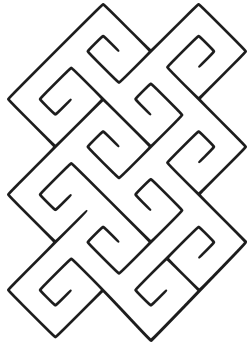
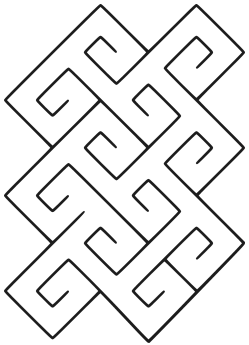


Patrones de diseño



Agenda

- Definiciones
- Patrones creacionales
- Ejercicios

Agenda

- Definiciones
- Patrones creacionales
- Ejercicios

Definición

"Un patrón describe un problema que ocurre una y otra vez, y luego describe el núcleo de la solución a dicho problema, de forma tal que se puede usar esta solución un millón de veces, sin repetir jamás la forma de aplicarla"

Christopher Alexander

> "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."
>
> Christopher Alexander

O más simplemente…

> "Un patrón es una solución a un problema, dado un contexto."
>
> GOF

¿Por qué patrones?

- Conocidos
- Convencionales
- Documentados
- Simples
- Comprobados

Facilitan comunicación

Fáciles de detectar

Describe el contexto

Reducen complejidad

Se sabe que funcionan

Agenda

- Definiciones
- Patrones creacionales
- Ejercicios

Patrones creacionales

- Factory Method
- Builder
- Singleton

Factory Method

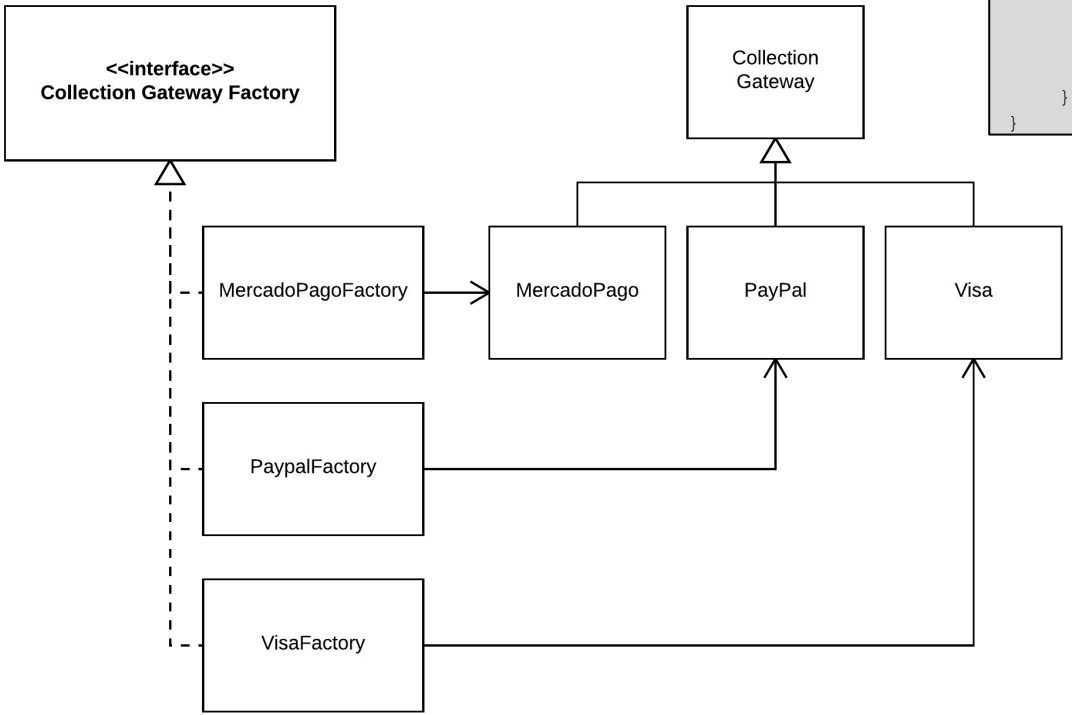
La instanciación concreta de un objeto respeta una interfaz o contrato, pero existen múltiples implementaciones, pero los factores que rigen la necesidad de cada una, surge en tiempo de ejecución

