Aplicación para el modelado de diagramas de clases y generación de código en lenguaje Ruby

elaborado por :

César Esteban Márquez Meneses, [cesare.marquezm@uqvirtual.edu.co](mailto:cesare.marquezm@uqvirtual.edu.co), 3213398080

Brian David Giraldo Paéz, [briand.giraldop@uqvirtual.edu.co](mailto:briand.giraldop@uqvirtual.edu.co), 3106947999

| *Prototipado funcional realizado con Eugenia Epsilon y Eclipse* |
| --- |

# índice

[**índice**](#_v9t4uof5cne6) **2**

[**1 Introducción**](#_xlxxd6sz9nvy) **3**

[1.1 Alcance](#_96jqvblmsewk) 3

[1.2 Palabras clave](#_oxuanh1kown5) 3

[1.3 Documentos relacionados](#_56kkt0uq91yq) 4

[**2 Enunciado o definición del problema**](#_6107f9wlfm6h) **4**

[**3 Dominio del Metamodelo**](#_godt4cdwmrj0) **4**

[**4 Metamodelo**](#_n8v8a6ajrrn2) **4**

[**5 Sintaxis Abstracta**](#_s6wt57wn3th8) **4**

[5.1 Diagrama de clases de la sintaxis abstracta](#_3oc34jq1j2an) 4

[5.2 Código fuente eugenia de la sintaxis abstracta](#_v21dqak230qx) 4

[5.3 Ejemplo de una producción de la sintaxis abstracta](#_wsmj01877fg0) 6

[5.4 Video de la sintaxis abstracta](#_br4gn6qamuqh) 6

[**6 Sintaxis Concreta**](#_ngzp0sfjuq8u) **6**

[6.1 Diagrama de clases de la sintaxis concreta](#_9l5tahaz4zjc) 7

[6.2 Código fuente eugenia de la sintaxis concreta](#_jp5dhuddq2pr) 7

[6.3 Ejemplo de una producción de la sintaxis concreta](#_mb9qagh85e75) 7

[6.4 Ejemplo de un diagrama del metamodelo (notación) relativo a la producción anterior](#_66xdceqbjuz4) 7

[6.5 Video de la sintaxis concreta](#_k3kj89l5lb6x) 8

[**7 Transformación de modelo a modelo (Concreto a Abstracto)**](#_mlnx6ykcv291) **8**

[**8 Transformación de modelo a texto (Abstracto a Reporte)**](#_2dsnk7gbo8n5) **8**

# 1 Introducción

El propósito de este de este documento es narrar el proceso mediante el cuál se construye una aplicación capaz interpretar un metamodelo y transformar su contenido al lenguaje de programación Ruby, siento su resultado un proyecto capaz de ser ejecutado en un entorno de trabajo para este lenguaje. El metamodelo que se usa es un diagrama de clases UML.

Un diagrama de clases es una herramienta para comunicar el diseño de un programa que se creó para orientar objetos y que permite modelar relaciones entre diferentes entidades. En UML, una clase se representa con un rectángulo que posee tres divisiones, nombre de la clase, atributos que tiene y mensajes que entiende. Para la construcción del editor visual, se utiliza la herramienta Eugenia, que hace parte de Epsilon (Eclipse), esta es una herramienta que simplifica el desarrollo de editores de modelos gráficos basados en GMF generando automáticamente los modelos .gmfgraph, .gmftool y .gmfmap necesarios para el editor GMF a partir de un único metamodelo Ecore anotado.

## 1.1 Alcance

Este documento muestra la implementación del diseño, análisis y construcción de una aplicación para el modelado de diagramas de clase y su respectiva transformación a código del lenguaje Ruby, haciendo uso de la herramienta Eugenia, Epsilon.

## 1.2 Palabras clave

* Eugenia Epsilon
* Elipse
* GMF
* ECore
* Diagrama de clases
* Relaciones
* Sintaxis concreta
* Sintaxis Abstracta
* Lenguaje Ruby
* Metamodelos

## 1.3 Documentos relacionados

* Repositorio Git del proyecto: [***Git***](https://github.com/Cemarquez/RM-Ruby)

# 2 Enunciado o definición del problema

La Ingeniería dirigida por modelos (MDE) es un paradigma de ingeniería de software, el cual se centra en la creación y explotación de modelos de dominio, representaciones abstractas de los conocimientos y actividades que rigen un dominio de aplicación particular, más que en conceptos informáticos o algoritmos.

