



Proyecto de Grado

Presentado ante la ilustre UNIVERSIDAD DE LOS ANDES como requisito final para
obtener el Título de INGENIERO DE SISTEMAS

PLUGIN DE ECLIPSE BASADO EN GEF PARA EDITAR GRÁFICAMENTE Y
GENERAR CÓDIGO EJECUTABLE DE MÁQUINAS DE ESTADOS

Por

Br. Jesús Daniel García Arzola

Tutor: Prof. Demián Gutiérrez

Enero 2013

©2013 Universidad de Los Andes Mérida, Venezuela

PLUGIN DE ECLIPSE BASADO EN GEF PARA EDITAR GRÁFICAMENTE Y GENERAR CÓDIGO EJECUTABLE DE MÁQUINAS DE ESTADOS

Br. Jesús Daniel García Arzola

Propuesta de Proyecto de Grado — Sistemas Computacionales, 137 páginas

Resumen: Las máquinas de estados o autómatas son utilizadas en el mundo de la computación desde hace muchos años. Un modelo de máquinas de estados describe cómo responde un sistema a eventos internos o externos. Los modelos de máquinas de estados sirven para responder a sistemas de tiempo real y sistemas a eventos discretos entre muchas otras aplicaciones.

En este trabajo se presenta un editor gráfico que hace uso del *framework* GEF y es implementado como un *Plugin* de Eclipse en el cual se puede describir una máquina de estados. Este editor gráfico está dirigido a desarrolladores de software y permite generar código ejecutable en Java que implemente la máquina de estados descrita gráficamente. Este software permite a los desarrolladores incorporar de forma directa modelos de máquina de estados a sus aplicaciones, reduciendo en algunos casos los tiempos y costos de desarrollo y aumentando la mantenibilidad de los productos desarrollados.

Palabras Clave: Editor Gráfico, Máquinas de Estados, GEF, Eclipse, Generación de Código.

Dedicatoria

A mis padres, a mi hermano y mi abuela

Índice

Dedicatoria	ii
Índice.....	iii
Índice de figuras	vi
Índice de tablas.....	x
Agradecimientos.....	xii
Capítulo 1	1
Introducción	1
1.1 Antecedentes.....	1
1.2 Planteamiento del Problema	2
1.3 Justificación	3
1.4 Objetivos.....	3
1.4.1 Objetivo General.....	3
1.4.2 Objetivos Específicos	3
1.5 Método de desarrollo	4
1.6 Alcance	5
1.7 Estructura del Documento	5
Capítulo 2	7
Marco teórico	7
2.1 Autómatas Finitos.....	7
2.1.1 Autómatas Finitos Deterministas.....	7
2.1.2 Modelos de Máquinas de Estados.....	8
2.1.3 Programación Orientada a Autómatas	11
2.2 Patrón de Diseño.....	17
2.2.1 Patrón MVC (Modelo- Vista- Controlador)	18
2.3 <i>Framework</i> (Marco de Trabajo)	19

Capítulo 3	21
Requisitos	21
3.4 Casos de Uso.....	22
3.4.1 Crear Documento de máquina de estados CU-01	23
3.4.2 Cargar documento de máquina de estados CU-02.....	24
3.4.3 Guardar documento de máquina de estados CU-03	25
3.4.4 Agregar estado CU-04	27
3.4.5 Eliminar estado CU-05	28
3.4.6 Cambiar tipo de estado CU-06	29
3.4.7 Agrega arco CU-07.....	30
3.4.8 Eliminar arco CU-08.....	32
3.4.9 Editar nombre de un elemento CU-09	33
3.4.10 Agregar evento a la máquina de estados CU-10.....	34
3.4.11 Eliminar Evento a la máquina de estados CU-11	35
3.4.12 Editar evento de máquina de estados CU-12.....	37
3.4.13 Cambiar evento de un arco CU-13	39
3.4.14 Seleccionar ruta destino para el archivo a generar CU-14	41
3.4.15 Generar Código de máquina de estados CU-15.....	43
3.4.16 Seleccionar plantilla alternativa para generar código CU-16.....	45
3.5 Vistas del Sistema.....	47
3.5.1 Iconos del Sistema	47
3.5.2 Editor	48
3.5.3 Creando nuevo Documento	49
3.5.4 Dialogo de Eventos en barra de propiedades.....	51
3.5.5 Dialogo para seleccionar una plantilla alternativa.....	52
3.5.6 Dialogo para seleccionar la ruta del archivo a generar.....	53
3.5.7 Menú para Generar Código	54
3.5.8 Propiedades y manipulación de un estado	55
3.5.9 Propiedades y manipulación de un arco	56

3.5.10	Validación para Máquina de Estados Determinista.....	57
Capítulo 4	58
Arquitectura y Desarrollo	58
4.1	Eclipse.....	58
4.2	Graphical Editing Framework (GEF)	60
4.2.1	Modelo.....	63
4.2.2	Vista.....	63
4.2.3	Controlador	64
4.3	Arquitectura del <i>Plugin</i> de Eclipse desarrollado	66
4.4	GEF State Machine Editor.....	67
4.4.1	Editor	67
4.4.2	Modelo.....	69
4.4.3	Controlador (<i>EditParts</i>).....	74
4.4.4	Comandos	83
4.4.5	Vistas	86
4.5	Generator Code Action Delegate.....	87
4.6	State Machine Creation Wizard.....	88
4.7	Definición de Máquina de Estados en Java	89
4.7.1	Eventos internos y externos	95
Capítulo 5	97
Casos de estudio	97
5.1	Calculadora	97
5.2	MagicRoot	103
Capítulo 6	113
Conclusiones y Recomendaciones	113
6.1	Conclusiones.....	113
6.2	Recomendaciones y Trabajos Futuros	114
Referencias	115
Apéndice A	116

XSD utilizado para el modelo de la máquina de estados	116
Plantilla de FreeMarker	119

Índice de figuras

Figura 2. 1. Autómata que acepta el lenguaje de palabras con un número impar de unos.	8
Figura 2. 2 . Máquina de estados de un horno.....	10
Figura 2. 3. Código para leer la primera palabra de una línea de manera clásica.	12
Figura 2. 4. Autómata para leer la primera palabra de una línea.	13
Figura 2. 5. Código para leer la primera palabra utilizando un autómata.	14
Figura 2. 6. Código para leer la primera palabra de una línea utilizando tabla de transición.....	16
Figura 2. 7. Máquina de estados con salida.....	17
Figura 2. 8. Arquitectura MVC.	19
Figura 3. 1. Casos de Uso del Sistema.	22
Figura 3. 2. Casos de Uso del Editor Gráfico.	23
Figura 3. 3. Diagrama de CU-01.	24
Figura 3. 4. Diagrama de Actividades de CU-01.	24
Figura 3. 5. Diagrama de CU-02.	25
Figura 3. 6. Diagrama de Actividades de CU-02.	25
Figura 3. 7. Diagrama de CU-03.	26
Figura 3. 8. Diagrama de actividades de CU-03.	26
Figura 3. 9. Diagrama de CU-04.	27
Figura 3. 10. Diagrama de actividades de CU-04.	28
Figura 3. 11. Diagrama de CU-05.	29
Figura 3. 12. Diagrama de actividades de CU-05.	29

Figura 3. 13. Diagrama de CU-06.	30
Figura 3. 14. Diagrama de actividades de CU-06.	30
Figura 3. 15. Diagrama de CU-07.	31
Figura 3. 16. Diagrama de actividades de CU-07.	31
Figura 3. 17. Diagrama de CU-08.	32
Figura 3. 18. Diagrama de actividades de CU-08.	32
Figura 3. 19. Diagrama de CU-09.	33
Figura 3. 20. Diagrama de actividades de CU-09.	33
Figura 3. 21. Diagrama de CU-10.	34
Figura 3. 22. Diagrama de actividades de CU-10.	35
Figura 3. 23. Diagrama de CU-11.	36
Figura 3. 24. Diagrama de actividades de CU-11.	37
Figura 3. 25. Diagrama de CU-12.	38
Figura 3. 26. Diagrama de CU-12.	39
Figura 3. 27. Diagrama de CU-13.	40
Figura 3. 28. Diagrama de CU-13.	41
Figura 3. 29. Diagrama de CU-14.	42
Figura 3. 30. Diagrama de actividades de CU-14.	43
Figura 3. 31. Diagrama de actividades de CU-15.	44
Figura 3. 32. Diagrama de actividades de CU-15.	45
Figura 3. 33. Diagrama de actividades de CU-16.	46
Figura 3. 34. Diagrama de actividades de CU-16.	46
Figura 3. 35. Editor de máquina de estados.	48
Figura 3. 36. <i>Wizard</i> para creación de documento 1.	49
Figura 3. 37. <i>Wizard</i> para creación de documento 2.	50
Figura 3. 38. Dialogo de Eventos.	51
Figura 3. 39. Dialogo para seleccionar una plantilla alternativa.	52
Figura 3. 40. Dialogo para seleccionar la ruta del archivo.	53
Figura 3. 41. Menú para generar Código.	54

Figura 3. 42. Propiedades de un estado.	55
Figura 3. 43. Propiedades de un arco.	56
Figura 3. 44. Validación para máquina de estados determinista.	57
Figura 4. 1. Arquitectura de Eclipse.	59
Figura 4. 2. Diagrama de secuencia de GEF.	61
Figura 4. 3 Registrar <i>listeners</i> y crear Vistas.	61
Figura 4. 4. Componentes de <i>EditPartViewer</i>	62
Figura 4. 5. Diagrama de secuencia de <i>ToolBar</i>	65
Figura 4. 6. Estructura del <i>Plugin</i> de Eclipse.	66
Figura 4. 7. Diagrama de Clases del Editor.	68
Figura 4. 8. Diagrama de clases de <i>EditorActionBarContributor</i>	69
Figura 4. 9. Diagrama de clase del Modelo.	71
Figura 4. 10. Diagrama de clases de <i>ElementEditPart</i> , <i>StateEditPart</i> y <i>LabelBaseEditPart</i>	75
Figura 4. 11. Diagrama de clases de <i>ElementEditPart</i>	76
Figura 4. 12. Diagrama de Clases de <i>StateEditPart</i>	77
Figura 4. 13. Diagrama de Clases de <i>LabelBaseEditPart</i>	78
Figura 4. 14. Diagrama de Clases de <i>ArcEditPart</i>	79
Figura 4. 15. Diagrama de Clases de <i>CanvasEditPart</i>	81
Figura 4. 16. Diagrama de Clases de <i>EditorEditPartFactory</i>	82
Figura 4. 17. Código de metodo <i>createEditPart</i>	83
Figura 4. 18. Diagrama de Clases de los Comandos.	85
Figura 4. 19. Código de la clase <i>StateFigure</i>	86
Figura 4. 20. Arquitectura usada con <i>FreeMarker</i> y JAXB.	87
Figura 4. 21. Diagrama de clase de <i>StateMachineCreationWizard</i>	88
Figura 4. 22. Diagrama de las salidas de la máquina de estados.	89
Figura 4. 23. Diagrama de las salidas de la máquina de Estados.	90
Figura 4. 24. Código de enumerado de estados.	90
Figura 4. 25. Código de Enumerado de eventos.	91

Figura 4. 26. Código de variable para manejar estados.	91
Figura 4. 27. Código de función de transición.	91
Figura 4. 28. Código de método <i>internalFireEvent</i>	92
Figura 4. 29. Código de método <i>StateNeutro</i>	93
Figura 4. 30. Código de <i>hook</i> para la transición.	93
Figura 4. 31. Código de método <i>extStateChange</i>	94
Figura 4. 32. Código de método <i>entStateChange</i>	94
Figura 4. 33. Código de método <i>eviStateChange</i>	95
Figura 4. 34. Eventos internos y externos.	96
Figura 5. 1. Máquina de estados de calculadora.	98
Figura 5. 2. Diagrama de clase de Máquina de estados de calculadora.	99
Figura 5. 3. Código de método <i>StateOPE1</i>	101
Figura 5. 4. <i>Hook</i> para el estado Operando 1 Entero (OPE1).	101
Figura 5. 5. Método <i>addNum</i>	102
Figura 5. 6. Metodo <i>handleEvent</i>	102
Figura 5. 7. Arquitectura de <i>MagicRoot</i>	104
Figura 5. 8. Máquina de estados de <i>MagicRoot</i>	105
Figura 5. 9. Código de método <i>estadoTurnoR</i>	107
Figura 5. 10. Diagrama de Clases de nueva máquina de estados de <i>MagicRoot</i>	108
Figura 5. 11. Código de método <i>stateTurnoRojo</i>	110
Figura 5. 12. <i>Hook</i> implementado en la clase <i>StateMachineMagicRootImpl</i>	111
Figura 5. 13. Método <i>initListo</i>	111
Figura A.1. XSD utilizado para el modelo de la máquina de estados.	118
Figura A.2. Plantilla de <i>FreeMarker</i>	124

Índice de tablas

Tabla 2. 1. Tabla de transición para autómata que acepta el lenguaje de palabras con un número impar de unos.	8
Tabla 2. 2. <i>Hooks</i> de la máquina de estados.....	17
Tabla 3. 1. CU-01.....	23
Tabla 3. 2. CU-02.....	24
Tabla 3. 3. CU-03.....	25
Tabla 3. 4. CU-04.....	27
Tabla 3. 5. CU-05.....	28
Tabla 3. 6. CU-06.....	29
Tabla 3. 7. CU-07.....	31
Tabla 3. 8. CU-08.....	32
Tabla. 3. 9 CU-09.....	33
Tabla 3. 10. CU-10.....	34
Tabla. 3. 11 CU-11.....	36
Tabla. 3. 12 CU-12.....	38
Tabla 3. 13. CU-13.....	40
Tabla 3. 14. CU-14.....	42
Tabla 3. 15. CU-15.....	44
Tabla 3. 16 CU-16.....	45
Tabla 3. 17. Iconos del Sistema.....	47
Tabla 3. 18. Editor.....	48
Tabla 3. 19. <i>Wizard</i> para crear nuevo documento.....	49
Tabla 3. 20. <i>Wizard</i> para crear nuevo Documento.....	50
Tabla 3. 21. Dialogo de eventos.....	51
Tabla 3. 22. Dialogo para seleccionar una plantilla alternativa.	52
Tabla 3. 23. Dialogo para seleccionar ruta para el archivo a generar.	53

Tabla 3. 24. Menú para generar código.....	54
Tabla 3. 25. Propiedades de un estado.	55
Tabla 3. 26. Propiedades de un arco.....	56
Tabla 3. 27. Validación para máquina de estados determinista.	57
Tabla 4. 1. Métodos de la Clase Editor.	68
Tabla 4. 2. Métodos de la clase <i>EditorActionBarContributor</i>	69
Tabla 4. 3. Tabla de Métodos de los modelos.....	73
Tabla 4. 4. Atributos de los modelos.....	73
Tabla 4. 5. Métodos de la clase <i>ElementEditPart</i>	77
Tabla 4. 6. Métodos de la clase <i>StateEditPart</i>	78
Tabla 4. 7. Métodos de la figura 4.14.....	80
Tabla 4. 8. Métodos de la figura 4.15.....	82
Tabla 4. 9. Métodos de la figura 4.18.....	83
Tabla 4. 10. Métodos de las clases <i>Wizard</i> y <i>CreationPage</i>	89
Tabla 4. 11 <i>Hooks</i> de máquina de estados generada.	89
Tabla 5. 1. Estado de la máquina de estados de una calculadora.	98
Tabla 5. 2. Eventos de la máquina de estados de una calculadora.	99

Agradecimientos

A dios todo poderoso y a la virgen de Guadalupe, por representar la fe y la esperanza en los momentos difíciles.

