Contents

Dlaczego Clojure?

Clojure jest językiem programowania, który pomaga nam sprawić aby programowanie było prostsze. Stara się nie kompilkować go obsługą stanu, mutowaniem danych, skomplikowanym dziedziczeniem itd. Polecam ogólnie wykład twórcy Clojure - Richa Hickeys "Simplicity matters".

- Funkcyjny ale pragmatyczny - funkcje są wartościami, niemutowalny stan, ale możliwe jest używanie zmiennych i pisanie z side-effectami itd.
- Rozszerzalny
  - w Clojure (tak jak w innych Lispach) kod wygląda tak jak dane - ma tą samą składnię
  - można go przetwarzać przed wykonaniem poprzez makra
  - daje to nieograniczone możliwości rozszerzenia składni, zaimplementowanie paradygmatów np obiektowość czy CSP
  - Uncle Bob o tym wspominał w kontekście Clojure jako języka przyszłości
- Wydzielenie stanu - stan aplikacji można kontrolować w jednym miejscu, nie zakłócając kodu
- Dobra integracja z kodem napisanym dla JVM
  - Łatwe korzystanie z bibliotek Javowych
  - Możliwość dołącznia Clojure do istniejących projektów (np jako konsola)
  - Biblioteki w Clojure z których można korzystać w Javie
- REPL interaktywne programowanie, czyli...

Interaktywne programowanie

Wchodzimy w bezpośrednią interakcję z komputerem, z programem który piszemy. To co zmieniamy wpływa na to co dzieje się na ekranie (trochę magii). Nie czekamy na to aż kompilator przetworzy kod i na uruchomienie programu od początku. Staje się to popularne, szczególnie w Javascripcie (live reload itd) ale często nie pozwala na zachowanie stanu całej aplikacji.

Pierwsze kroki

Jak uruchomić Clojure?

- upewniamy się, że mamy zainstalowaną Javę 1.6
- ściągamy ze strony http://clojure.org/downloads
- uruchamiamy przez `java -jar clojure-1.x.0.jar`
- pojawia się prompt readera

Otrzymujemy w ten sposób czyste środowisko pracy w Clojure, które - pracuje on metodą Read Eval Print w pętli czyli REPL - przypomina to Node.js (node), Python (python) czy Ruby (irb) - jest interaktywne, zapewnia szybki feed-back

Usprawnieniem jest Leiningen - narzędzie podobne do Apache Maven - pozwala na tworzenie i zarządzenie projektem w Clojure - zapewnia udoskonalony interfejs REPL z autouzupełnianiem :)

Leiningen ściągamy ze strony http://leiningen.org/.

Poznajemy Clojure

Celem prezentacji ma być stworzenie gry więc:

- tworzymy projekt gry (lein new breakout)
- odpalamy REPL (lein repl)

Jak wygląda kod w Clojure?

- Składa się z wyrażeń (przykłady)
  - liczby (nawet wymierne),
  - znaki,
  - ciągów znaków,
  - symbole (identyfikatory symbolizujące funkcje, nazwy klas, parametry czy globalne zmienne (Var)),
  - keywordy (stałe cachowane wyrażenia używane jako klucze, które dodatkowo są funkcjami :)),
  - wyrażeń regularnych,
  - nil (podobny do null ale jest wartością logiczną - fałszywy)
  - listy (jednokierunkowe + count - O(1)),
  - wektory (persistent vector map),
  - mapy (persistent map),
  - sety (oparte o persistent map)
- Reguły ewaluacji
  - symbole - do tego co identyfikują
  - listy
    - pierwszym elementem listy jest funkcja
    - pozostałe to argumenty funkcji
  - reszta wylicza się do samego siebie
- Deklaracja zmiennych
- Deklaracja funkcji

Tworzymy grę

Dodajemy bibliotekę, która będzie nam rysować grę.