De acuerdo con lo anterior, es pertinente resaltar el valor de los modelos para la ingeniería y desarrollo de software, además al momento de codificar lo anteriormente modelado es una tarea redundante, repetitiva y que requiere un tiempo proporcional al tamaño del modelo.

Se identificó que se requiere automatizar la codificación típica de un modelo de clases en el lenguaje Ruby, como se menciona anteriormente es una tarea larga y repetitiva, puesto que en un proyecto de un nivel alto de complejidad, con un diagrama de clases extenso, la tarea de codificar lo representado por modelos, demanda de un tiempo proporcional al tamaño del mismo, en el cual se realiza codificación redundante y repetitiva.

# 3 Dominio del Metamodelo

Los diagramas de clases son uno de los tipos de diagramas más útiles en UML, ya que trazan claramente la estructura de un sistema concreto al modelar sus clases, atributos, operaciones y relaciones entre objetos, en este caso con el lenguaje unificado de modelado UML es el lenguaje para modelos de sistemas de software más conocido y utilizado en la actualidad, es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema.

En el desarrollo del proyecto es necesario crear un metamodelo que representa los diagramas de clases, es decir, una abstracción que se utiliza para describir diagramas de clases, esto con la finalidad de utilizar la ingeniería dirigida por modelos para desarrollar una herramienta que permita convertir los modelos creados con el metamodelo a otros modelos y a codigo en lenguaje Ruby.

# 4 Metamodelo

En el contexto del modelado, un metamodelo es un modelo cuyas instancias son en sí mismas modelos. Como indica el prefijo meta, un metamodelo es una abstracción que se utiliza para describir modelos. Metamodelado se refiere en términos generales al diseño de metamodelos.

Los metamodelos pueden adoptar diversas formas y utilizarse en diversos campos. En informática, y más específicamente en ingeniería de software, la noción de metamodelo es fundamental para la ingeniería impulsada por modelos, un conjunto de prácticas que tienen en común la creación y uso de modelos para conceptualizar un dominio de aplicación. Este uso también se extiende a la ingeniería de sistemas.

# 5 Sintaxis Abstracta

La sintaxis abstracta es una representación del metamodelo construido por el usuario, pero desde una vista tal, que el sistema internamente pueda interpretar los elementos que están dentro de este y trabajar con ellos. El metamodelo de la sintaxis abstracta varía en ciertos aspectos del metamodelo concreto, ya que se omiten o se agregan algunas relaciones que no son de importancia para el usuario y/o el desarrollador.

## 5.1 Diagrama de clases de la sintaxis abstracta

El siguiente es el diagrama de clases (metamodelo) usado para la construcción de la sintaxis abstracta, este diagrama fue construido usando la herramienta TD-MBUILD proporcionada por el docente del curso de ingeniería dirigida por modelo. El metamodelo fue validado mediante una producción realizada en la misma herramienta, para así pasar a desarrollarlo en Eugenia.

Se evidencia que este modelo posee algunas relaciones compartidas con el modelo de la sintaxis concreta, exceptuando por las relaciones de containment de MClass con MAssociation, MContainment y MInheritance, esto con el fin que desde el modelo abstracto al recorrer las clases, se pueda saber de manera más fácil las relaciones salientes y entrantes de cada clase. De la misma manera, hace necesario conocer todos los paquetes y las clases que pertenecen a cada uno de estos, y se tiene en cuenta el hecho de que un paquete puede contener otros paquetes.

| Diagrama de clases de la sintaxis abstracta |
| --- |
|  |

## 5.2 Código fuente eugenia de la sintaxis abstracta

El siguiente fragmento de código es la representación textual del metamodelo anteriormente descrito. Está hecho con Eclipse Emfatic, el cual es una sintaxis textual para metamodelos de EMF Core. En el código se plantean exactamente las mismas relaciones y nombre del metamodelo, para así mantener una consistencia.