A mis padres, por ser un ejemplo a seguir durante toda mi vida, y estar allí en los momentos más difíciles apoyándome incondicionalmente, mucho de lo que soy hoy se lo debo a ellos. Gracias por todo.

A mi abuela, por ser mi segunda madre, y estar allí conmigo en cada momento de mi vida, siempre apoyándome con sus consejos y vivencias.

A mi hermano, por acompañarme y apoyarme incondicionalmente, siempre su nobleza y humildad serán un ejemplo a seguir.

A mi tío Alejandro, por ser un apoyo a lo largo de mi vida, y darme un consejo siempre que lo he necesitado, siempre será un ejemplo de lucha y perseverancia para mí.

A mi tutor el profesor Demián, por ser mi gran mentor y amigo durante la mitad de mi carrera, me ha dejado aprendizajes con muchísimo valor en lo personal y profesional. Muchísimas gracias.

Al profesor Francisco Carreras, por darme sus conocimientos en mi primer semestre, y ser un gran apoyo dentro de la Universidad, cuando apenas estaba comenzando.

Al profesor Domingo, por haberme ayudado y aconsejado en un momento difícil de mi carrera, y en su momento haberme brindado sus conocimientos para mi formación.

A todos los Profesores, Personal Obrero y Administrativo de la Facultad de Ingeniería, por darme sus conocimientos y su apoyo a lo largo de mi carrera.

A todos los familiares y amigos, por apoyarme a lo largo de mi carrera, y darme un consejo cuando lo he necesitado.

Capítulo 1

Introducción

En este capítulo se expondrán los antecedentes, el planteamiento del problema, la justificación y los objetivos que encaminarán el desarrollo del mismo, el alcance del proyecto, la metodología para su desarrollo y finalmente se dará una descripción de la estructura del presente documento.

1.1 Antecedentes

Hoy en día existen diferentes tipos de software para editar autómatas o máquinas de estado gráficamente. Uno de ellos es JFLAP el cual es un paquete de herramientas gráficas que puede ser usado como una ayuda en el aprendizaje de los conceptos básicos de los lenguajes formales y la teoría de autómatas (Rodger, 2012).

Duffner & Strobel (2012) crean la aplicación Qfsm, la cual es un editor para máquinas de estados finitos que implementa las siguientes características:

- Exportar el diagrama en EPS, SVG, PNG.
- Simulación interactiva.
- Generar un archivo de exportación para la generación de código en los lenguajes de programación C/C++, Java o Ruby.

Darovsky (2012) crea la aplicación FSME que implementa la siguiente funcionalidad:

- Permite editar gráficamente un autómata y guarda la descripción de dicho autómata en un archivo XML.
- A través de un archivo XML que contiene en texto plano una descripción que pertenece un autómata finito, permite generar el código fuente para los lenguajes de programación C++ y Python.
- Permite monitorear las máquinas de estado descritas gráficamente por un usuario en tiempo real.

En la Escuela de Ingeniería de Sistemas en el semestre A2011, durante el curso de la materia Ingeniería de Software, se creó el juego MagicRoot. Este es un juego de cartas con una arquitectura cliente servidor. En el servidor se codificó explícitamente una máquina de estados, la cual describía las reglas del juego y controlaba todo lo que sucedía mientras dos usuarios jugaban en línea.

1.2 Planteamiento del Problema

Muchas aplicaciones de software implementan sistemas a eventos discretos. Por experiencias de los autores de este trabajo, muchos desarrolladores atacan estos sistemas sin los formalismos correctos generando aplicaciones difíciles de mantener, lo que deriva en altos costos de desarrollo. Esto se debe a que cuando surge la necesidad de realizar cambios en el sistema, es difícil para los desarrolladores porque dichos sistemas contienen clases con grandes cantidades de líneas de código, muchas estructuras de decisión, y en la mayoría de los casos, estructuras excesivamente anidadas unas dentro de las otras. Utilizando los formalismos correctos se pueden generar aplicaciones con alta calidad y mantenibilidad.

1.3 Justificación

Muchos productos de software en el mundo de la computación presentan problemas que pueden ser resueltos por medio de máquinas de estados. Un modelo de máquina de estados describe cómo responden sistemas a eventos internos o externos, este tipo de formalismo se utiliza a menudo para modelar sistemas a eventos discretos entre muchas otras aplicaciones. El modelo matemático de un Autómata Finito describe los estados del sistema y los eventos que provocan transiciones de un estado a otro. Contar con una herramienta que permita que en base a la descripción de una máquina de estados se pueda generar el código ejecutable correspondiente, reduciría los tiempos y costos de desarrollo y mejoraría la calidad y mantenibilidad del producto a desarrollar.

1.4 Objetivos

1.4.1 Objetivo General

Desarrollar un editor de gráfico de máquina de estados que genere una instrumentación ejecutable en Java de dicha máquina de estados.

1.4.2 Objetivos Específicos

- Entender la arquitectura de Eclipse y GEF para implementar un *Plugin* que permita editar gráficamente una máquina de estados.
- Manejar funcionalmente JAXB que es una herramienta de lectura/escritura de XML orientada a objetos.
- Manejar funcionalmente *FreeMarker* que es una herramienta de plantillas para generar código.
- Implementar un *Plugin* de Eclipse utilizando GEF que permita editar gráficamente una máquina de estados y las propiedades de todos sus elementos.

- Definir la estructura ejecutable de una máquina de estados en Java.
- Escribir una Plantilla de *FreeMarker* para generar la estructura ejecutable de una máquina de estados en Java.
- Realizar dos casos de estudio que permitan validar la funcionalidad del editor y del generador de código.

1.5 Método de desarrollo

La metodología que se seguirá en este proyecto es el desarrollo en espiral. Según Sommerville (2007), esta metodología fue propuesta inicialmente por Boehm (1988). El desarrollo en espiral, más que representar el proceso de software como una secuencia de actividades con retrospectiva de una actividad a otra, lo representa como una espiral. Cada ciclo de la espiral representa una fase del desarrollo de software. Así el ciclo más interno podría referirse a la viabilidad del sistema, el siguiente ciclo a la definición de requerimientos, el siguiente al desarrollo del sistema y así sucesivamente.

Cada ciclo del espiral se divide en cuatro regiones bien definidas:

- Definición de Objetivos: en esta región se definen los objetivos específicos. Se identifican los riesgos, las restricciones del producto y del proceso. Por último se traza un plan detallado de gestión.
- Evaluación y reducción de Riesgos: se lleva a cabo un análisis detallado para cada uno de los riesgos y se definen los pasos para reducir dichos riesgos.
- Desarrollo y Validación: se elige un modelo para el desarrollo del sistema dependiendo de los resultados que arroje la evaluación de riesgos.
- Planificación: se evalúa el proyecto hasta el trabajo hecho en el ciclo actual. En caso de necesitar otro ciclo se planifica dicho ciclo.

Según Sommerville (2007), en resumen se puede decir que:

Un ciclo de la espiral empieza con la elaboración de objetivos, como el rendimiento y la funcionalidad. Entonces se enumeran formas alternativas de alcanzar estos objetivos y las restricciones impuestas en cada una de ellas. Cada alternativa se evalúa contra cada objetivo y se identifican las fuentes de riesgo del proyecto. El siguiente paso es resolver estos riesgos mediante actividades de recopilación de información como la de detallar más el análisis, la construcción de prototipos y la simulación. Una vez que se han evaluado los riesgos, se lleva a cabo cierto desarrollo, seguido de una actividad de planificación para el siguiente ciclo.

1.6 Alcance

Diseñar, implementar y validar un *Plugin* de Eclipse que permita editar gráficamente máquinas de estados y generar el código ejecutable en Java correspondiente a dicha máquina de estados.

1.7 Estructura del Documento

Capítulo 1, Introducción. En este se definen los antecedentes, así como también, el planteamiento del problema, la justificación y los objetivos, el alcance del proyecto y la metodología para su desarrollo.

Capítulo 2, Marco teórico. Contiene los fundamentos teóricos necesarios para el entendimiento y comprensión del proyecto, entre los cuales, se describe detalladamente los autómatas finitos y los autómatas con salida.

Capítulo 3, Requerimientos. Comprende el todos los requisitos del sistema, utilizando casos de uso diagramas de actividades de UML, y mostrando las vistas prototipos del sistema para luego hacer la correspondencia de cada una de ellas con los casos de uso.

Capítulo 4, Arquitectura y Desarrollo. En este capítulo se da una breve explicación de la arquitectura de Eclipse que componentes se usuario, luego se explica la arquitectura de GEF, utilizando diagramas de bloques y de secuencia de UML, por último se explican las clases del sistema y sus responsabilidades utilizando diagramas de clases de UML.

Capítulo 5, Pruebas. Comprende dos pruebas realizadas a la herramienta, la primera es la máquina de estados de MagicRoot (Zamora, 2012), la segunda es una calculadora como la que brinda los sistemas operativos Linux o Windows.

Capítulo 6, Conclusiones y recomendaciones. Se describe las conclusiones generales del trabajo realizado y las recomendaciones para trabajos futuros.

Capítulo 2

Marco teórico

En este capítulo, se describen los fundamentos teóricos necesarios para el entendimiento y comprensión de este proyecto.

2.1 Autómatas Finitos

2.1.1 Autómatas Finitos Deterministas

Un AFD es aquel que siempre está en un solo estado, después de leer cualquier secuencia de entrada. El término “determinista” hace referencia al hecho de que, para cada entrada, existe un único estado al que el autómata puede llegar partiendo del estado actual (Hopcroft, Motwani, & Ullman, 2002). La definición formal la podemos describir de la siguiente manera:

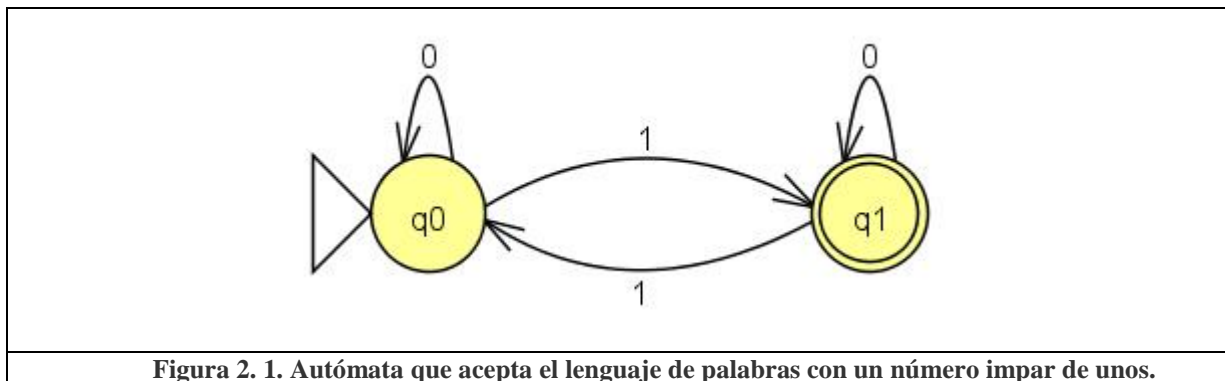
Un autómata finito es una quintupla $M = (Q; A; \delta; q_0; F)$ en que

- Q es un conjunto finito llamado conjunto de estados.
- A es un Alfabeto llamado alfabeto de entrada.
- δ es una aplicación llamada función de transición ($\delta: Q \times A \rightarrow Q$).
- q_0 es un elemento de Q , llamado estado inicial.
- F es un subconjunto de Q , llamado conjunto de estados finales.

El diagrama de transición de un Autómata de Estado Finito es un grafo, en el que los vértices representan los distintos estados, y los arcos las transiciones entre los estados. Cada

arco va etiquetado con el símbolo que corresponde a dicha transición. El estado inicial y los finales vienen señalados de forma especial (por ejemplo, con un ángulo el estado inicial y con un doble círculo los finales). (Moral, 2001)

Una tabla de transiciones es una representación tabular convencional de una función como δ , que recibe dos argumentos y devuelve un valor. Las filas de la tabla corresponden a los estados, y las columnas a las entradas. El valor correspondiente a una fila del estado “q” y a la columna de entrada “a” es el estado $\delta(q, a)$ (Hopcroft, Motwani, & Ullman, 2002). La figura 2.1 tomada de Moral (2001) ilustra un autómata que acepta el lenguaje de palabras con un número impar de unos. Su tabla de transición se encuentra en la tabla 2.1.



Δ	0	1
$\rightarrow q_0$	q_0	q_1
$*q_1$	q_1	q_0

Tabla 2. 1. Tabla de transición para autómata que acepta el lenguaje de palabras con un número impar de unos.

2.1.2 Modelos de Máquinas de Estados

Un modelo de máquina de estados describe cómo responde un sistema a eventos internos o externos. El modelo de máquina de estados muestra los estados del sistema, y los eventos provocan las transiciones de un estado a otro. Este tipo de modelo se utiliza a

menudo para modelar sistemas de tiempo real, debido a que estos sistemas suelen estar dirigidos por estímulos procedentes del entorno del sistema. Un modelo de máquinas de estados de un sistema supone que, en cualquier momento, el sistema está en uno de varios estados posibles. Cuando se recibe un estímulo, éste puede disparar una transición a un estado diferente. (Sommerville, 2007). Los autómatas con salida son máquinas de estados finitos, existen dos tipos:

- Máquinas de Moore: con salida asociada al estado.
- Máquinas de Mealy: con salida asociada a la transición.

Una máquina de Moore es una séxtupla $\{(Q, A, B, \delta, \lambda, q_0)\}$ donde todos los elementos son como en los autómatas finitos deterministas, excepto:

- B alfabeto de salida.
- $\lambda: Q \rightarrow B$ que es una aplicación que hace responder a cada estado su salida correspondiente.

No hay estados finales, porque ahora la respuesta, es la cadena formada por los símbolos de salida, correspondientes a los estados por los que pasa el autómata. Si el autómata lee la cadena u , y pasa por los estados $q_0, q_1 \dots q_n$ entonces, la salida que produce será $\lambda(q_0) \lambda(q_1) \dots \lambda(q_n)$. Si se le suministra la cadena vacía como entrada, la salida es $\lambda(q_0)$. (Moral, 2001).

Una máquina de Mealy es una séxtupla $M = (Q, A, B, \delta, \lambda, q_0)$, donde es igual que en las máquinas de Moore, excepto:

$\lambda: Q \times A \rightarrow B$ que es una aplicación que da una salida para cada transición.