- Define una interfaz para crear un objeto
- El objeto a instanciar se define en runtime
- Permite postergar la elección de la implementación a usar

- De gran utilidad para
  - Frameworks
  - Plug-in Arqs

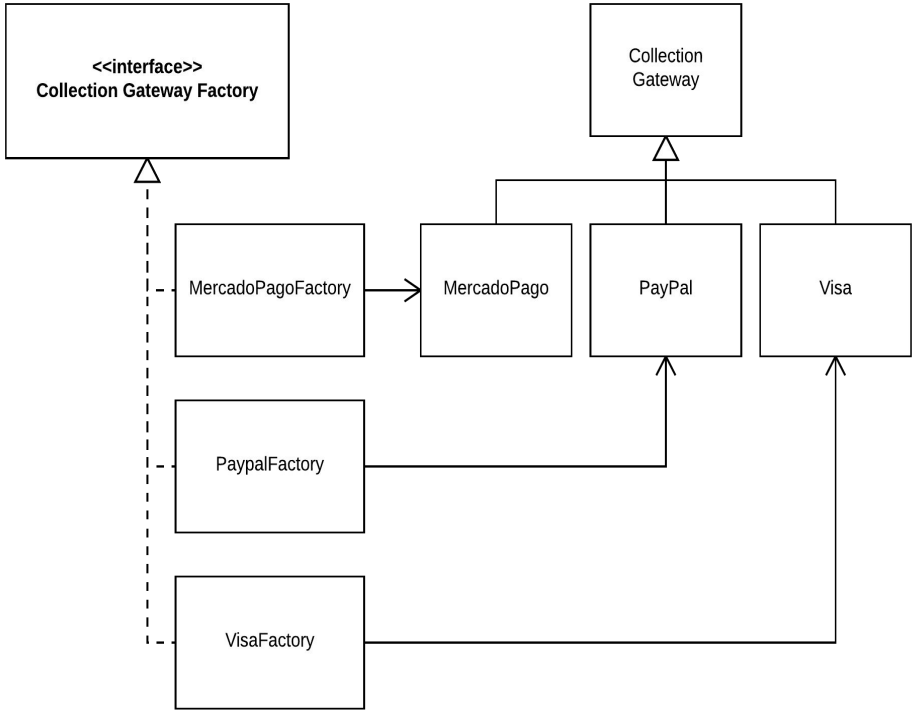
Factory Method

```
public class Collector {

    private CollectionGatewayFactory gatewayFactory;

    public void setGatewayFactory(
                        CollectionGatewayFactory gatewayFactory)
{
         this.gatewayFactory = gatewayFactory;
    }

    public void collect(...) {
        // ...
        CollectionGateway gateway = this.gatewayFactory.create();
        gateway.collect(...);
        //...
    }
}
```



¿Dudas?

Factory Method




```
(defn collection-gateway-collect-mercadopago [col] (
  str "Collecting mercadopago with " col))

(defn collection-gateway-collect-paypal [col] (
  str "Collecting paypal with " col))

(defn collection-gateway-collect-visa [col] (
  str "Collecting visa with " col))

(defmulti collection-gateway-collect (fn [type] type))
(defmethod collection-gateway-collect "mercadopago" [type]
collection-gateway-collect-mercadopago)
(defmethod collection-gateway-collect "paypal" [type]
collection-gateway-collect-paypal)
(defmethod collection-gateway-collect "visa" [type]
collection-gateway-collect-visa)

((collection-gateway-collect "mercadopago") '(1 2 3))
((collection-gateway-collect "paypal") '(4 5))
((collection-gateway-collect "visa") '(6 7))
```

¿Dudas?

