Programación Funcional

Mayo, 2017 - Semana 2

Agenda

- Ámbitos de las declaraciones y modelo de ejecución
- Tipos complejos de datos (records y types)
- El problema de extensión (multimétodos y protocolos)
- Patrones de diseño
- Tests
- Recursos

Ámbito de las declaraciones y modelo de ejecución

- Modelo de sustitución para la aplicación de procedimientos
- Scope Estático (*lexical*)

Ámbito de las declaraciones y modelo de ejecución - Modelo de sustitución

(defn **square** [x] (* x x))

(defn **sum-of-squares** [x y] (+ (**square** x) (**square** y)))

(defn **f** [a] (**sum-of-squares** (+ a 1) (* a 2)))

(**f** 5)

Orden Aplicativo:
- Evaluar cada parametro
- Evaluar el cuerpo de la función, sustituyendo cada instancia del argumento por su valor.
- Repetir hasta llegar a una expresión primitiva

Ámbito de las declaraciones y modelo de ejecución - Modelo de sustitución (cont.)

(sum-of-squares (+ a 1) (* a 2))

Se reemplaza el parámetro a por su valor, 5:

(sum-of-squares (+ **5** 1) (* **5** 2))

Se resuelve cada operación y se reemplaza la expresión por su resultado, obteniendo así nuevas expresiones con valores primitivos para resolver:

(sum-of-squares **(+ 5 1) (* 5 2)**)

(sum-of-squares **6 10**)

(+ **(square 6) (square 10)**)

(+ 36 100)

136

(square 6) => (* 6 6) => 36

(square 10) => (* 10 10) => 100

Ámbito de las declaraciones y modelo de ejecución - Modelo de sustitución (cont.)

Otra forma, **Orden Normal**:
- Evaluar el cuerpo de la función, sustituyendo cada argumento por su expresión.
- Una vez que no pueda sustituirse más, evaluar las expresiones primitivas

```
(f 5)

(sum-of-squares (+ 5 1) (* 5 2))

(+ (square (+ 5 1)) (square (* 5 2)) )

(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))

(+ (* 6 6) (* 10 10))

(+ 36 100)

136
```

Ámbito de las declaraciones y modelo de ejecución - Modelo de sustitución (cont.)

Orden Aplicativo:

- Evalúa cada argumento exactamente una vez

Orden Normal:

- El programador controla cuántas veces se evalúa cada argumento
  - **Nunca**
  - 1 vez
  - **N veces**

Si no hay mutación o efectos colaterales, ambos dan el mismo resultado

Scope Estático (*lexical*)

; root binding (namespace)
(def MAX-CONNECTIONS 10)

; unbound
(def RABBITMQ_CONNECTION)

Scope Estático (*lexical*)

```
(let [x 10
      y 20]
  (println “x, y:” x “,” y))
```

La expresión **let** tiene **visibilidad dentro** del **bloque** donde es **definida**, fuera de ese ámbito, las variables x e y no existen.

Scope Estático (*lexical*)

```
(defn average-pets []
  (let [user-data (vals users)
        pet-counts (map :number-pets user-data)
        total (apply + pet-counts)]
    (/ total (count users))))
```

En este caso sirve para **escribir código más claro** sin tener que definir funciones para cada expresión.

Tipos complejos de datos - Types

```
(deftype BinaryTree [left right] ...)
```

Introduce un nuevo tipo de dato estructurado y algunas operaciones:

- Construcción

  ```
  (def tree (new BinaryTree 1 2)) => #'user/tree
  tree => #user.BinaryTree{:left 1 :right 2}
  ```

- Acceso

  ```
  (. tree left) => 1
  ```

- Consulta de tipo

  ```
  (type tree) => user.BinaryTree
  ```

Tipos complejos de datos - Records

```
(defrecord Pet [name species] ...)
```

Define un tipo, al igual que deftype, pero además:

- Define igualdad y hash.

```
(= (new Pet "Tweety" "Canary") (new Pet "Silvester" "Cat")) => false
```

- Define los mismos accesores que tienen los diccionarios.

```
(:species (new Pet "Tweety" "Canary")) => "Canary"
```

- Conforma a las interfaces de Java `java.util.Map` y `java.io.Serializable`

  Pero: `(.clear (new Pet "Tweety" "Canary"))`

```
;; Inmutable => UnsupportedOperationException
```

Todas estas son definiciones típicas para tipos del dominio de negocio.