Es decir, la salida depende del estado en que este al autómata y del símbolo leído. Si la entrada es $a_1 \dots a_n$ y pasa por los estados $q_0, q_1 \dots q_n$, la salida es:

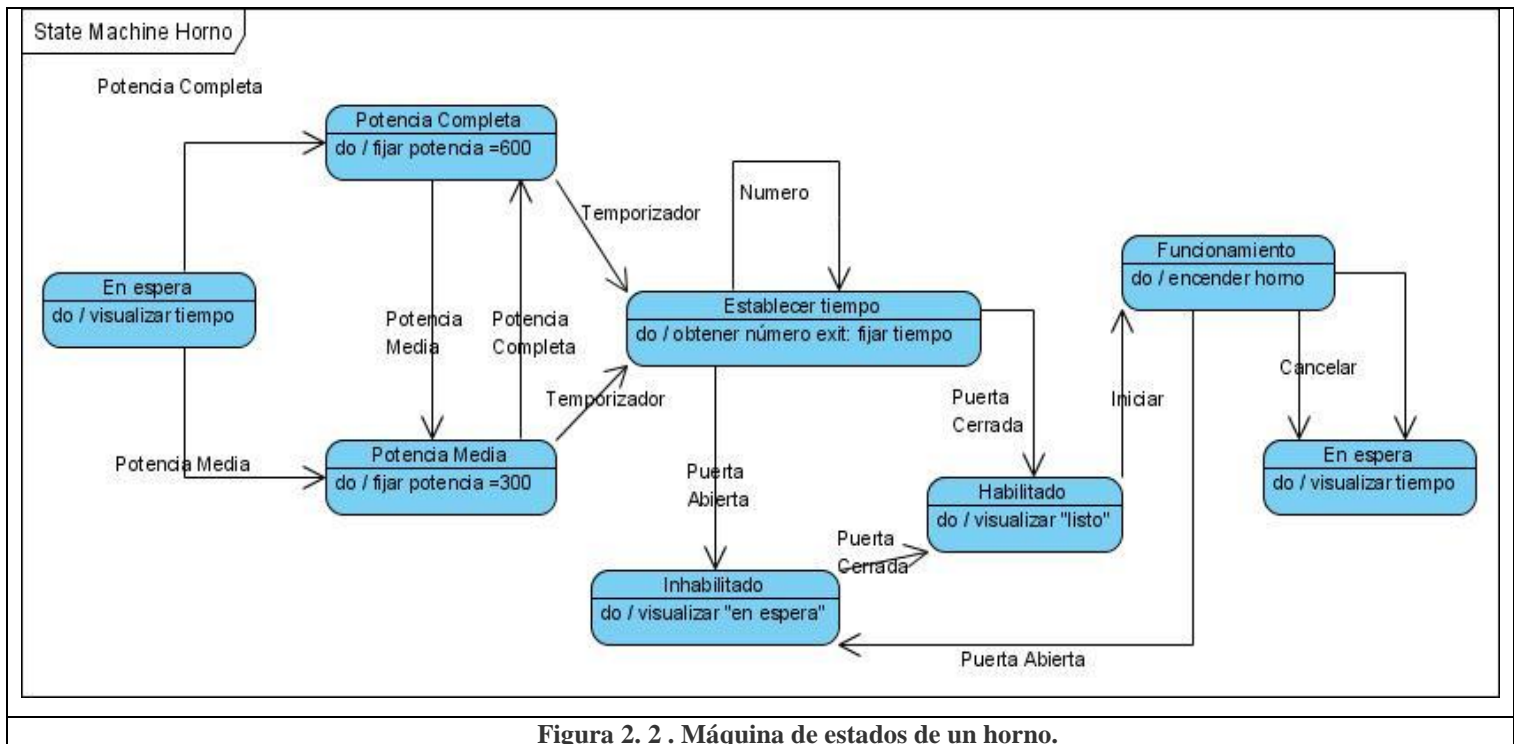
$$\lambda(q_0, a_1) \lambda(q_1, a_2) \dots \lambda(q_{n-1}, a_n)$$

Si se le suministra la cadena vacía como entrada, produce la cadena vacía como salida. (Moral, 2001).

Un ejemplo para un autómata con salida, se encuentra en la figura 2.2 tomada de Sommerville (2007). Es un horno microondas, equipado con botones para fijar la potencia, el temporizador y para iniciar el sistema. Para simplificar el modelo, se supone que la secuencia de acciones al usar el microondas son:

1. Seleccionar el nivel de potencia.
2. Introducir el tiempo de cocción.
3. Pulsar el botón de inicio, y la comida se cocina durante el tiempo establecido.

El horno no debe funcionar cuando la puerta está abierta y cuando se completa la cocción suena un timbre. El horno dispone de una pantalla para visualizar los mensajes de alerta y precaución. El horno es un autómata con salida, debido a que realiza una acción en cada estado, dicha acción representa el alfabeto de salida.



2.1.3 Programación Orientada a Autómatas

La programación orientada a Autómatas es un paradigma de programación en el cual el programa se realiza utilizando un modelo de máquina de estados finita. (Wikipedia, 2012)

Las siguientes propiedades son los indicadores claves de la programación basada en autómatas:

- La ejecución del programa es distribuida a lo largo de del conjunto de transiciones del autómata. Cada transición es una ejecución de una sección de código que tiene un punto único de entrada. Esa sección puede ser una función, otra rutina u otro autómata anidado. Las transiciones pueden ser divididas en subsecciones para ser ejecutadas en función de los diferentes estados anidados. (Wikipedia, 2012)
- Cualquier comunicación entre dos cambios de estados es solo posible por medio de un conjunto de variables bien definidas llamadas “el estado”. Entre dos cambios de estado no pueden existir componentes implícitos del estado, tales como variables locales o de otro tipo. Es decir el estado de todo el programa tomado dos momentos cualesquiera a la entrada de cualquier cambio de estado, solo puede ser diferenciarse por los valores del conjunto de variables definidas como “el estado”. (Wikipedia, 2012)

A continuación se explicara un breve ejemplo tomado de Wikipedia (2012). La idea es leer un texto de una entrada hecha por el teclado, línea por línea e imprimir la primera palabra de cada línea. Se necesita obviar los espacios de línea y todos los caracteres restantes hasta el fin de línea. La figura 2.3 tomada de Wikipedia (2012) ilustra un ejemplo hecho en C que se encarga de hacer esta tarea de la manera clásica. Existe un *do while* grande para leer la entrada por el teclado. El primer *while* se encarga de recibir los espacios en blanco, el segundo *while* es el que imprime la primera palabra de una línea, por último el tercer *while*

lee la entrada hecha por el teclado mientras el carácter sea diferente de un salto de línea, es decir lee el resto de la línea.

```
#include <stdio.h>
int main(void)
{
    int c;
    do {
        c = getchar();
        while(c == ' ')
            c = getchar();
        while(c != EOF && c != ' ' && c != '\n') {
            putchar(c);
            c = getchar();
        }
        putchar('\n');
        while(c != EOF && c != '\n')
            c = getchar();
    } while(c != EOF);
    return 0;
}
```

Figura 2. 3. Código para leer la primera palabra de una línea de manera clásica.

El mismo programa puede ser resultado pensándolo en función de un autómata finito determinista en el que existirían tres estados: *before*, *inside* y *after*. El programa sería como el que se muestra en la figura 2.5 tomada de Wikipedia (2012). La idea es tener los estados en un tipo enumerado y una función *step* que es la encargada del manejo de los estados. Esta función recibe una variable enumerada de tipo estado junto con la entrada del teclado. Dependiendo del estado actual se ejecuta cierto código. En el estado *before* se imprime el carácter fin de línea, y si viene un carácter diferente al espacio en blanco se pasa al estado *inside*. En el estado *inside* es cuando se manda a imprimir la primera palabra de la línea, si viene el carácter salto de línea se cambia al estado *before*, y si viene el carácter espacio en blanco se cambia al estado *after*. Por último el estado *after* espera a que llegue el carácter salto de línea para cambiar al estado *before*. Las ventajas en cuanto al código anterior es que

se está leyendo la entrada una sola vez, y existe un solo bucle que maneja todo. En la figura 2.4 tomada de Wikipedia (2012) se puede ver el autómata que maneja el programa, en el que:

- “N*” representa los saltos de línea.
- “A*” representa la primera palabra de una línea.
- “A” representa las demás palabras de una línea.
- “S” los espacios de línea.

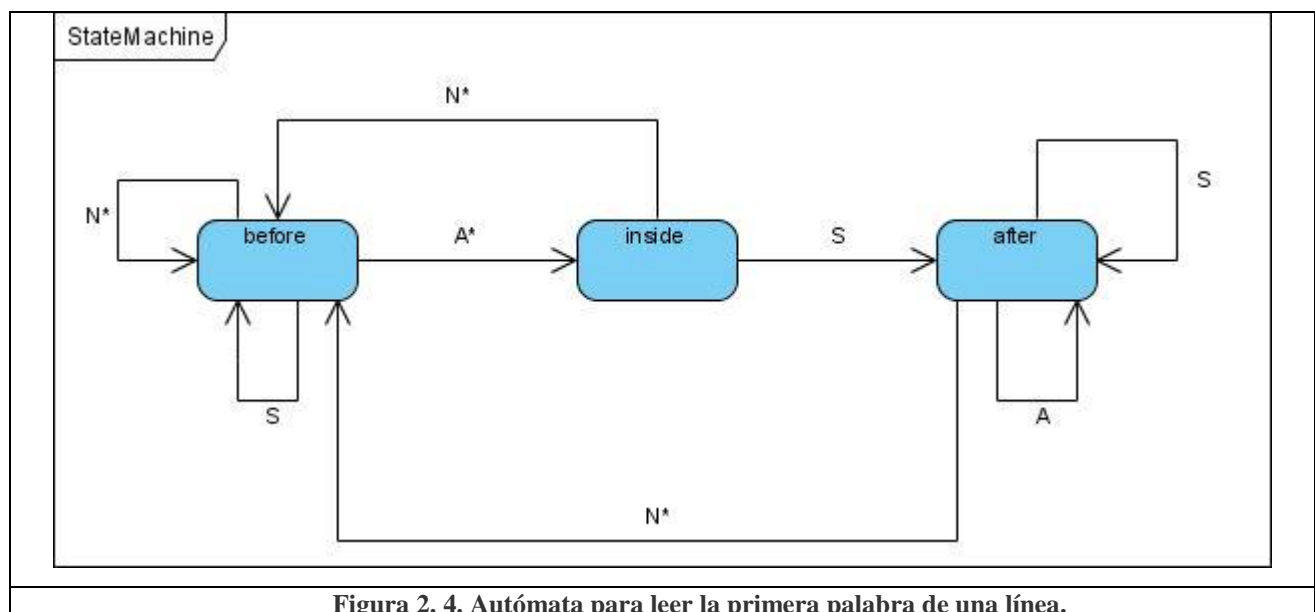


Figura 2. 4. Autómata para leer la primera palabra de una línea.

```
#include <stdio.h>

enum states { before, inside, after };

void step(enum states *state, int c) {
    if(c == '\n') {
        putchar('\n');
        *state = before;
    } else
    switch(*state) {
        case before:
            if(c != ' ') {
                putchar(c);
                *state = inside;
            }
            break;
        case inside:
            if(c == ' ') {
                *state = after;
            } else {
                putchar(c);
            }
            break;
        case after:
            break;
    }
}

int main(void) {
    int c;
    enum states state = before;
    while((c = getchar()) != EOF) {
        step(&state, c);
    }
    return 0;
}
```

Figura 2. 5. Código para leer la primera palabra utilizando un autómata.

El código de la figura 2.6 tomada de Wikipedia (2012) ilustra otra manera de representar el mismo programa utilizando una tabla de transición. La estructura *the_table[3][3]* representa la tabla de transición del autómata, cada posición de la tabla tiene el estado al que debe cambiar el autómata, y una bandera para saber si imprimir un carácter o no, las filas representan los estados, y las columnas los eventos, estos últimos son los caracteres de entrada. Existe una estructura llamada *branch* que tiene dos atributos *new_state* que es para llevar el estado actual y *should_putchar* que es una bandera para saber si imprimir un carácter en el estado *inside*, o un fin de línea en el estado *before*, para el estado *after* no se imprime nada. La función *step* básicamente tiene el mismo funcionamiento que la mostrada en la figura 2.5, dependiendo de la entrada del carácter “c” y del estado actual buscará la transición correspondiente en la tabla de transición y luego actualiza el estado e imprime el carácter correspondiente a dicha transición.

```
#include <stdio.h>
enum states { before = 0, inside = 1, after = 2 };
struct branch {
    unsigned char new_state:2;
    unsigned char should_putchar:1;
};
struct branch the_table[3][3] = {
    /* ' ' ' \n' others */
    /* before */ { {before,0}, {before,1}, {inside,1} },
    /* inside */ { {after, 0}, {before,1}, {inside,1} },
    /* after */ { {after, 0}, {before,1}, {after, 0} }
};
void step(enum states *state, int c)
{
    int idx2 = (c == ' ') ? 0 : (c == '\n') ? 1 : 2;
    struct branch *b = & the_table[*state][idx2];
    *state = (enum states) (b->new_state);
    if(b->should_putchar) putchar(c);
}
int main(void)
{
    int c;
```

```
enum states state = before;
while((c = getchar()) != EOF)
    step(&state, c);
return 0;
}
```

Figura 2. 6. Código para leer la primera palabra de una línea utilizando tabla de transición.

La programación orientada a autómatas es muy importante en el mundo de la computación, debido a que, muchas aplicaciones de software implementan sistemas a eventos discretos. Muchos desarrolladores de software atacan estos problemas sin utilizar los formalismos correctos, generando aplicaciones difíciles de mantener. Utilizando programación orientada a autómatas, cuando un software implementa sistemas a eventos discretos, se pueden conseguir aplicaciones de alta calidad y mantenibilidad.

En este trabajo la idea es brindarle al desarrollador un editor gráfico de máquina de estados, que genere el código ejecutable de la máquina de estados descrita gráficamente en el editor, con la finalidad de facilitarle al desarrollador el uso de la programación orientada a autómatas. Las máquinas de estados que se van a generar con el editor gráfico presentado en este trabajo, son las máquinas de Moore, y las Máquinas de Mealy. El editor de máquina de estados, va a generar una clase, que se encargara de manejar toda la lógica de la máquina de estados como tal, y brindara al desarrollador extensiones de código llamadas *hook*, donde el desarrollador, podrá darle una salida a la máquina de estados, dicha salida puede ser, tanto para los estados (Moore), como para las transiciones (Mealy). Se debe agregar que en los capítulos siguientes se cambiara el vocabulario, el alfabeto de entrada, pasara a ser el conjunto de eventos de la máquina de estados, y un evento es un carácter del alfabeto de entrada. Como se muestra en la figura 2.7, existe una salida asociada a los estados y a las transiciones. La tabla 2.2, muestra la descripción de cada *hook* que tiene el desarrollador, para darle una, o más salidas a la máquina de estados.

