentu MVP

Uwaga: W Dolphinie model i prezenter tworzymy jako klasy, zaś widok (zwykle) tworzymy jako zasób.

Zwróćmy uwagę na inną kolejność niż w typowych środowiskach wizualnych, tam zaczyna się od widoku.

1. Tworzymy model.
   - może być podklasą klasy Model (ale nie jest to konieczne).
   - definiujemy wszystkie metody, które mają działać na modelu.
   - możemy zdefiniować metodę initialize (jeśli jesteśmy podklasą klasy Model, to będzie ona wywołana w new).
   - testujemy model (w Smalltalku jest to dość proste, mimo że na razie nie mamy jeszcze widoku ani prezentera).
   - model jest zobowiązany do informowania o zmianie swojej wartości zdarzeniem #valueChanged. Najprostszym sposobem osiągnięcia tego celu jest użycie podklas ValueModel. Uwaga: zwykle poszczególne elementy interfejsu są zainteresowane jedynie częściami całego modelu, dlatego zwykle nie cały model będzie podklasą ValueModel, lecz jedynie jego fragmenty.

2. Tworzymy Prezenter (1/2)

- Z powodów technicznych tworzymy prezentera już teraz (będzie potrzebny przy zapisywaniu naszego widoku przy wychodzeniu z ViewComposera).
- Tworzymy go jako podklasę klasy Shell
- (Ewentualnie definiujemy w nim tyle zmiennych obiektowych ile będzie składowych prezenterów).

3. Tworzymy View używając ViewComposera

- Draw|ShowToolbox.
- przeciągamy te widoki, których potrzebujemy do naszego widoku.
- umieszczamy je w odpowiednich miejscach, dopasowujemy rozmiary (np. Draw|Match|Widths).
- każdy element naszego widoku (z nim samym włącznie) ma listę aspektów, które można teraz modyfikować.
- Nadajemy wartość aspektowi name dla tych elementów widoku, do których będziemy się odwoływać.
- Zapisujemy utworzony widok. Wybieramy klasę prezentera i wpisujemy nazwę dla naszego widoku. Od tego momentu stworzony przez nas widok będzie tworzył zasób (razem z prezenterem) o nazwie: <prezenter>.<nazwa widoku>. Zasoby są dostępne przez Resource Browsera.

4. Tworzymy Prezenter (2/2)

Podajemy którego widoku ma używać. (Jeśli nie podamy, to będzie używany standardowy, pusty widok.) Robimy to definiując metodę _klasową_ #defaultView, tak by jako wynik dawała nazwę (String) naszego widoku (tę podaną przy zapisywaniu w VC).

- Określamy model używany przez prezentera. Można to zrobić na jeden z dwu sposobów:
  - na stałe: definiując metodę klasową defaultModel dającą obiekt będący modelem,
  - jednorazowo: używając do wyświetlenia okienka metody #showOn:
- Definiujemy te metody, których nazwy są podwiązane do akcji (np. jako aspekt command z PushButton) w widoku.
- Dołączamy prezentery do podwidoków, które dołączyliśmy do naszego widoku. Robimy to definiując metodę obiektową prezentera #createComponents. Ta metoda musi:
  - najpierw wywołać tę samą metodę z nadklasy. (W Shell nie jest definiowana, w CompositeComponent jest pusta, ale to oczywiście może się zmienić.)
  - używając metody #add: dodać odpowiednie prezentery. Wynikiem metody #add: jest dodawany prezenter. Dodawane prezentery muszą być zgodne ze stworzonym uprzednio widokiem, tzn.:
    - należy im nadać nazwy takie jak podaliśmy w podwidokach jako aspekt #name.
    - muszą być tych samych typów co podwidoki, które dodaliśmy.