| código fuente de la sintaxis abstracta |
| --- |
| @namespace(uri="abstract", prefix="abstract") @gmf package abstracts;  class ModelFactory{  attr String nombre;  attr String ruta;  val MPackage[\*] lstPackages;  ref MClass[\*] lstAllClass;  ref MPackage[\*] lstAllPackage; }  class MPackage{  attr String name;  attr String path;  val MClass[\*] lstMClass;  val MPackage[\*] lstMPackage; }  class MClassDiagram{  attr String name;  val MPackage[\*] lstMPackage;  val MClass[\*] lstMClass; }  class MClass{  attr String name;  attr String accessModifier;  attr String comments;  attr String path;  val MAssociation[\*] lstMAssoctiation;  val MInheritance[\*] lstMInheritance;  val MContainment[\*] lstMContainment;  val MAttribute[\*] lstAttributes;  val MFunction[\*] lstFunction; }  class MAssociation{  ref MClass source;  ref MClass target;  attr String relationshipType;  attr String multiplicitySource;  attr String multiplicityTarget;  attr String sourceRole;  attr String targetRole; } class MInheritance{  ref MClass source;  ref MClass target;   }  class MContainment{  ref MClass source;  ref MClass target;  attr String multiplicityTarget;  attr String sourceRole;  attr String targetRole; }  class MAttribute{  attr String name;  attr String defaultValue;  attr String comments;  attr boolean constant;  attr boolean removeToInit; }  class MFunction{  attr String name;  attr String accessModifier;   attr String semantics;  attr String comments;  attr String parameters; } |

## 

## 5.3 Ejemplo de una producción de la sintaxis abstracta

En la siguiente imagen se evidencia una pequeña producción, donde hay varios paquetes, y dentro de estos paquetes existen clases, que están unidas mediante diferentes tipos de relaciones, de la misma manera, estas clases poseen atributos y funciones.

| Ejemplo de una producción de la sintaxis abstracta |
| --- |
|  |

## 5.4 Video de la sintaxis abstracta

| Video que ilustra la sintaxis abstracta |
| --- |
|  |

# 6 Sintaxis Concreta

La sintaxis concreta es una representación del metamodelo que permite su construcción por parte del usuario. El metamodelo de la sintaxis concreta varía en ciertos aspectos del metamodelo abstracto, ya que se omiten o se agregan algunas relaciones que no son de importancia para el usuario y/o el desarrollador.

## 6.1 Diagrama de clases de la sintaxis concreta

El metamodelo diagrama de clases usado para la construcción de la sintaxis concreta, al igual que la sintaxis abstracta, fue construido usando la herramienta TD-MBUILD. El metamodelo fue validado mediante una producción realizada en la misma herramienta, para así pasar a desarrollarlo en Epsilon Eugenia.

Se evidencia que este modelo posee algunas diferencias notorias con el modelo de la sintaxis abstracta, ya que en comparación con la sintaxis abstracta en este metamodelo no se establece una relación de containment de MClass con las clases MPackage, MContainment, MAssociation y MInheritance, esto para simplificar el uso al usuario, ya que esa estructura es propia del sistema y de la sintaxis abstracta.

| diagrama de clases de la sintaxis concreta |
| --- |
|  |

## 6.2 Código fuente eugenia de la sintaxis concreta

El siguiente fragmento de código es la representación textual del metamodelo anteriormente descrito. Está hecho con Eclipse Emfatic, el cual es una sintaxis textual para metamodelos de EMF Core. En el código se plantean exactamente las mismas relaciones y nombre del metamodelo, para así mantener una consistencia.

| código fuente de la sintaxis concreta |
| --- |
| @namespace(uri="concrete", prefix="concrete") @gmf package concrete;   class ModelFactory{  attr String nombre;  attr String ruta;  val MClassDiagram[\*] lstMDiagrams; }  @gmf.diagram class MClassDiagram{  attr String name;  attr String description;  val MPackage[\*] lstMPackage;  val MClass[\*] lstMClass;  val MAssociation[\*] lstMAssoctiation;  val MInheritance[\*] lstMInheritance;  val MContainment[\*] lstMContainment; }  @gmf.node(label = "name", color ="255,240,237" ) class MPackage{  attr String name;  attr String path; }  @gmf.node(label="name",figure="rounded", label.layout="center", color = "255,216,208", tool.description="Añade una clase al diagrama") class MClass{  attr String name;  attr String accessModifier;  attr String comments;  attr String path;    @gmf.compartment(layout="list", collapsible="false")  val MAttribute[\*] lstMAttribute;  @gmf.compartment(layout="list", collapsible="false")  val MFunction[\*] lstMFunction;  }  @gmf.node(label="name", figure="rectangle",label.icon="false", label.pattern="{0}", color = "255,216,208",tool.description="Añade un atributo a una clase") class MAttribute{  attr String name;  attr String defaultValue;  attr String comments;  attr boolean constant;  attr boolean removeToInit;   } @gmf.link(source = "source", target = "target", style = "solid", width = "1", color="255,0,0") class MAssociation{  ref MClass source;  ref MClass target;  attr String relationshipType;  attr String multiplicitySource;  attr String multiplicityTarget;  attr String sourceRole;  attr String targetRole;  attr boolean bidirectional; }  @gmf.link(source = "source", target = "target", style = "solid", width = "1", target.decoration="closedarrow", color="255,0,0") class MInheritance{  ref MClass source;  ref MClass target;   }  @gmf.link(source = "source", target = "target", style = "solid", width = "1", target.decoration="filledrhomb", color="255,0,0") class MContainment{  ref MClass source;  ref MClass target;  attr String multiplicityTarget;  attr String sourceRole;  attr String targetRole; }  @gmf.node(label="name,parameters", figure="rectangle",label.icon="false", label.pattern="{0}({1})", color = "255,216,208", tool.description="Añade una función a una clase") class MFunction{  attr String name;  attr String accessModifier;   attr String semantics;  attr String comments;  attr String parameters;   } |