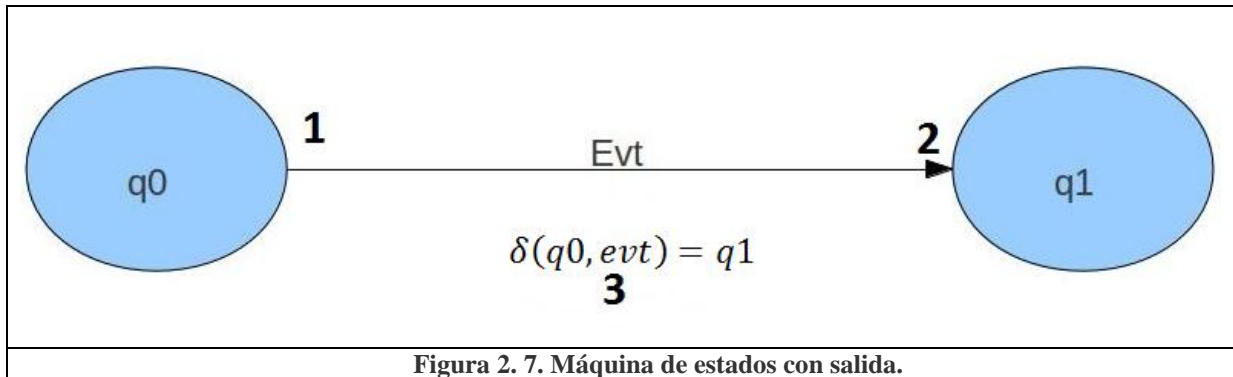


Figura 2. 7. Máquina de estados con salida.

Hook	Descripción
1,2	hook asociado a un estado (Moore)
3	hook asociado a una transición, dependiendo del estado actual y del evento se produce una salida (Mealy)

Tabla 2. 2. Hooks de la máquina de estados.

2.2 Patrón de Diseño

Un patrón de diseño es una descripción de clases y objetos relacionados, que estén particularizados para resolver un problema de diseño en general, en un determinado contexto. Un patrón de diseño nomina, abstrae e identifica los aspectos claves de una estructura de diseño común, lo que los hace útiles para crear un diseño orientado a objetos reusable. El patrón de diseño identifica las clases e instancias participantes, sus roles y colaboraciones, y la distribución de responsabilidades. (Gamma, Helm, Johnson, & Vlissides, 2003)

En este trabajo se utiliza el *framework* GEF, el cual implementa el patrón MVC que permite desacoplar el modelo de la vista utilizando un intermediario llamado controlador. Este patrón se explicará más a detalle en la sección subsiguiente.

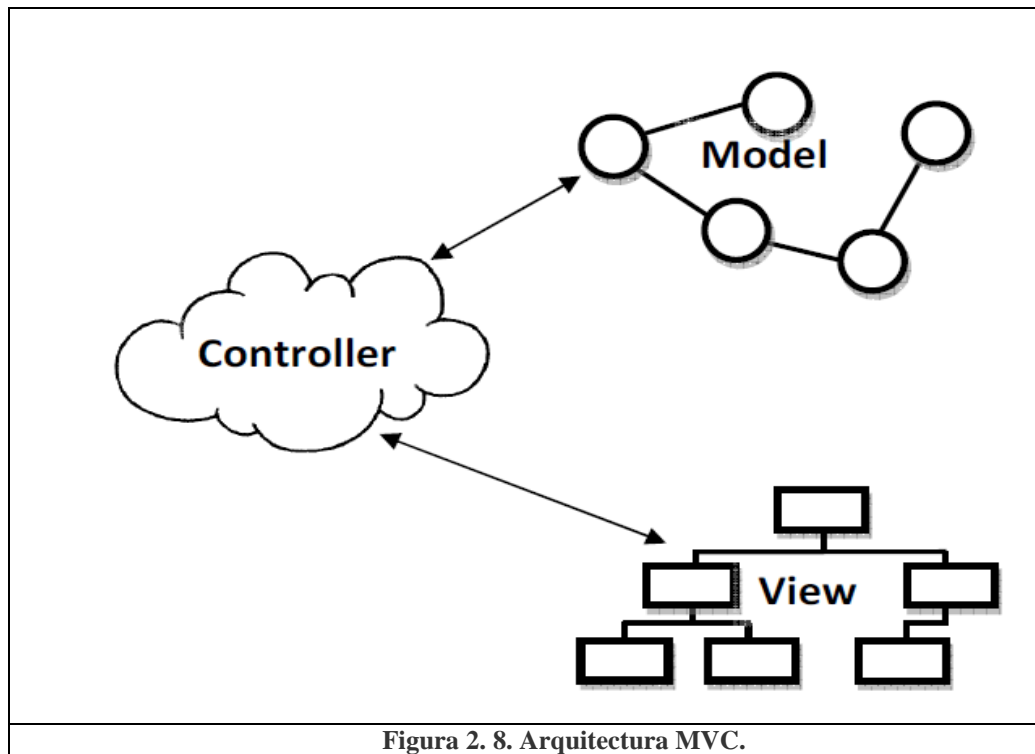
2.2.1 Patrón MVC (Modelo- Vista- Controlador)

Una buena pauta de diseño es mantener separado el software requerido para la presentación de la información misma. Separar el sistema de presentación de los datos nos permite cambiar la representación en la pantalla del usuario sin tener que cambiar el sistema de cálculo subyacente (Sommerville, 2007). El patrón MVC, el cual estuvo disponible por primera vez en *Smalltalk* (Goldberg & Robson, 1983), es una forma efectiva para permitir representaciones múltiples de datos. Los usuarios pueden interactuar con cada presentación utilizando un estilo apropiado a esta. Los datos a visualizar se encapsulan en un modelo de objetos. Cada modelo de objetos puede tener asociado varias vistas diferentes donde cada vista es una representación de visualización diferente del modelo. Cada vista tiene un objeto controlador asociado que maneja las entradas del usuario y la interacción de los dispositivos. (Sommerville, 2007).

MVC consiste en tres tipos de objetos. El modelo es el objeto de la aplicación, la vista su representación en pantalla y el controlador define el modo en que la interfaz reacciona ante una entrada del usuario. (Gamma, Helm, Johnson, & Vlissides, 2003).

MVC desacopla las vistas de los modelos estableciendo entre ellos un protocolo de suscripción/notificación. Una vista debe asegurarse de que su apariencia refleje el estado del modelo. Cada vez que cambian los datos del modelo, este se encarga de avisar a las vistas que dependen de él. Como respuesta a dicha notificación, cada vista tiene la oportunidad de actualizarse a sí misma. MVC encapsula los mecanismos de respuesta en un objeto llamado controlador. Una vista usa una instancia de una subclase de controlador para implementar una determinada estrategia de respuesta; para implementar una estrategia diferente, simplemente basta con sustituir la instancia por otra clase de controlador. Incluso es posible cambiar el controlador de una vista en tiempo de ejecución, para hacer que la vista cambie el modo en que responde a la entrada de usuario. (Gamma, Helm, Johnson, & Vlissides, 2003)

En la figura 2.8, se puede observar como interactúa el modelo, la vista y el controlador, la idea es que el controlador es el intermediario entre el modelo y la vista, si el modelo tiene un cambio, es el controlador el encargado de reflejar en la vista dicho cambio. Si el usuario hace un cambio en la vista, es el controlador el encargado de modificar el modelo.



2.3 Framework (Marco de Trabajo)

Un *framework*, es un diseño de un subsistema formado por una colección de clases concretas y abstractas y la interfaz entre ellas (Wirfs-Brock & Johnson, 1990).

Tal y como el nombre sugiere, un *framework* es una estructura genérica que puede ser extendida para crear un subsistema o aplicación más específica. Éste, es implementado como una colección de clases de objetos concretas y abstractas. Para extender el *framework*, se tienen que añadir clases concretas que hereden operaciones de las clases abstractas en el *framework*. (Sommerville, 2007).

En este trabajo se usó el *framework* GEF, en la mayoría de los casos se escribieron clases, que heredaban de clases base de GEF y Eclipse, y se tenían que sobrescribir ciertos métodos dependiendo de las necesidades. El uso del *framework* facilitó mucho el trabajo ya que éste obliga al desarrollador a utilizar el patrón MVC, y brinda un conjunto de operaciones genéricas para que el desarrollador haga uso de ellas. Como por ejemplo arrastrar figuras en un *Canvas*, o la pila para deshacer y rehacer, entre otras operaciones.

Capítulo 3

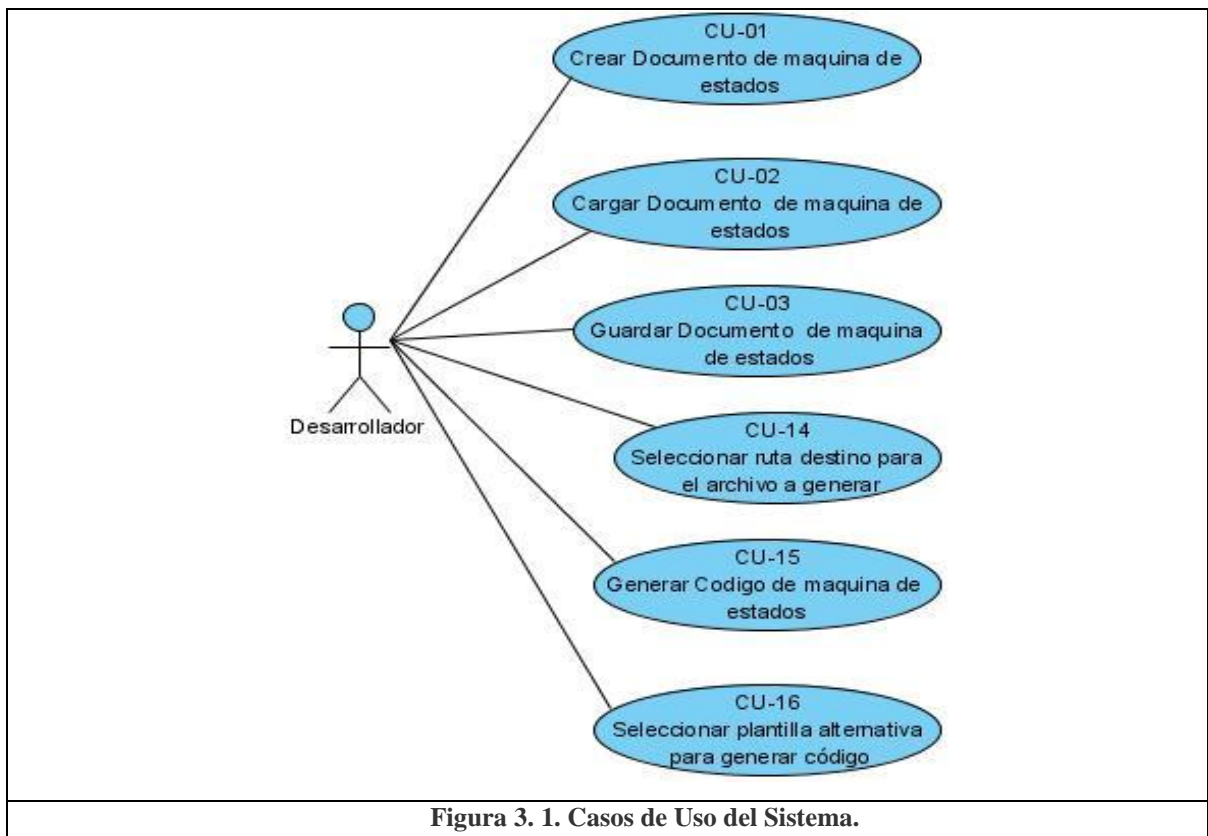
Requisitos

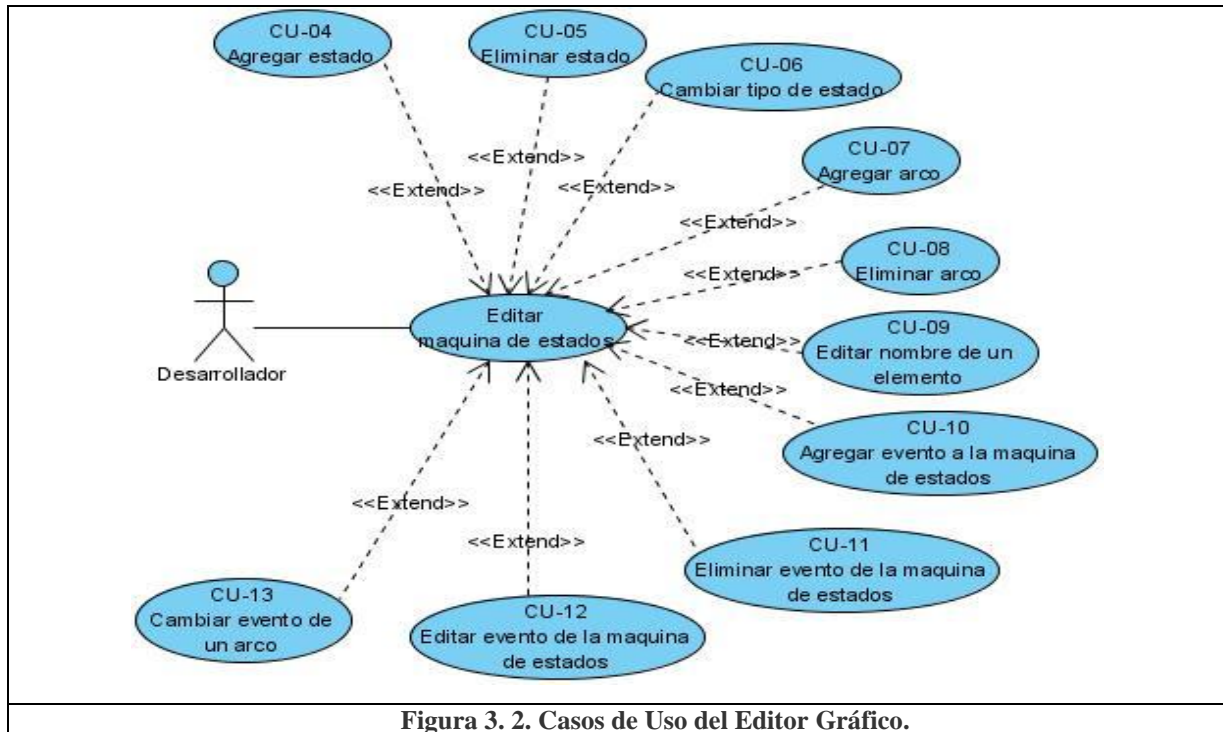
En este capítulo se explicará con casos de uso, y las vistas prototipos todos los requerimientos del sistema presentado en este trabajo. Se necesita un editor gráfico lo suficientemente flexible como para poder representar una máquina de estados con todas sus características. Se debe agregar que el sistema está dirigido a desarrolladores de software por lo tanto el desarrollador se define como actor que hace uso del sistema, para luego utilizar el código generado por el mismo en algún software que necesite una máquina de estados. El sistema debe tener las siguientes características:

- Agregar un estado en la posición señalada por el desarrollador con el ratón.
- Mover estados por toda el área de dibujo.
- Agregar arcos desde un estado origen a uno destino o hacer bucles hacia un mismo estado.
- Los arcos deben tener algún algoritmo de ruteo, para darle mayor flexibilidad al desarrollador, tal como por ejemplo usar *Bendpoints* de manera que se pueda editar la posición de los arcos.
- Permitir editar las propiedades de cada uno de los componentes gráficos.
- Agregar eliminar y editar eventos.
- Crear un nuevo documento de máquina de estados.
- Cambiar los eventos de los arcos.
- Guardar un documento de máquina de estados.
- Cargar un documento ya existente de máquina de estados.