- edytujemy plik project.clj
- dodajemy do :dependecies wektor [quil “2.2.5”]
- uruchamiamy ponownie REPL (lein repl)

Narysujmy coś na ekranie

- importujemy bibliotekę przez `(require '(quil [core :as q]))`
  - teraz do wszystkich funkcji odwołujemy się przez q/funkcja
- tworzymy funkcję rysującą

```
(defn draw []
  (q/background 200)
  (q/stroke 255 255 255)
  (q/stroke-weight 2)
  (q/fill 0 255 0)
  (q/rect 140 180 60 6))
```

- tworzymy funkcję ustawiająca parametry rysowania

```
(defn setup []
  (q/smooth)
  (q/frame-rate 30))
```

- wyświetlamy okienko

```
(q/defsketch breakout
  :title "Breakout"
  :setup setup
  :draw draw
  :size [320 200])
```

Struktura kodu

Zmienne są globalne, ale na szczęście są globalne tylko zakresie modułu. Jest to tzw. *namespace*. Dzięki niemu możemy podzielić kod funkcjonalnie i stosować te same nazwy funkcji w różnych kontekstach. Namespace deklarujemy przez funkcję *ns*.

Przenieśmy funkcje do odpowienich namespaceów

- otwórzmy plik `src/breakout/core.clj`
- dodajmy jako parametr ns `(:require [quil.core :as q])`
- usuńmy funkcje
- wstawmy nasze trzy funkcje

Stan i struktury danych

Jak uporządkować kod, wydzielając stan i tworząc rekordy - typy danych.

Sprawmy aby paletka się poruszała

- stwórzmy rekord paletki Board `(defrecord Board [x])`
  - rekord to tak na prawdę klasa Javowa
  - pola mogą mieć tzw type-hint i mogą być prymitywami
  - rekord ma wszystkie funkcje mapy
- zapiszmy stan paletki w atomie `(def board-state (atom (->Board 140)))`
  - atom - blokuje wykonanie w innych wątkach do czasu wykonania operacji zmiany stanu
- zmieniamy draw tak żeby korzystał z pozycji paletki `(q/rect (:x @board-state))`
  - @ - powoduje wydobycie wartości atomu
- ustawiamy nową pozycję: `(swap! board-state assoc :x 10)`

Uzupełnienie gry o inne elementy. Rysowanie elementów.

Dodajemy kolejne elementy. Aby uogólnić kod rysowania elementów zastosujemy protokół, który zaimplementujemy dla poszczególnych elementów.

Namespace elements

Tworzymy rekordy cegły i piłeczki.

- stwórzmy sobie namespace elements (plik src/breakout/elements.clj):

  ```
  (ns breakout.elements)
  ```

- przenieśmy deklarację Board do elements.clj
- dodajmy funkcję tworzącą board

  ```
  (defn board []
    (->Board 130))
  ```

- dodajmy `[breakout.elements :as elements]` do `(:require)` w core.clj
- zmieniamy `->Board` na `elements/board`

Powinno wszystko działać tak jak przed zmianami

- tworzymy w elements rekordy dla cegły i piłki oraz funkcje tworzące

```
(defrecord Brick [x y])
(defn brick [col row]
  (->Brick (* col 32) (* row 10)))

(defrecord Ball [x y])
(defn ball []
  (->Ball 160 100))
```

- tworzymy namespace graphics (plik src/breakout/graphics.clj) korzystający z quil.core i breakout.elements
- tworzymy protokół GraphicElement

  ```
  (defprotocol GraphicElement
    (draw [this]))
  ```

- tworzymy domyślną implementację która nic nie rysuje

```
(extent-type Object
  GraphicElement
  (draw [this] this))
```

- kasujemy stan paletki *board-state*
- dodajemy graphics do required
- dodajemy nowy stan

```
(def game-state (atom [(elements/brick 0 0)
                       (elements/ball)
                       (elements/board)]))
```

- zmieniamy draw

```
(defn draw []
  (q/background 100)
  (doseq [element @game-state]
    (graphics/draw element)))
```

  - *doseq* służy do wykonania operacji z side-effectem na sekwencji
  - sekwencja to abstrakcja służąca do poruszania się po kolekcji

- kopiujemy kod który rysuje paletkę i wstawiamy do implementacji protokołu

```
(extend-type breakout.elements.Board
  GraphicElement
  (draw [this]
    (comment Tu wstawiamy)))
```

Dodajmy rysowanie cegły i piłki

- implementujemy rysowanie cegły (wykorzystamy deconstructing parametrów)

```
(extend-type breakout.elements.Brick
  GraphicElement
  (draw [{:keys [x y]}]
    (q/stroke 255 255 0)
    (q/stroke-weight 2)
    (q/fill 255 0 0)
    (q/rect x y 32 10)))
```

- implementujemy rysowanie piłki

```
(extend-type breakout.elements.Ball
  GraphicElement
  (draw [{:keys [x y]}]
    (q/stroke 255 255 255)
    (q/stroke-weight 1)
    (q/fill 255 255 255)
    (q/ellipse (- x 4) (- y 4) 8 8)))
```

Teraz wszystko powinno być widoczne.