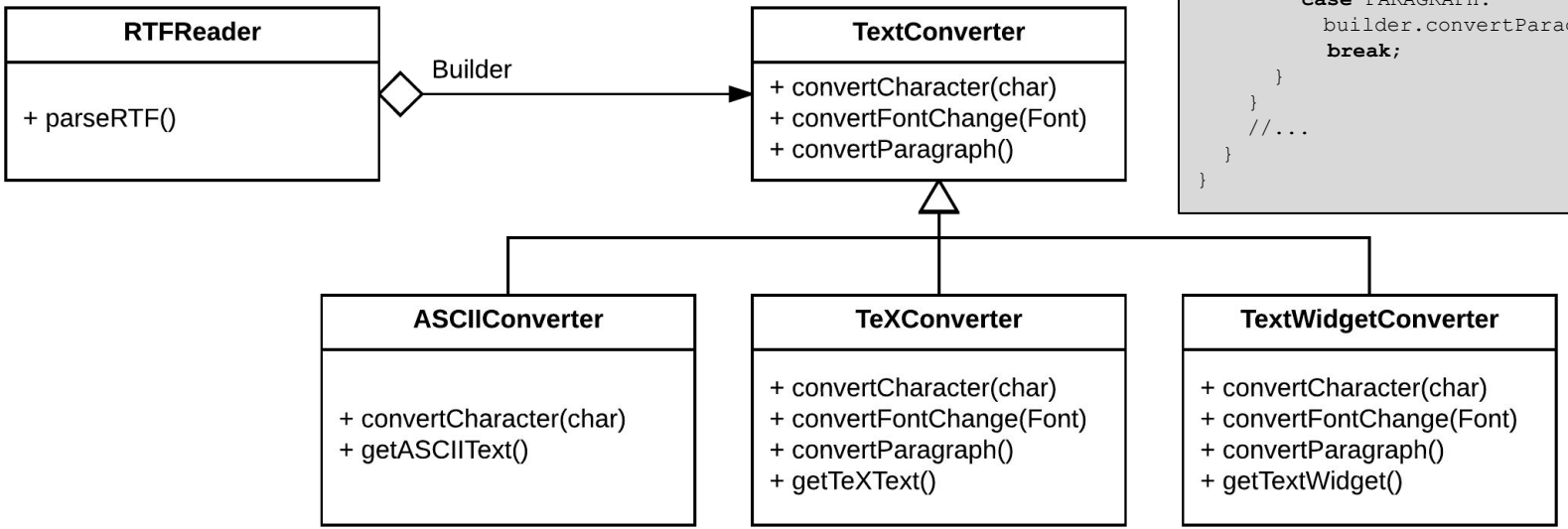
Builder

La construcción de un determinado objeto es compleja, requiere de múltiples pasos, dependencias o relaciones.

Se quiere poder crear diferentes representaciones del objeto creado, o los objetos a crear pueden crecer con el tiempo.

- Desacopla el proceso de construcción del objeto construido
- Agrega claridad al código cliente
- Segrega la construcción en un proceso paso a paso
- Facilita la extensión, como nuevas representaciones

Builder

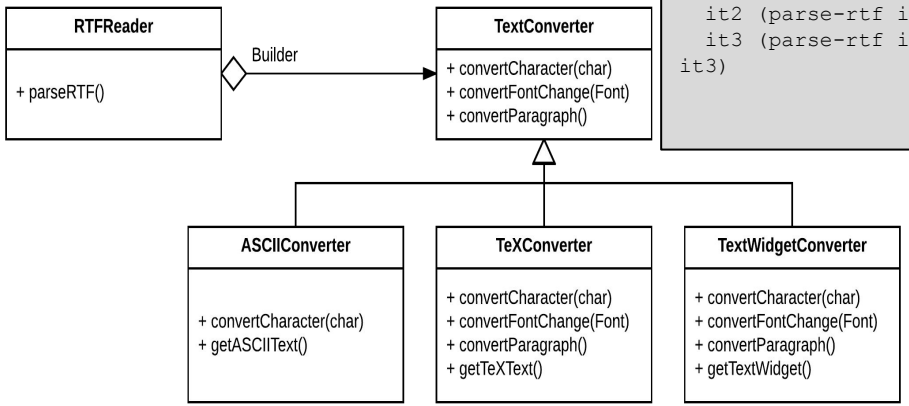



```
public class RTFReader {

  public void parseRTF(Builder builder) {
    // ...
    while (t = getNextToken()) {
      switch (t.Type) {
        case CHAR:
          builder.convertChar(t.data);
          break;
        case FONT:
          builder.convertFontChange(t.data);
          break;
        case PARAGRAPH:
          builder.convertParagraph(t.data);
          break;
      }
    }
    //...
  }
}
```

¿Dudas?