En cuanto a la generación de código el sistema debe generar una clase que maneja toda la lógica de la máquina de estados y brindar ciertos *hooks* al desarrollador, estos son métodos que se deben sobrescribir y como se muestra en el capítulo 2 sección 2.1.2 las máquinas de estados con las que se está trabajando son máquinas de estados de *Moore* y *Mealy*. Los métodos que sobrescribía el desarrollador son las salidas de la máquina de estados, dependiendo del caso puede ser una máquina de *Moore* de *Mealy* o ambas. Para poder sobrescribir estos *hooks* el desarrollador debe crear una clase donde sobrescriba todos o algunos de estos métodos. Como se puede observar en las figuras 3.1 y 3.2, todo lo explicado anteriormente está representado con un diagrama de casos de uso donde el actor es el desarrollador.

3.4 Casos de Uso





3.4.1 Crear Documento de máquina de estados CU-01

Crear Documento de máquina de estados	
Numero	01
Descripción	Permite crear un nuevo documento de máquina de estados.
Flujo Normal	<ul style="list-style-type: none"> El desarrollador desplaza el ratón hacia la barra de herramientas superior, selecciona la opción archivo → nuevo archivo → documento de máquina de estados, y coloca el nombre de su preferencia. El sistema crea un nuevo archivo de máquina de estados, con el nombre especificado por el desarrollador.
Actores	Desarrollador.
Poscondiciones	Se crea un nuevo documento de máquina de estados.

Tabla 3. 1. CU-01.

Diagrama de Casos de Uso

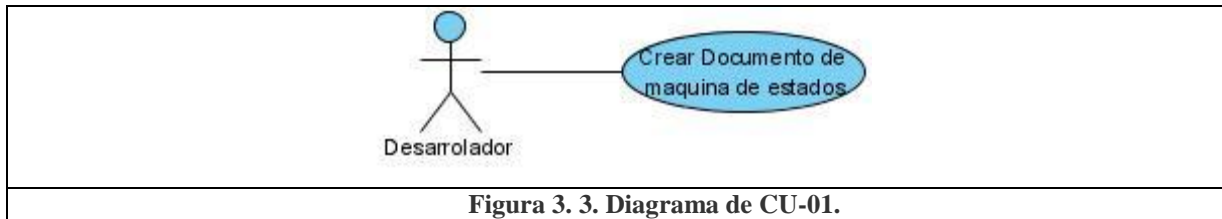
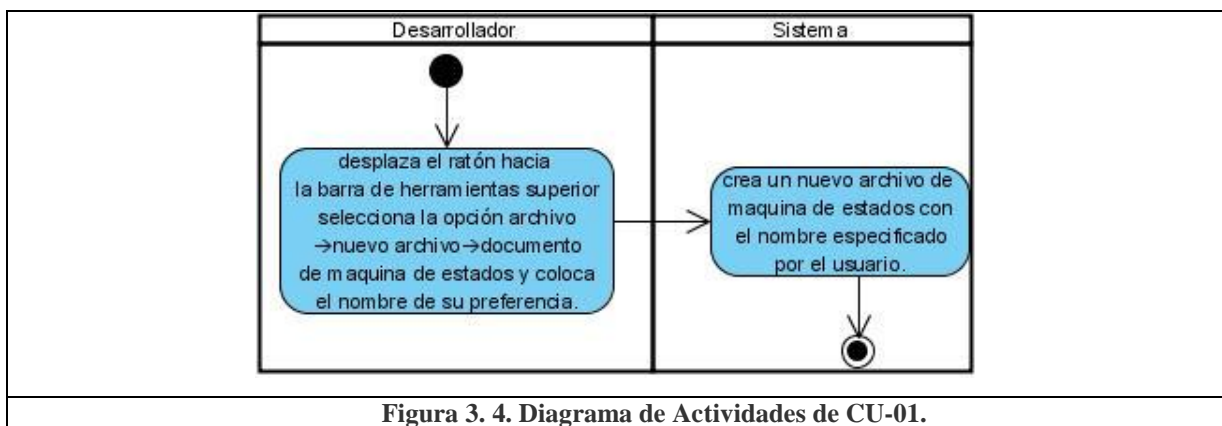


Diagrama de Actividades



3.4.2 Cargar documento de máquina de estados CU-02

Cargar documento de máquina de estados	
Numero	02
Descripción	Permite cargar un documento de máquina de estados.
Flujo Normal	<ul style="list-style-type: none"> El desarrollador hacer doble click en algún documento de máquina de estados. El sistema detecta el documento seleccionado por el desarrollador, y muestra en el <i>Canvas</i> todos los elementos existente dicho documento.
Actores	Desarrollador.
Precondiciones	Debe existir un documento de máquina de estados.
Poscondiciones	El sistema una máquina de estados en el <i>Canvas</i> .

Tabla 3. 2. CU-02.

Diagrama de Casos de Uso

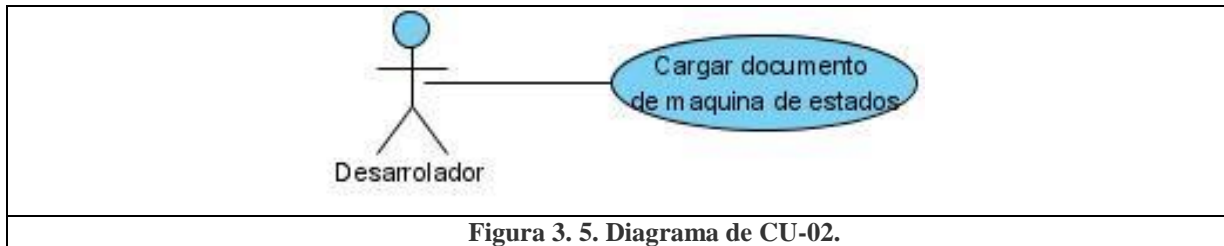
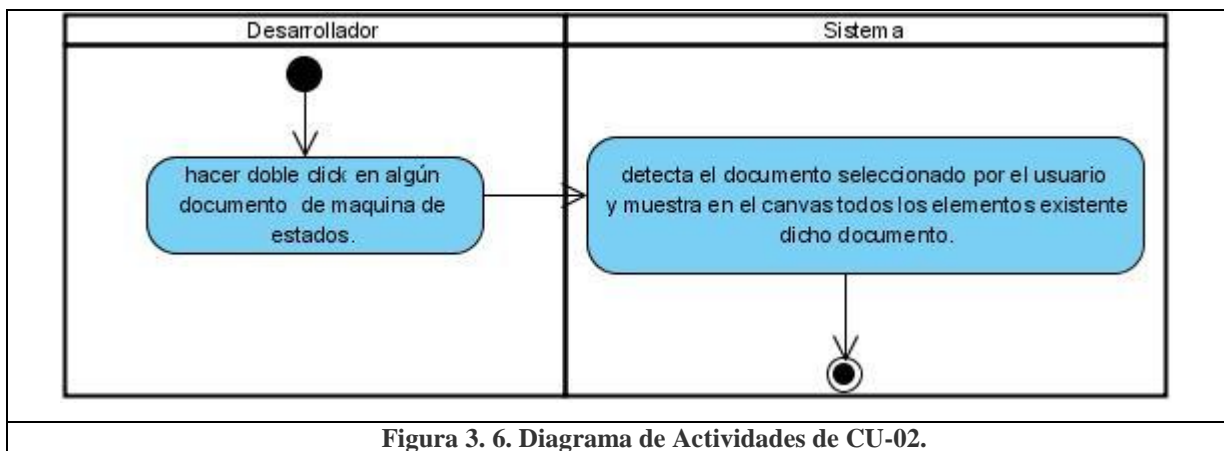


Diagrama de Actividades



3.4.3 Guardar documento de máquina de estados CU-03

Guardar documento de máquina de estados	
Numero	03
Descripción	Permite guardar un documento de máquina de estados.
Flujo Normal	<ul style="list-style-type: none"> El desarrollador realiza un cambio. El sistema detecta el cambio realizado por el desarrollador. El desarrollador selecciona la opción de guardar en la barra de herramientas mostrada por el sistema. El sistema guarda los cambios realizados.
Actores	Desarrollador.
Precondiciones	Debe existir un documento de máquina de estados abierto y que el desarrollador realice algún cambio.
Poscondiciones	El sistema guarda los cambios realizados en el documento seleccionado por el desarrollador.

Tabla 3. 3. CU-03.

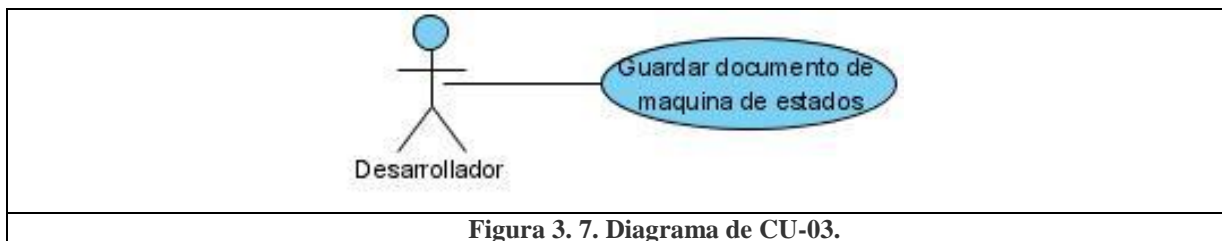
Diagrama de Casos de Uso

Figura 3. 7. Diagrama de CU-03.

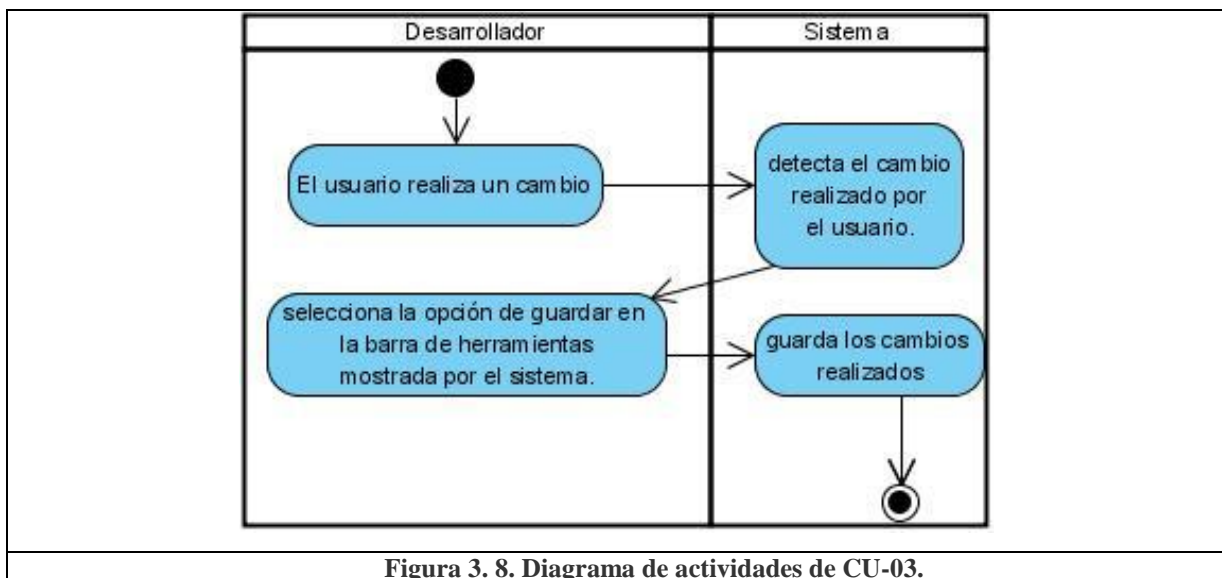
Diagrama de Actividades

Figura 3. 8. Diagrama de actividades de CU-03.

3.4.4 Agregar estado CU-04

Agregar Estados al Editor	
Numero	04
Descripción	Permite al desarrollador a través de una barra de herramientas agregar estados al Editor.
Flujo Normal	<ul style="list-style-type: none"> • El desarrollador hace click en la barra de herramientas, sobre la herramienta de añadir un estado. • El sistema selecciona la herramienta de añadir estado como la herramienta activa. • El desarrollador hace click sobre un lugar en el <i>Canvas</i>. • El sistema añade un estado, posicionado en el lugar sobre el cual el desarrollador hizo click.
Actores	Desarrollador.
Precondiciones	Se debe tener un documento de máquina de estados abierto para que se muestre la barra de herramientas.
Poscondiciones	Se dibuja un nuevo estado en el <i>Canvas</i> .

Tabla 3. 4. CU-04.

Diagrama de Casos de Uso

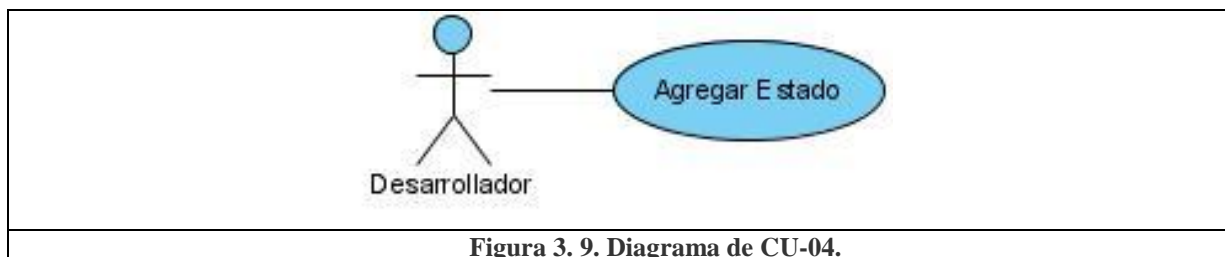
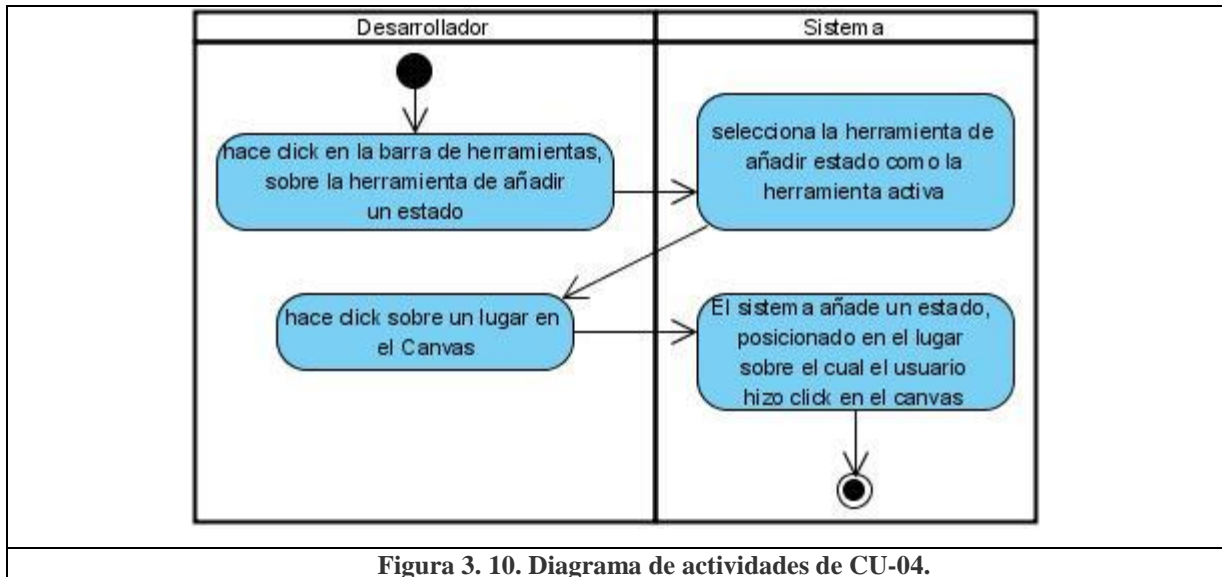


Figura 3. 9. Diagrama de CU-04.