Dodajmy jeszcze dodatkowy element, który będzie tłem dla gry, czyli Playfield.

- nowy element z funkcją tworzącą

```
(defrecord Playfield [x1 y1 x2 y2])
(defn playfield [& elements]
  (cons (->Playfield 0 0 320 200) elements))
```

- skorzystajmy z tej funkcji w core

Animacja

W grze wszystko się rusza, dlatego musimy kontrolować elementy i badać kolizje między nimi.

Kontrola

Dodajmy kolejny protokół. Tym razem nazywa się Control i będzie odpowiadał za zmiany stanu gry wynikające z różnych przyczyn (czasu, ruchu myszki, startu gry).

- tworzymy nowy namespace control.clj a w nim protokół

```
(defprotocol Control
  (start [this])
  (mouse [this position])
  (time [this frame))
```

- implementację domyślną

```
(extend-type Object
Control
  (start [this] this)
  (mouse [this position])
  (time [this frame] this))
```

- użyjmy tego protokołu przy rysowaniu kolejnej klatki

```
(defn update [game]
  (->> game
       (map #(control/time % (q/frame-count)))))

(defn draw []
  (q/background 100)
  (swap! game-state update)
  (doseq [element @game-state]
    (graphics/draw element)))
```

Spróbujmy przesuwać piłkę z daną prędkością.

- dodajmy odpowiednie parametry do piłki i zmieńmy funkcję tworzącą

```
(defrecord Ball [x y speed delta-x delta-y])

(defn ball
  ([] (ball 2))
  ([speed] (->Ball 160 100 speed 0 0)))
```

- dodajmy funkcję która będzie uaktualniać pozycje piłki

```
(defn update-ball [{:keys [x y delta-x delta-y] :as ball}]
  (-> ball
    (assoc :x (+ x delta-x))
    (assoc :y (+ y delta-y))))
```

- zaimplementujmy protokół Control dla piłki

```
(extend-type breakout.elements.Ball
  Control
  (start [this] this)
  (mouse [this position])
  (time [this frame] (breakout.elements/update-ball this)))
```

Teraz będziemy przesuwać paletkę

- dodajmy obsługę przesuwania myszki w defsketch

```
(q/defsketch
  ...
  :mouse-moved mouse-moved)
```

- i funkcję zawiadamiająca o pozycji myszki

```
(defn mouse-moved []
  (let [publisher (partial map #(control/mouse % {:x (q/mouse-x) :y (q/mouse-y)}))]
    (swap! game-state publisher)))
```

- teraz dodajmy obsługę poruszania myszki do paletki implementując protokół

```
(extend-type breakout.elements.Board
  Control
  (start [this] this)
  (mouse [this position] (assoc this :x (:x position)))
  (time [this frame] this))
```

- możemy jeszcze dodać żeby paletka nie znikała z pola
- i żeby nie było widac kursora (w setup (q/no-cursor))

Dodajemy obsługę startu gry.

- start uaktywniamy klawiszem myszki

```
(q/defsketch
  ...
  :mouse-pressed mouse-pressed))
```

- dodajmy jeszcze funkcję, która ustawia nam inicjalizuje stan gry

```
(defn create-game
  (elements/playfield
    (elements/brick 0 0)
    (elements/ball)
    (elements/board)))
```

- i funkcję zawiadamiającą o starcie

```
(defn mouse-pressed []
  (let [publisher (partial map control/start)]
    (reset! game-state (-> (create-game)
                           (publisher)))))
```

- implementujemy start dla piłki

```
(start [{speed :speed :as this}] (-> this
                                     (assoc :delta-x (- speed))
                                     (assoc :delta-y (- speed))))
```

- użyjmy jej w deklaracji game-state

Kolizje

Załóżmy, że będziemy sprawdzać kolizje piłek z pozostałymi elementami. Funkcja sprawdzająca kolizję będzie zwracać nam stan po kolizji (co ostało się na polu, a co zostało usunięte). Na koniec funkcja będzie składać z tych informacji stan gry.