## 6.3 Ejemplo de una producción de la sintaxis concreta

En la siguiente imagen se evidencia una producción de la sintaxis concreta, donde a diferencia de la producción abstracta, en esta los paquetes no contienen ningún elemento, además se ve que las relaciones están por fuera de sus clases y son contenidas por MClassDiagram, también se evidencia el correcto funcionamiento según el metamodelo y codificación para esta sintaxis concreta.

| ejemplo de una producción de la sintaxis concreta |
| --- |
|  |

## 6.4 Ejemplo de un diagrama del metamodelo (notación) relativo a la producción anterior

En la siguiente imagen se evidencia el diagrama de clases de la producción anterior, en el cual podemos ver las cuatro relaciones que existen, dos de herencia, una asociación y una de containment, además se ven los paquetes creados y las cinco clases con sus respectivos atributos y métodos.

| ejemplo de un diagrama del metamodelo |
| --- |
|  |

## 

## 

## 6.5 Video de la sintaxis concreta

| video que ilustra la sintaxis abstracta |
| --- |
|  |

# 7 Transformación de modelo a modelo (Concreto a Abstracto)

En la transformación modelo a modelo se pasa de la vista del usuario (concreto) a la vista del desarrollador (abstracto) para así convertir el nuevo modelo a código.

Para las transformaciones se crea un plugin de eclipse, mediante el cual se accionan los botones que harán el trabajo deseado. Para la construcción de la transformación Model To Model se creó una clase llamada TransformationM2M.java que contiene todos los métodos necesarios para esta tarea, tales como:

* transformarM2M
* crearPaquete
* crearClase
* crearAtributos
* crearFunciones
* crearAssociation
* crearHerencia
* crearContainment
* buscarPaquete
* buscarClase

En el anexo se evidencia sólo una porción del código debido a su extensión, para verlo completo y conocer su documentación acceda al siguiente link:

Clase de la creación de interfaz y plugin de eclipse: [***Link***](https://github.com/Cemarquez/RM-Ruby/blob/master/workspace/workspaceEpsilon/diagrama.view/src/diagrama/view/views/Generador.java)

Clase de transformación Model To Model: [***Link***](https://github.com/Cemarquez/RM-Ruby/blob/master/workspace/workspaceEpsilon/diagrama.view/src/diagrama/model/TransformationM2M.java)

| Código que realiza la transformación concreto a abstracto |
| --- |
| /\*\*  \* Método principal, desde el cual se llama a los demás  \* @return  \*/  public String transformarM2M() {  String mensaje = "Se ha realziado la transformación M2M";    for(concrete.MClassDiagram dConcreta : modelFactoryConcreta.getLstMDiagrams()) {  //crear los paquetes  for(concrete.MPackage p : dConcreta.getLstMPackage()) {  crearPaquete(p);  }    for(concrete.MClass c : dConcreta.getLstMClass()) {  crearClase(c);  }    for(concrete.MAssociation r : dConcreta.getLstMAssoctiation() ) {  crearAssociation(r);  }    for(concrete.MInheritance r : dConcreta.getLstMInheritance() ) { // crearAssociation(r);  crearHerencia(r);  }    for(concrete.MContainment c : dConcreta.getLstMContainment()) {  crearContainment(c);  }    }    return mensaje;  }    /\*\*  \* Método que verifica la existencia de una clase, crea una copia de tipo abstracto y la guarda  \* @param c clase concreta  \*/  private void crearClase(concrete.MClass c) {  String path = c.getPath();  abstracts.MPackage p = buscarPaqueteParent(path);  abstracts.MClass cl = buscarClase(path, c.getName(), p);  if(cl==null) {  abstracts.MClass newClass = AbstractsFactory.eINSTANCE.createMClass();  newClass.setPath(c.getPath());  newClass.setName(c.getName());  newClass.setComments(c.getComments());  newClass.setAccessModifier(c.getAccessModifier());  newClass.getLstFunction().addAll(crearFunciones(c));  newClass.getLstAttributes().addAll(crearAtributos(c));  p.getLstMClass().add(newClass);  modelFactoryAbstracta.getLstAllClass().add(newClass);  }    } |