El problema de extensión - Protocolos

Parecido a interfaces, sirven para definir contratos...

```
(defprotocol Fly
    (fly [this]))

(defrecord Bird [name species]
    Fly
    (fly [this]
        (str (:name this) " is a flying " (:species this))))

(fly (new Bird "Tweety" "canary"))
=> "Tweety is a flying canary"


(defrecord Cat [name])

(fly (new Cat "Silvester"))
```

!> IllegalArgumentException No implementation of **method: :fly** of **protocol: #'user/Fly** found for **class: user.Cat** clojure.core/-cache-protocol-fn (core_deftype.clj:544)

El problema de extensión - Protocolos

Hay una diferencia con interfaces:

```
;; Planes invented here

(extend-protocol Fly
    Cat
    (fly [this]
      (str (:name this) " is on a plane")))

(fly (new Cat "Silvester"))
=> "Silvester is on a plane"
```

Es posible agregar otros casos sin cambiar codigo existente.

El problema de extensión

Tenemos un conjunto de operaciones y un conjunto de casos a los que se deben poder aplicar todas las operaciones.

Soluciones:

	Operaciones	**Casos**
Objetos	Métodos	Clases
Funcional	Funciones	*Switch?*

El problema de extensión

Problema:

Cuantos cambios hay que hacer para agregar un caso o una operación?

	Operaciones	**Casos**
Objetos	**Modificar** N métodos	Agregar 1 clase
Funcional	Agregar 1 función	**Modificar** N funciones

El problema de extensión

Intento hacer lo mismo con dos argumentos:

```
class A {
    void print(A a) {
      System.out.println("A/A")
    }

    void print(B b) {
      System.out.println("A/B")
    }
}
```

```
class B extends A {
    void print(A a) {
      System.out.println("B/A")
    }

    void print(B b) {
      System.out.println("B/B")
    }
}
```

El problema de extensión

```
A a = new A();
A b = new B();

a.print(a); => "A/A"
a.print(b); => "A/A"
b.print(a); => "B/A"
b.print(b); => "B/A"
```

El polimorfismo de Java aplica únicamente al objeto destinatario.

El problema de extensión - ad-hoc

```
(defn ad-hoc-type-namer [thing]
    (condp = (type thing)
           java.lang.String "string"
           clojure.lang.PersistentVector "vector"))

(ad-hoc-type-namer "") => "string"
(ad-hoc-type-namer []) => "vector"
(ad-hoc-type-namer {}) ;; No hay caso para mapa => excepcion
```

No se puede agregar nuevos "casos" a la función ad-hoc-type-namer sin modificar el código de la función. (Closed dispatch)

El problema de extensión - Case

```
(def example-user {:login "rob" :referrer "mint.com" :salary 100000})

(defn fee-amount [percentage user]
    (with-precision 16 :rounding HALF_EVEN
    (* 0.01M percentage (:salary user))))

(defn affiliate-fee [user]
    (case (:referrer user)
    "google.com" (fee-amount 0.01M user)
    "mint.com"   (fee-amount 0.03M user)
    (fee-amount 0.02M user)))
```

Sigue siendo un ejemplo de closed dispatch, sobre un atributo en lugar de la clase.

El problema de extensión - Multimétodos

```
(defmulti multimethod-type-namer (fn [thing] (type thing)))

(defmethod multimethod-type-namer  java.lang.String [thing]
  "string")
(defmethod multimethod-type-namer  clojure.lang.PersistentVector [thing]
  "vector")

(multimethod-type-namer "") => "string"
(multimethod-type-namer []) => "vector"
(multimethod-type-namer {}) ;; No hay caso para mapa => excepcion
```

La última llamada al multimétodo falla porque no existe un caso para los mapas.

El problema de extensión - Multimétodos

```
(def example-user {:login "rob" :referrer "mint.com" :salary 100000})

(defmulti affiliate-fee (fn [user] (:referrer user)))

(defmethod affiliate-fee "mint.com" [user]
    (fee-amount 0.03M user))
(defmethod affiliate-fee "google.com" [user]
    (fee-amount 0.01M user))
(defmethod affiliate-fee :default [user]
    (fee-amount 0.02M user))

(affiliate-fee example-user) => (fee-amount 0.03M user))
```

Podemos ver a defmulti como la definición de la función polimórfica que especifica qué función de dispatch utilizará para delegar cada ejecución.

El problema de extensión - Multimétodos

```
(defmulti beat (fn [drum stick]  [(class drum) (class stick)]))

; Cada uno describe el sonido apropiado
(defmethod beat [Snare-drum Wooden-drumstick] [drum stick] ...)
(defmethod beat [Snare-drum :default] [drum stick] ...)
(defmethod beat [:default Wooden-drumstick] [drum stick] ...)
(defmethod beat [:default Brush] [drum stick] ...)
(defmethod beat :default [drum stick] ...)
```

Sigue siendo necesario escribir tantos métodos como hay casos.
Los métodos pueden distribuirse sin restricciones de que operación implementan o que casos cubren. Entonces, se pueden distribuir de la forma que mejor explique la solución.