Builder

```
(def token-1 {:type "char" :data "a"})
(def token-2 {:type "font" :data "blue"})
(def token-3 {:type "paragraph" :data "p"})

(defn convert-char [actual data]
  {:chars (str (:chars actual) "c") :fonts (:fonts actual) :paragraphs (:paragraphs actual)})

(defn convert-font [actual data]
  {:chars (:chars actual) :fonts (str (:fonts actual) "f") :paragraphs (:paragraphs actual)})

(defn convert-paragraph [actual data]
  {:chars (:chars actual) :fonts (:fonts actual) :paragraphs (str (:paragraphs actual) "p")})

(defmulti parse-rtf-tex (fn [actual token] (:type token)))
(defmethod parse-rtf-tex "char" [actual token] (convert-char actual (:data token)))
(defmethod parse-rtf-tex "font" [actual token] (convert-font actual (:data token)))
(defmethod parse-rtf-tex "paragraph" [actual token] (convert-paragraph actual (:data token)))

(defmulti parse-rtf (fn [actual token] (:type actual)))
(defmethod parse-rtf-tex "tex" [actual token] (parse-rtf-tex actual token))

(def initial {:type "tex" :chars "" :fonts "" :paragraphs ""})

(let [
  it1 (parse-rtf initial token-1)
  it2 (parse-rtf it1 token-2)
  it3 (parse-rtf it2 token-3)]
it3)
```



¿Dudas?

Singleton

Necesito una instancia única de una clase que se necesita acceder desde diversos puntos.

- Asegura la existencia de una única instancia
- Facilita y asegura un único punto de acceso
- Es *statefull*
- Permite ser reemplazada por Mocks

- Usos más frecuentes
  - Configuraciones
  - Caches
  - Pools

Singleton

```
public class LocksDatabase {

  private Map<String, Lock> registeredLocks;

  private LocksDatabase() {
       this.registeredLocks = new TreeMap<String, Lock>();
  }

  public Lock getLock(String macAddress) {
       return this.registeredLocks.get(macAddress);
  }
  // ...

  private static final INSTANCE = new LocksDatabase();

  public static LocksDatabase getInstance() {
       return LocksDatabase.INSTANCE;
  }

}
```

```
public class LocksDatabase {

  private Map<String, Lock> registeredLocks;

  private LocksDatabase() {
    this.registeredLocks = new TreeMap<String, Lock>();
  }

  public Lock getLock(String macAddress) {
    return this.registeredLocks.get(macAddress);
  }
  // ...

  private static final INSTANCE;

  public static LocksDatabase getInstance() {
    if (LocksDatabase.INSTANCE == null) {
      LocksDatabase.INSTANCE = new LocksDatabase();
    }
    return LocksDatabase.INSTANCE;
  }

}
```

Singleton

```
(def mac-1 "mac-1")
(def mac-2 "mac-2")
(def mac-3 "mac-3")

(defn lock-acquire-mac [mac] (println (str "Aquiring lock for " mac)))

(def locks-database [{:mac mac-1 :lock lock-acquire-mac}
                     {:mac mac-2 :lock lock-acquire-mac}
                     {:mac mac-3 :lock lock-acquire-mac}])

(let [mac-item mac-1
      lock-item (first (filter (fn [lock-item] (= (:mac lock-item)   mac-item)) locks-database))]
     ((:lock  lock-item) mac-item)
)
```

```
public class LocksDatabase {

  private Map<String, Lock> registeredLocks;

  private LocksDatabase() {
       this.registeredLocks = new TreeMap<String, Lock>();
  }

  public Lock getLock(String macAddress) {
       return this.registeredLocks.get(macAddress);
  }
  // ...

  private static final INSTANCE = new LocksDatabase();

  public static LocksDatabase getInstance() {
       return LocksDatabase.INSTANCE;
  }

}
```

Agenda

- Definiciones
- Patrones creacionales
- Ejercicios

Bibliografía

- Design Patterns CD - Gamma, Helm, Johnson, Vlissides