## 

| video que ilustra la transformación |
| --- |
|  |

# 8 Transformación de modelo a texto (Abstracto a Reporte)

En la transformación modelo a texto se pasa de la vista del usuario desarrollador al producto final, que en este caso es el código escrito en lenguaje Ruby.

Para las transformaciones se crea un plugin de eclipse, mediante el cual se accionan los botones que harán el trabajo deseado. Para la construcción de la transformación Model To Text se creó una clase llamada TransformationM2T.java que contiene todos los métodos necesarios para esta tarea, tales como:

* transformarM2T
* crearPaquete
* crearClase
* crearAtributos
* crearFunciones
* crearConstructor
* crearConstructorVacio
* replaceLast
* crearHerencia
* crearContainment
* guardarArchivo

En el anexo se evidencia sólo una porción del código debido a su extensión, para verlo completo y conocer su documentación acceda al siguiente link:

Clase de la creación de interfaz y plugin de eclipse: [***Link***](https://github.com/Cemarquez/RM-Ruby/blob/master/workspace/workspaceEpsilon/diagrama.view/src/diagrama/view/views/Generador.java)

Clase de transformación Model To Text: [***Link***](https://github.com/Cemarquez/RM-Ruby/blob/master/workspace/workspaceEpsilon/diagrama.view/src/diagrama/model/TransformationM2T.java)

| código que realiza la transformación abstracto a reporte |
| --- |
| public String transformarM2T() {    String mensaje = "Se ha realziado la transformación M2T";  chooser.setFileSelectionMode(JFileChooser.DIRECTORIES\_ONLY);  String nombreProyecto ="RubyGeneration";  int returnVal = chooser.showOpenDialog(null);  if(returnVal == JFileChooser.APPROVE\_OPTION) {  File f = chooser.getSelectedFile();  limpiarTexto(f.getPath()+"/"+nombreProyecto);  f = new File(f.getPath()+"/"+nombreProyecto);  System.out.println("Creacion de paquetes:");  crearCarpetas(f.getPath());    System.out.println();  System.out.println("Creacion de clases:");  for (abstracts.MClass cl : modelFactoryAbstracta.getLstAllClass()) {  System.out.println(crearClase(f.getPath(), cl));  }    }else {    mensaje = "Debe seleccionar una ubicación para continuar!";  }      return mensaje;  }    /\*\*  \* Método que crea una clase, con sus atributos, funciones, relaciones y guarda el archivo.  \* @param path  \* @param cl  \* @return  \*/  private String crearClase(String path, MClass cl) {  StringBuilder textoCodigo = new StringBuilder();  MClass classTarget = obtenerHerencia(cl);  textoCodigo.append("=begin\n");  if(cl.getComments() == null) {  textoCodigo.append("Auto-generated code");  }else {  textoCodigo.append(cl.getComments());  }    textoCodigo.append("\n=end\n");    if(classTarget!=null) {  textoCodigo.append("class "+cl.getName() + " < "+classTarget.getName()+"\n");  }else {  textoCodigo.append("class "+cl.getName() + "\n");  }      //Creación de atributos  textoCodigo.append(crearAtributos(cl)+"\n");      //Creación de constructor  textoCodigo.append(crearConstructor(cl, classTarget) +"\n");    if(!cl.getLstMContainment().isEmpty()) {  textoCodigo.append(crearConstructorVacio() +"\n");  textoCodigo.append(crearContainment(cl) +"\n");  }    //Creación de funciones  textoCodigo.append(crearFunciones(cl));        textoCodigo.append("end");        guardarArchivo(textoCodigo.toString(), path, cl.getPath()+"/"+cl.getName());    return "Se ha generado la clase "+cl.getName();    } |

## 

| Video que ilustra la transformación |
| --- |
|  |