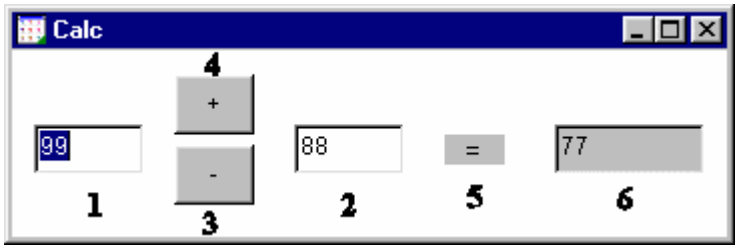
Przykład:
[<zmienna> :=] self add: <KlasaPrezentera> new name: <Nazwa>
Żeby móc potem odwoływać się do tych prezenterów można zapisać je na zmienne obiektowe tworzonego prezentera. Inne rozwiązanie polega na posługiwaniu się metodą:
Presenter>>presenterNamed: <nazwa>
dającą prezentera o nazwie podanej w add: (i w widoku).

- Definiujemy metodę #model: zwykle trzeba podłączyć do podprezenterów inne modele. Te podprezentery mają swoje domyślne modele (dokładnie w taki sam sposób jak nasz prezenter) lub nil. My (zwykle) chcemy, by były związane z fragmentami naszego modelu. Dlatego w metodzie #model:, która określa co jest naszym modelem, wysyłamy im komunikat #model: informujący je, kto jest ich modelem. Zwykle wykorzystujemy tu komunikat Object>>aspectValue: aSymbol, pozwalający widzieć jeden aspekt obiektu (coś co jest określone metodami aSymbol i aSymbol:) jako model. Metoda #model: musi:
  - wywołać metodę #model: z nadklasy. (Nie jest zdefiniowana w CompositePresenter ani w Shell, w Presenter zapamiętuje model i informuje widok (o ile jest), jaki ma model).
  - określić modele podprezenterów.
    □

<jeszcze createSchematicWiring>

Przykład prostego programu z MVP
(oparte na http://www.iandb.nildram.co.uk/)

WIDOK



MODEL

```
Model subclass: #CalcModel
    instanceVariableNames: 'field1 field2 result'
    classVariableNames: ''
     poolDictionaries: ''

add
result value: field1 + field2

initialize
  field1 := 99.
  field2 := 88.
  result := 77 asValue.
^self

subtract
  result value:  field1 - field2
```

PREZENTER

```
Shell subclass: #CalcShell
    instanceVariableNames: ''
    classVariableNames: ''
     poolDictionaries: ''

addFields
  model add.

createComponents

 super createComponents.

  self add: NumberPresenter new name: 'inputField1'.
  self add: NumberPresenter new name: 'inputField2'.
  self add: NumberPresenter new name: 'resultField'.

model: aCalcModel

  super model: aCalcModel.

  (self presenterNamed: 'inputField1') model: (aCalcModel
aspectValue: #field1).
  (self presenterNamed: 'inputField2') model: (aCalcModel
aspectValue: #field2).
  (self presenterNamed: 'resultField') model: aCalcModel
result.
```

subtractFields
  model subtract

defaultView (klasowa)
  ^'DefaultView'

Można w Prezenterze obsługiwać część akcji użytkownika:

onKeyPressed: aKeyEvent
onKeyReleased: aKeyEvent
onKeyTyped: aKeyEvent
onLeftButtonDoubleClicked: aMouseEvent
onLeftButtonPressed: aMouseEvent
onLeftButtonPressed: aMouseEvent „I analogiczne dla prawego”
onMouseMoved: aMouseEvent
onMoved: aMoveEvent „Dotyczy przesunięcia okna”
onResized: aSizeEvent „Dotyczy okna”

Event subclass: #KeyEvent
instanceVariableNames: ''
classVariableNames: 'AltKeyMask CtrlKeyMask ExtendedKeyMask PrevStateMask TransitionMask'
poolDictionaries: ''
Klasa zdarzeń obsługujących naciśnięcia/zwolnienia klawiszy. Ma metody obiektowe:

- code
- data
- isAlreadyDown
- isAltKeyDown
- isBeingReleased
- isExtended

```
PointEvent subclass: #MouseEvent
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

Klasa obsługująca zdarzenia związane z myszką. Ma metody obiektowe:

- x
- y
- isCtrlDown
- isLButtonDown,
- isMButtonDown
- isRButtonDown
- isShiftDown

Jest jeszcze kilka użytecznych klas zdarzeń (np. PaintEvent, SizeEvent).