Patrones de diseño - Command

Problema: Encapsular una acción para que pueda realizarse en un momento posterior.

Patrones de diseño - Command - Clojure

```
(let [login-command (fn []
                      (db/login "django" "unCh@1ned"))
      logout-command (fn []
                       (db/logout "django"))]

  (login-command))
```

En Clojure el patrón de diseño command se traduce en invocar a una función.

Patrones de diseño - Command - Java

```
interface Command {
  void execute();
}
```

```
public class LoginCommand implements Command {

  private String user;
  private String password;

   public LoginCommand(String user, String password) {
     this.user = user;
     this.password = password;
   }

   @Override
   public void execute() {
     DB.login(user, password);
   }

}
```

Patrones de diseño - Command - Java

```
public class LogoutCommand implements Command {

  private String user;

  public LogoutCommand(String user) {
    this.user = user;
  }

  @Override
  public void execute() {
    DB.logout(user);
  }

}
```

```
(new LoginCommand("django", "unCh@1ned")).execute();

(new LogoutCommand("django")).execute();
```

Patrones de diseño - Strategy

Problema: Encapsular acciones para poder seleccionar un miembro de una familia de funciones.

Patrones de diseño - Strategy - Java

```java
class SubsComparator implements Comparator<User> {

  @Override
  public int compare(User u1, User u2) {
    if (u1.isSubscription() == u2.isSubscription()) {
      return u1.getName().compareTo(u2.getName());
    } else if (u1.isSubscription()) {
      return -1;
    } else {
      return 1;
    }
  }

}
```

```java
class ReverseSubsComparator implements Comparator<User> {

  @Override
  public int compare(User u1, User u2) {
    if (u1.isSubscription() == u2.isSubscription()) {
      return u2.getName().compareTo(u1.getName());
    } else if (u1.isSubscription()) {
      return -1;
    } else {
      return 1;
    }
  }

}
```

Patrones de diseño - Strategy - Java

```
// forward sort
Collections.sort(users, new SubsComparator());

// reverse sort
Collections.sort(users, new ReverseSubsComparator());
```

En Java el patrón de diseño Strategy se traduce a una **clase** que encapsula cierta lógica que se aplica por **composición**.

Patrones de diseño - Strategy - Clojure

```
(sort (comparator
      (fn [u1 u2]
       (cond
        (= (:subscription u1)
           (:subscription u2)) (neg? (compare (:name u1)
                                              (:name u2)))
           (:subscription u1) true
           :else false))) users)
```

En Clojure el patrón de diseño Strategy se traduce a una **función** que encapsula cierta lógica que se aplica por **composición**.

Patrones de diseño - State - Java

```
public enum UserState {
  SUBSCRIPTION(Integer.MAX_VALUE),
  NO_SUBSCRIPTION(10);

  private int newsLimit;

  UserState(int newsLimit)  {
    this.newsLimit = newsLimit;
  }
  public int getNewsLimit() {
    return newsLimit;
  }
}
```

```
public class User  {
  private int money = 0;
  private UserState state = UserState.NO_SUBSCRIPTION;
  private final static int SUBSCRIPTION_COST = 30;

  public List<News> newsFeed() {
    return DB.getNews(state.getNewsLimit());
  }

  public void pay(int money) {
    this.money += money;
    if (state == UserState.NO_SUBSCRIPTION && this.money >=
SUBSCRIPTION_COST)  {
      // buy subscription
      state = UserState.SUBSCRIPTION;
      this.money -= SUBSCRIPTION_COST;
    }}
}
```

Patrones de diseño - State - Java

```
User user = new User(); // create default user

user.newsFeed(); // show him top 10 news

user.pay(10); // balance changed, not enough for subs

user.newsFeed(); // still top 10

user.pay(25); // balance enough to apply subscription

user.newsFeed(); // show him all news
```

Patrones de diseño - State - Clojure

```
(defmulti news-feed :user-state)
(defmethod news-feed :subscription [user] (db/news-feed))
(defmethod news-feed :no-subscription [user]  (take 10 (db/news-feed)))

(def ^:const SUBSCRIPTION_COST 30)

(defn pay [user amount]
  (swap! user update-in [:balance] + amount)
  (when (and (>= (:balance @user) SUBSCRIPTION_COST)
             (= :no-subscription (:user-state @user)))
    (swap! user assoc :user-state :subscription)
    (swap! user update-in [:balance] - SUBSCRIPTION_COST)))
```

Patrones de diseño - State - Clojure

```
(def user (atom {:name "Jackie Brown" :balance 0 :user-state :no-subscription}))

(news-feed @user) ;; top 10

(pay user 10)

(news-feed @user) ;; top 10

(pay user 25)

(news-feed @user) ;; all news
```

Patrones de diseño - Singleton

Problema: Existe una única instancia de cierta entidad, por lo que queremos que solo exista una única representación de ella.