Diagrama de Actividades



3.4.5 Eliminar estado CU-05

Eliminar estado	
Numero	05
Descripción	Permite al desarrollador eliminar un estado.
Flujo Normal	<ul style="list-style-type: none"> • El desarrollador hace click sobre un estado dibujado en el <i>Canvas</i>. • El sistema detecta la selección. • El desarrollador presiona suprimir. • El sistema borra el estado del <i>Canvas</i>.
Actores	Desarrollador.
Precondiciones	Se debe tener un documento de máquina de estados abierto y debe existir un estado dibujado en el <i>Canvas</i> .
Poscondiciones	Se elimina un estado del <i>Canvas</i> .

Tabla 3. 5. CU-05.

Diagrama de Casos de Uso

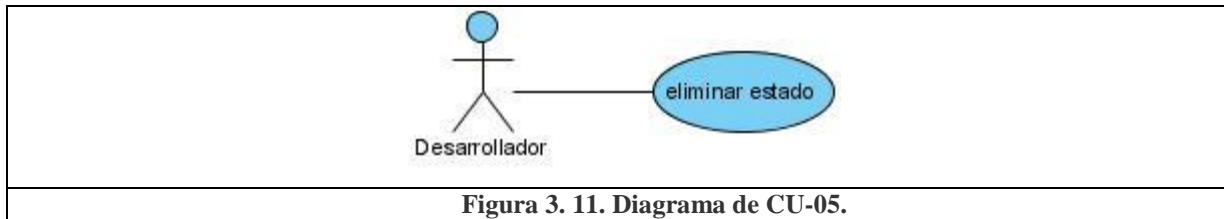
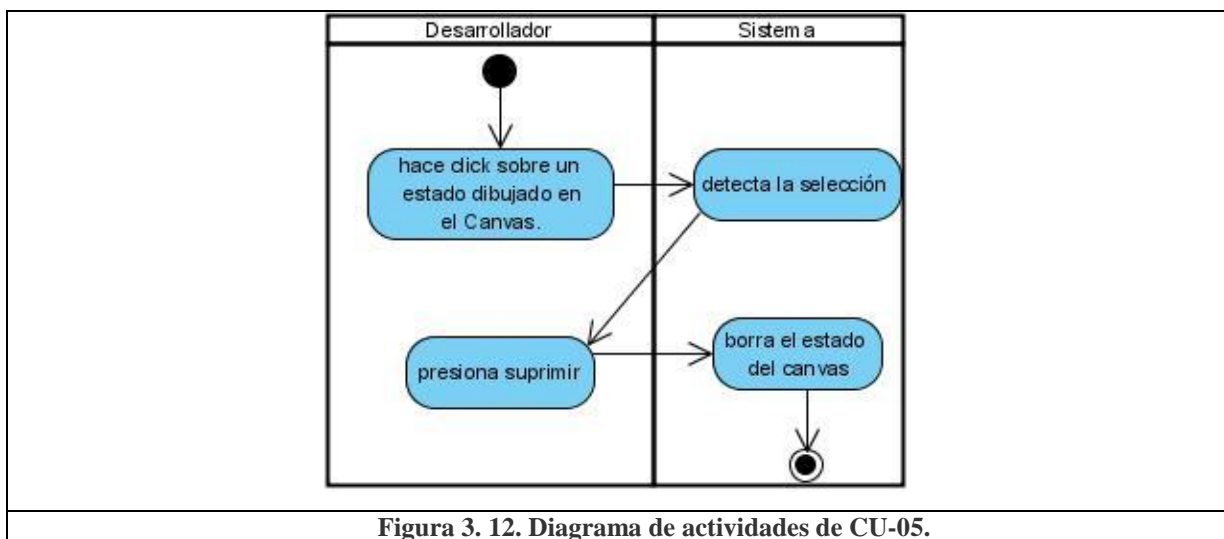


Diagrama de Actividades



3.4.6 Cambiar tipo de estado CU-06

Cambiar el tipo de un estado	
Numero	6
Descripción	Permite al desarrollador cambiar el tipo de un estado ya sea a normal inicial o final.
Flujo Normal	<ul style="list-style-type: none"> El desarrollador selecciona un estado. El sistema muestra una vista con los posibles tipos de estados. El desarrollador selecciona el tipo de estado de su preferencia. El sistema verifica el tipo de estado, si es normal lo pinta de blanco, si es inicial lo pinta de negro, y si es final lo pinta de rojo.
Actores	Desarrollador.
Precondiciones	Debe existir algún documento de máquina de estados y un estado dibujado en el <i>Canvas</i> .
Poscondiciones	El sistema pinta un estado de rojo negro o blanco dependiendo de la selección del desarrollador.

Tabla 3. 6. CU-06.

Diagrama de Casos de Uso

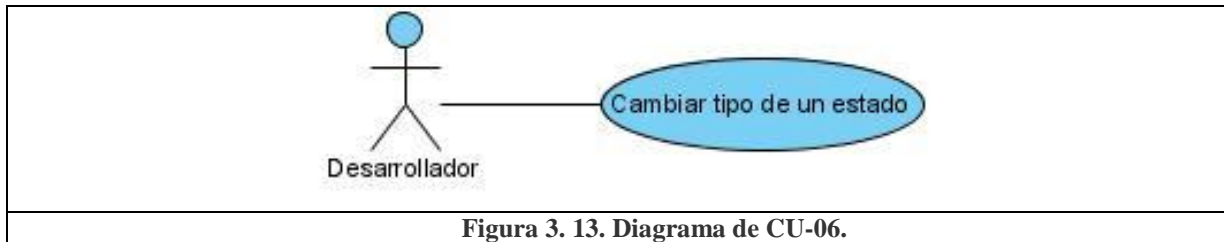


Figura 3. 13. Diagrama de CU-06.

Diagrama de Actividades

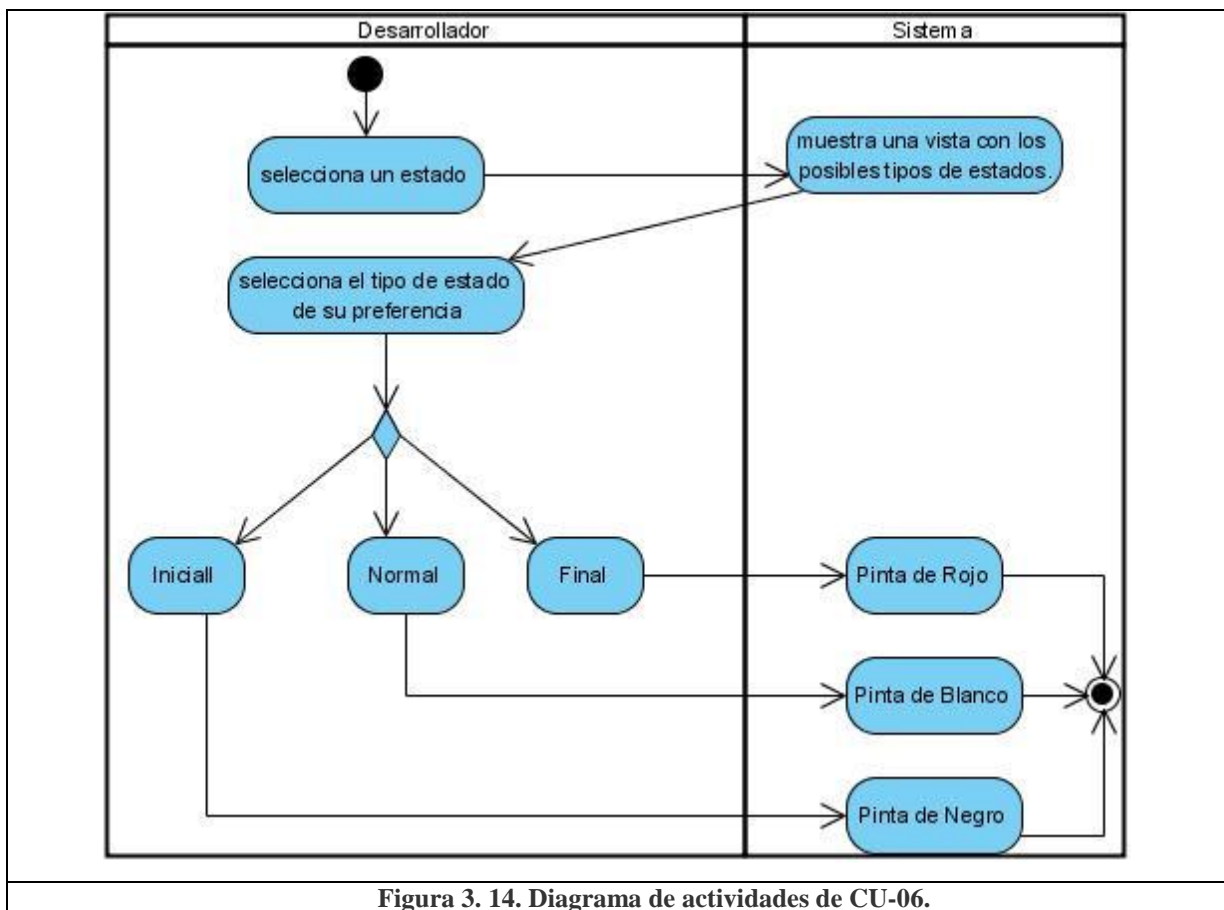


Figura 3. 14. Diagrama de actividades de CU-06.

3.4.7 Agrega arco CU-07

Agregar arco	
Numero	07

Descripción	Permite al desarrollador agregar arcos nuevos al Editor.
Flujo Normal	<ul style="list-style-type: none"> • El desarrollador hace click en la barra de herramientas, sobre la herramienta de añadir un arco. • El sistema selecciona la herramienta de añadir estado como la herramienta activa. • El desarrollador selecciona en el <i>Canvas</i> un estado origen y uno destino, pudiendo ser el destino el mismo estado origen. • El sistema agrega un nuevo arco en el <i>Canvas</i>, entre los estados origen y destino seleccionados por el desarrollador.
Actores	Desarrollador.
Precondiciones	Debe estar desplegada la barra de herramientas y debe existir un estado en el <i>Canvas</i> .
Poscondiciones	Se dibuja un nuevo arco en el <i>Canvas</i> .

Tabla 3. 7. CU-07.

Diagrama de Casos de Uso

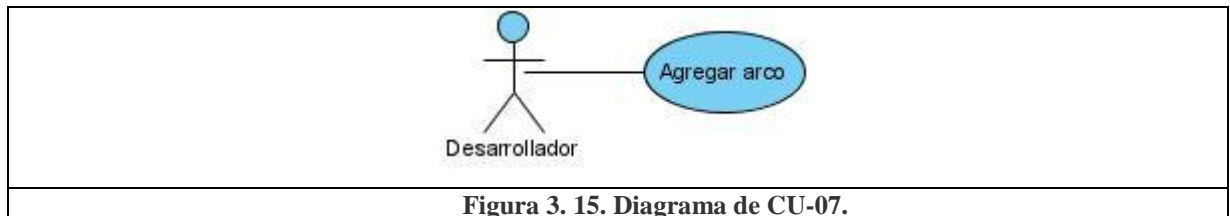


Figura 3. 15. Diagrama de CU-07.

Diagrama de Actividades

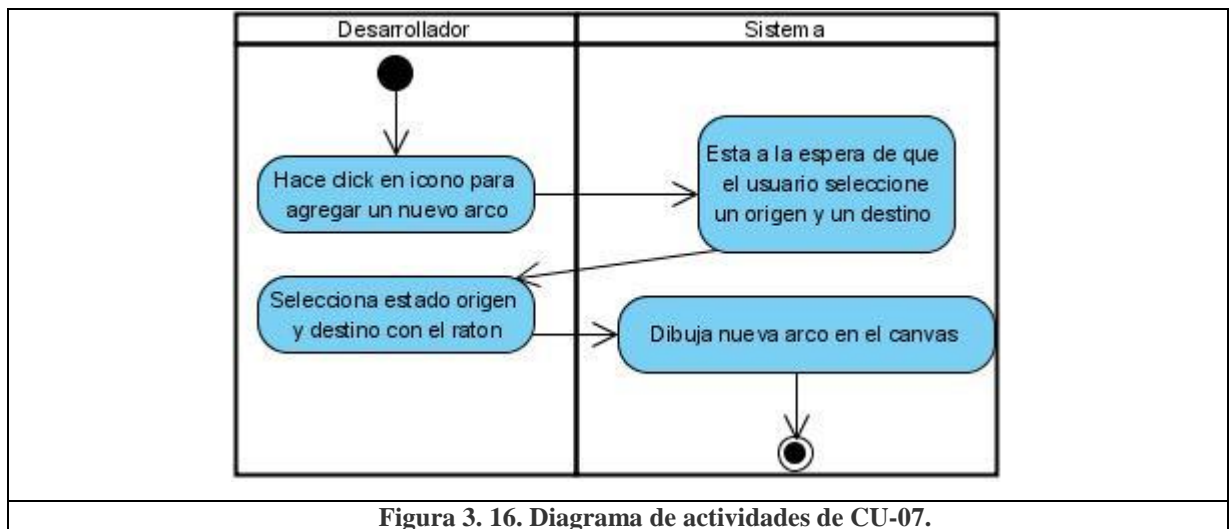


Figura 3. 16. Diagrama de actividades de CU-07.

3.4.8 Eliminar arco CU-08

Eliminar arco	
Numero	08
Descripción	Permite al desarrollador eliminar un arco.
Flujo Normal	<ul style="list-style-type: none"> • El desarrollador sobre un arco pintado en el <i>Canvas</i>. • El sistema detecta la selección. • El desarrollador presiona suprimir. • El sistema borra al arco del <i>Canvas</i>.
Actores	Desarrollador.
Precondiciones	Debe existir un arco pintado en el <i>Canvas</i> .
Poscondiciones	Se elimina un arco del <i>Canvas</i> .

Tabla 3. 8. CU-08.