- stwórzmy sobie namespace collision
- potrzebujemy funkcję odróżniającą piłki (ball?) w elements

```
(defn ball? [o]
  (instance? Ball o))
```

- funkcja kolizji będzie zwracać mapę z kluczami :set, :replaced. Wartościami kluczy będzie wektor lub lista. Musimy wyodrębnić same wyniki :set i :replaced. Najlepiej zrobić to osobną funkcją.

```
(defn- only [k c]
  (->> c
       (map k)
       (filter identity)
       (flatten)
       (set)))
```

  - korzystamy tutaj z makra ->>
  - jest to czytelniejsze potokowe przetwarzanie danych

- funkcja kolizji

```
(defn collide [ball other]
  { :set [ball other] })
```

- funkcja collision-check:

```
(defn collision-check [game]
  (let [{balls true others false} (group-by elements/ball? game)
        output (for [ball balls
                     other others]
                 (collide ball other))
        all-set (only :set output)
        to-remove (only :replaced output)]
    (remove to-remove all-set)))
```

Prosta implementacja kolizji

Mamy tutaj bardzo prostą funkcję kolizji, która zwraca tylko obiekty kolizji, tak żeby nie zniknęły. Dodajmy kolizję piłki z polem gry. Stwórzmy multimetodę - dispatcher, który na podstawie rezultatu funkcji dispatchującej podejmuje decyzję którą implementację użyć.

- zmieniamy implementację

```
(defmulti collide (fn [x y] [(class x)] [(class y)]))

(defmethod collide :default
  [x y]
  { :set [x y] })
```

- dodajemy funkcję która będzie odbijać piłkę

```
(defn bounce-ball [{:keys [speed] :as ball} direction]
  (case direction
    :left (assoc ball :delta-x speed)
    :right (assoc ball :delta-x (- speed))
    :top (assoc ball :delta-y speed)
    :bottom (assoc ball :delta-y (- speed)))0
```

- dodajemy metodę odbicia od krawędzi pola gry

```
(defmethod collide
  [breakout.elements.Ball breakout.elements.Playfield]
  [{x :x y :y :as ball} playfield]
  (let [new-ball (-> ball
                     (elements/bounce-ball (cond (<= x (:x1 playfield)) :left
                                                 (>= x (:x2 playfield)) :right))
                     (elements/bounce-ball (cond (<= y (:y1 playfield)) :top)))]
    (if (= ball new-ball)
      {:set [playfield ball]}
      {:replaced [ball] :set [playfield new-ball]})))
```

Piłeczka powinna odbijać się już od ścian.

Dodajmy teraz odbijanie się od paletki. Aby wiedzieć że piłka jest w polu paletki lub cegły musimy znać ich rozmiary. Dotychczas były one na sztywno wpisane w funkcjach rysujących. Przenieśmy je w jedno miejsce, do struktur paletki i cegły. - dodajmy :w i :h do struktur - stwórzmy funkcję ktora będzie określać czy coś jest wewnątrz struktury z :x, :y, :w i :h.

```
(defn- inside [a b]
  (let [{a-x :x a-y :y w :w h :h} a
        {b-x :x b-y :y} b]
    (and (< a-x b-x (+ a-x w))
         (< a-y b-y (+ a-y h)))))
```

- dodać metodę do kolizji z paletką

```
(defmethod collide
  [breakout.elements.Ball breakout.elements.Board]
  [ball board]
  (if (inside board ball)
    {:replaced [ball] :set [(elements/bounce-ball ball :bottom) board]}
    {:set [ball board]}))
```

Zbijanie cegieł

```
(defmethod collide
  [breakout.elements.Ball breakout.elements.Brick]
  [ball brick]
  (if (inside brick ball)
    (let [new-ball (elements/bounce-ball ball :top)]
      (if (= new-ball ball)
        {:set [ball]}
        {:set [new-ball] :replaced [ball]}))
    {:set [brick ball]}))
```