Patrones de diseño - Singleton - Clojure

```
(defn load-config [config-file]
  ;; process config file and return map with configuratios
 {:bg-style "black" :font-style "Arial"})

(def ui-config (load-config "ui.config"))
```

En clojure, un **Singleton**, no es más que una **definición global**.

Patrones de diseño - Singleton - Java

```
public final class UIConfiguration {

 public static final UIConfiguration INSTANCE = new
UIConfiguration("ui.config");

 private String backgroundStyle;
 private String fontStyle;

 private UIConfiguration(String configFile) {
   loadConfig(configFile);
 }

 private static void loadConfig(String file) {
    // process file and fill UI properties
      INSTANCE.backgroundStyle = "black";
   INSTANCE.fontStyle = "Arial";
 }

 public String getBackgroundStyle() {
   return backgroundStyle;
 }

 public String getFontStyle() {
   return fontStyle;
 }

}
```

Patrones de diseño - Visitor

Problema: Separar una estructura de datos de los algoritmos que operan sobre ella.

Patrones de diseño - Visitor - Clojure

```
(defmulti export
  (fn [item format] [(:type item) format]))


;; Message
{:type :message :content "Say what again!"}

;; Activity
{:type :activity :content "Quoting Ezekiel 25:17"}

;; Formats
:pdf, :xml
```

```
(defmethod export [:activity :pdf] [item format]
(exporter/activity->pdf item))

(defmethod export [:activity :xml] [item format]
(exporter/activity->xml item))

(defmethod export [:message :pdf] [item format]
(exporter/message->pdf item))

(defmethod export [:message :xml] [item format]
(exporter/message->xml item))

(defmethod export :default [item format]
(throw (IllegalArgumentException. "not supported")))
```

Patrones de diseño - Visitor - Java

```
abstract class Format { }

class PDF extends Format { }
class XML extends Format { }

public abstract class Item {
  void export(Format format) {
    throw new UnknownFormatException(f);
  }

  abstract void export(PDF pdf);
  abstract void export(XML xml);

}
```

```
class Message extends Item {
  void export(PDF pdf) {
    PDFExporter.export(this);
  }

  void export(XML xml) {
    XMLExporter.export(this);
  }
}

class Activity extends Item {
  void export(PDF pdf) {
    PDFExporter.export(this);
  }

  void export(XML xml) {
    XMLExporter.export(this);
  }
}
```

Patrones de diseño - Visitor - Java

Si queremos agregar un **nuevo formato** debemos **modificar** la clase **Item**.

Patrones de diseño - Visitor - Java

```
public interface Visitor {
  void visit(Activity a);
  void visit(Message m);
}

public class PDFVisitor implements Visitor {

  public void visit(Activity a) {
    PDFExporter.export(a);
  }

  public void visit(Message m) {
    PDFExporter.export(m);
  }

}
```

```
public abstract class Item {
  abstract void accept(Visitor v);
}

class Message extends Item {
  void accept(Visitor v) {
    v.visit(this);
  }
}

class Activity extends Item {
  void accept(Visitor v) {
    v.visit(this);
  }
}
```

Patrones de diseño - Visitor - Java

El patrón de diseño **Visitor** propone resolver este problema aplicando una técnica conocida como **double dispatch.** La idea es utilizar de manera polimórfica el parámetro de un método que se resuelve polimórficamente.

Tests

Clojure tiene un **framework** de **unit testing** incluído en el lenguaje:

```
(use 'clojure.test)

(deftest test-math-operations
(is (= 4 (+ 2 2)))
(is (instance? Long 256))
(is (.startsWith "abcde" "ab")))

(run-tests)
```

Tests (Cont.)

Escribamos un test para la función de fibonacci:

```
(use 'clojure.test)

(defn fibonacci [x]
      (if (<= x 1)
            1
            (+ (fibonacci (- x 1)) (fibonacci (- x 2)))))
```

Tests (Cont.)

```
(deftest test-fibonacci
        (is (= 1 (fibonacci 0)))
        (is (= 1 (fibonacci 1)))
        (is (= 2 (fibonacci 2)))
        (is (= 3 (fibonacci 3)))
        (is (= 5 (fibonacci 4)))
        (is (= 8 (fibonacci 5)))
)

(run-tests)
```

Recursos

- Repl
- Try Clojure
- Clojure in Action
- Structure and Interpretation of Computer Programs
- Functional Thinking
- Clojure Design Patterns
- Ejercicios
- Más Ejercicios