Diagrama de Casos de Uso

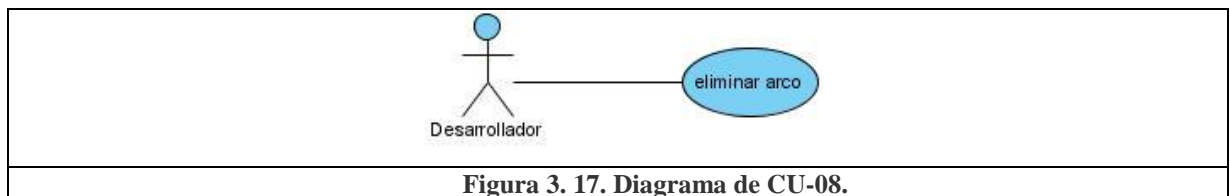
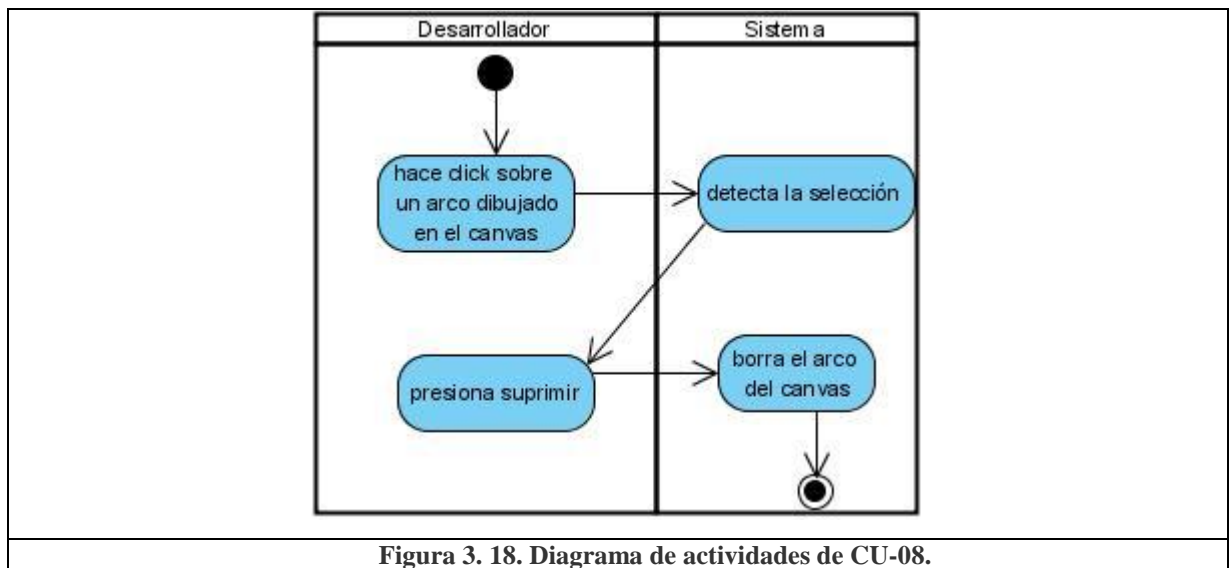


Diagrama de Actividades



3.4.9 Editar nombre de un elemento CU-09

Eliminar arco	
Numero	09
Descripción	Permite al desarrollador cambiar el nombre de algún elemento del editor.
Flujo Normal	<ul style="list-style-type: none"> El desarrollador selecciona un elemento que este dibujado en el <i>Canvas</i>. El sistema detecta que un elemento fue seleccionado, y muestra en una vista el nombre del elemento seleccionado. El desarrollador desplaza el ratón, hacia la vista desplegada y edita el nombre del elemento seleccionado.
Actores	Desarrollador.
Precondiciones	Se debe tener un documento de máquina de estados abierto y debe existir un elemento en el <i>Canvas</i> .
Poscondiciones	Se cambia el nombre del elemento seleccionado.

Tabla. 3. 9 CU-09.

Diagrama de Casos de Uso

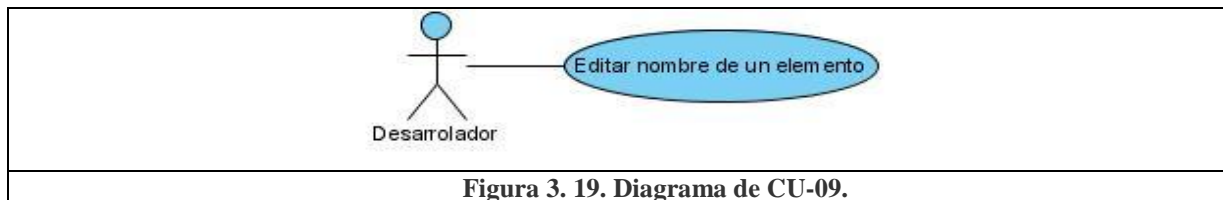


Figura 3. 19. Diagrama de CU-09.

Diagrama de Actividades

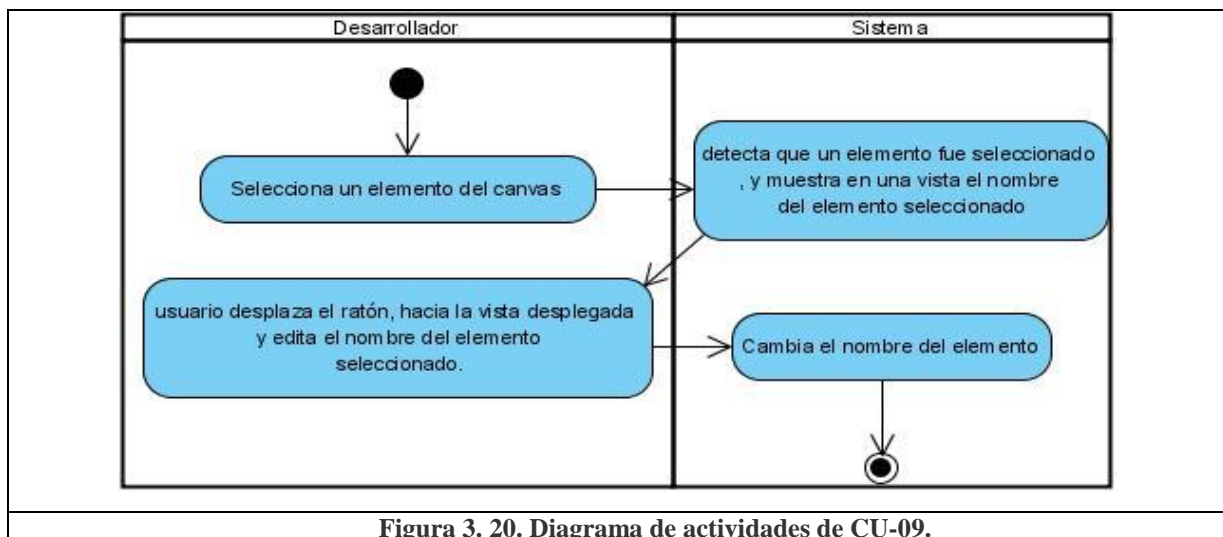


Figura 3. 20. Diagrama de actividades de CU-09.

3.4.10 Agregar evento a la máquina de estados CU-10

Agregar evento a la máquina de estados	
Numero	10
Descripción	Permite agregar eventos a la máquina de estados.
Flujo Normal	<ul style="list-style-type: none"> • El desarrollador abre el dialogo de eventos de la máquina de estados. • El sistema muestra el dialogo de eventos. • El desarrollador hace click en el botón de agregar un evento. • El sistema añade un nuevo evento a su lista de eventos con un nombre por defecto. • El desarrollador presiona el botón aceptar. • El sistema guardar los cambios.
Flujo Alternativo	<ul style="list-style-type: none"> • El desarrollador hace click en el botón de cancelar. • El sistema no realiza los cambios.
Actores	Desarrollador.
Precondiciones	Debe estar abierto un documento de máquina de estados.
Poscondiciones	Se agrega uno o más eventos de la máquina de estados.

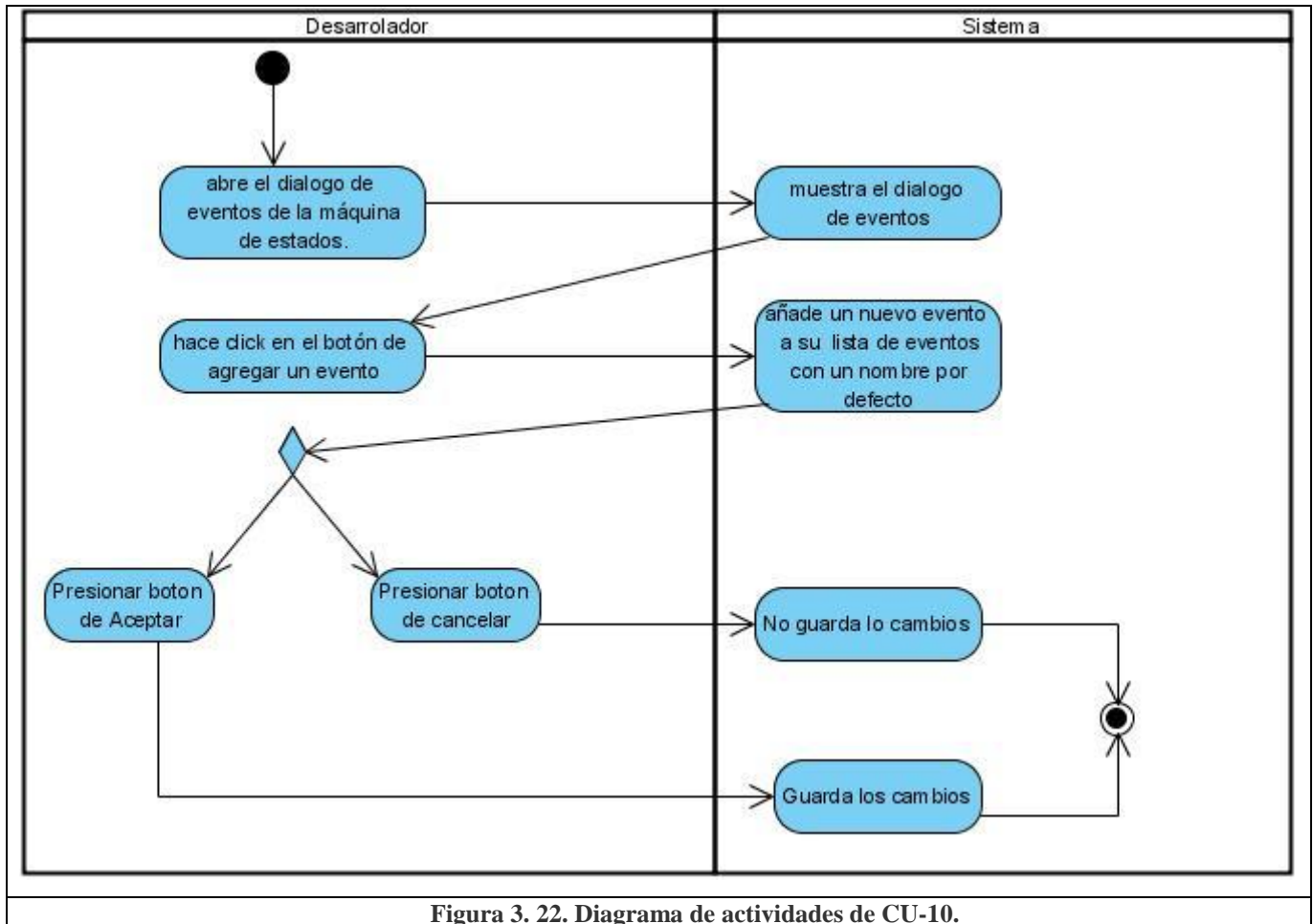
Tabla 3. 10. CU-10.

Diagrama de Casos de Uso



Figura 3. 21. Diagrama de CU-10.

Diagrama de Actividades



3.4.11 Eliminar Evento a la máquina de estados CU-11

Eliminar evento de máquina de estados	
Numero	11
Descripción	Permite Eliminar eventos de la máquina de estados.
Flujo Normal	<ul style="list-style-type: none"> El desarrollador abre el dialogo de eventos de la máquina de estados. El sistema lista en ese dialogo, todos los eventos de la máquina de estados. El desarrollador selecciona uno de los eventos para eliminarlo. El sistema detecta la selección.

	<ul style="list-style-type: none"> • El desarrollador hace click en el botón eliminar. • El sistema elimina el evento de su lista de eventos. • El desarrollador hace click en aceptar. • El sistema guarda los cambios.
Flujo Alternativo	<ul style="list-style-type: none"> • El desarrollador hace click en el botón de cancelar. • El sistema no realiza los cambios.
Actores	Desarrollador
Precondiciones	Se debe tener un documento de máquina de estados abierto y debe existir algún evento añadido a la máquina de estados.
Poscondiciones	Se elimina un evento de la máquina de estados.

Tabla. 3. 11 CU-11.

Diagrama de Casos Uso

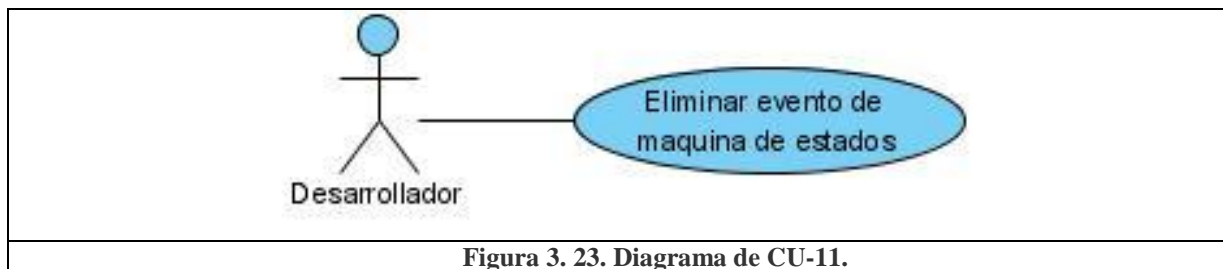
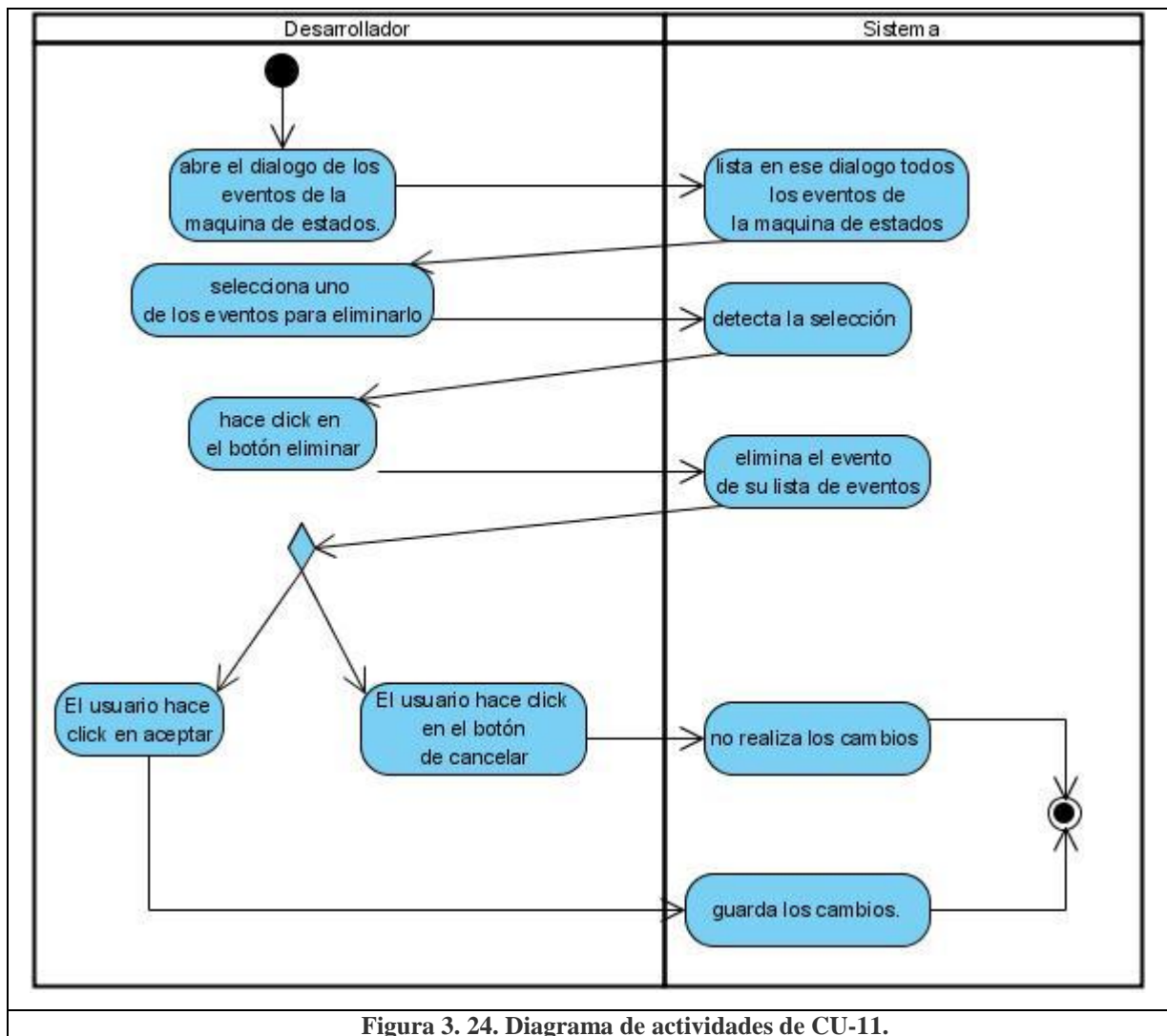


Figura 3. 23. Diagrama de CU-11.

Diagrama de Actividades



3.4.12 Editar evento de máquina de estados CU-12

Editar evento de máquina de estados	
Numero	12
Descripción	Permite Editar eventos de la máquina de estados.

Flujo Normal	<ul style="list-style-type: none"> • El desarrollador abre el dialogo de eventos de la máquina de estados. • El sistema lista en ese dialogo todos los eventos de la máquina de estados. • El desarrollador selecciona uno de los eventos para editarlo. • El sistema detecta la selección. • El desarrollador edita el nombre del evento con el teclado. • El sistema cambia el nombre del evento. • El desarrollador hace click en aceptar. • El sistema guarda los cambios.
Flujo Alternativo	<ul style="list-style-type: none"> • El desarrollador hace click en el botón de cancelar. • El sistema no realiza los cambios.
Actores	Desarrollador.
Precondiciones	Se debe tener un documento de máquina de estados abierto y debe existir algún evento añadido a la máquina de estados.
Poscondiciones	Se edita un evento de la máquina de estados.

Tabla. 3. 12 CU-12.

Diagrama de Casos de Uso

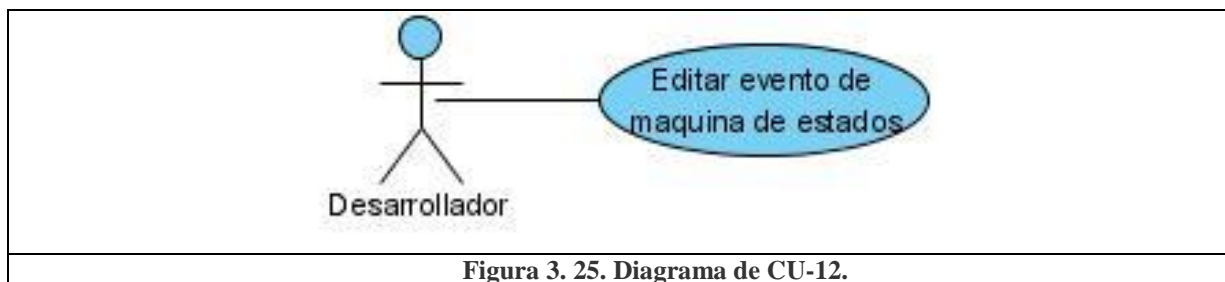


Figura 3. 25. Diagrama de CU-12.

Diagrama de Actividades

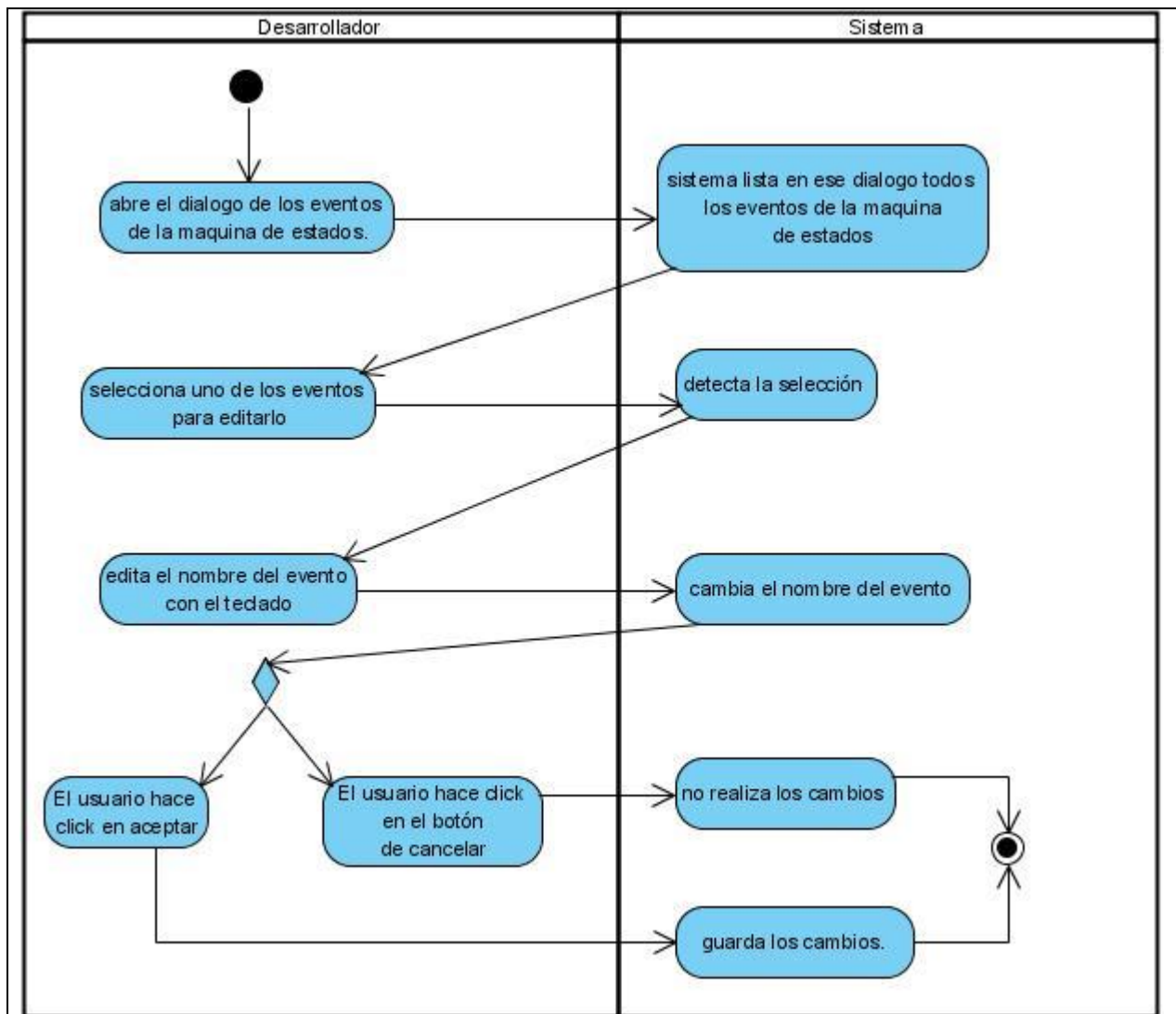


Figura 3. 26. Diagrama de CU-12.

3.4.13 Cambiar evento de un arco CU-13

Cambiar evento de un arco	
Numero	13
Descripción	Permite Cambiar eventos a los arcos.

Flujo Normal	<ul style="list-style-type: none"> • El desarrollador selecciona un arco. • El sistema detecta la selección, y muestra una vista donde lista todos los eventos de la máquina de estados. • El desarrollador selecciona un evento de la vista desplegada por el sistema. • El sistema verifica que no exista otro evento igual para el estado origen, y pinta todos los arcos de negro. • El sistema pinta el evento seleccionado sobre el <i>Canvas</i>.
Flujo Alternativo	<ul style="list-style-type: none"> • El desarrollador coloca un evento que existe en otro arco para el estado origen. • El sistema pinta de rojo todos los arcos para ese estado origen.
Actores	Desarrollador
Precondiciones	Debe estar abierto un documento de máquina de estados y deben existir en el al menos un estado y una transición.
Poscondiciones	El sistema cambia el evento asociado a esa transición y lo muestra en el <i>Canvas</i> .

Tabla 3. 13. CU-13.

Diagrama de Casos de Uso

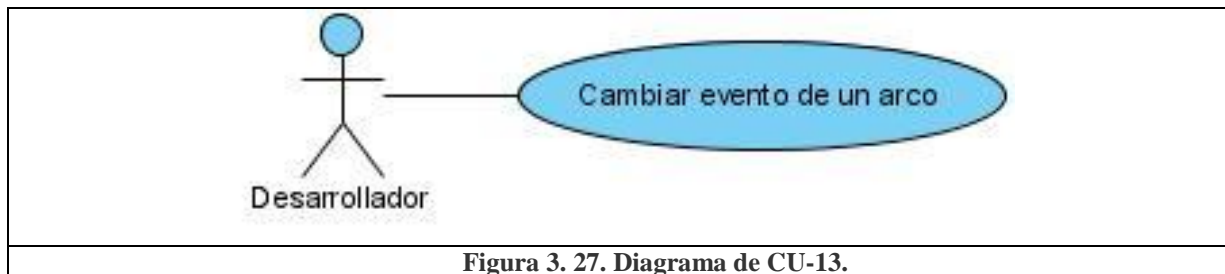
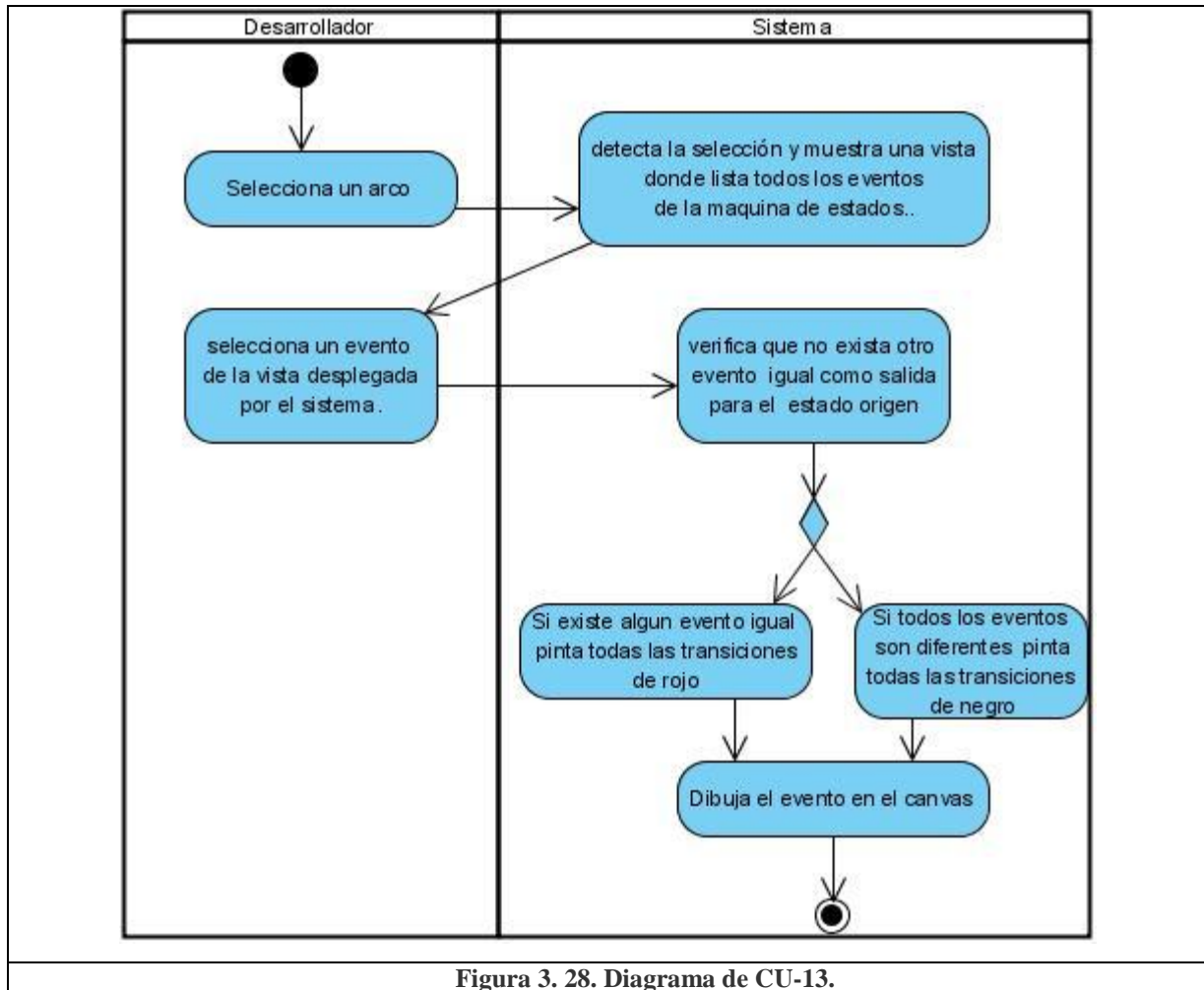


Figura 3. 27. Diagrama de CU-13.

Diagrama de Actividades



3.4.14 Seleccionar ruta destino para el archivo a generar CU-14

Seleccionar ruta destino para el archivo a generar	
Numero	14
Descripción	Permite seleccionar una ruta destino
Flujo Normal	<ul style="list-style-type: none"> • El desarrollador abre el dialogo para seleccionar la ruta destino. • El sistema abre el dialogo para seleccionar la ruta destino. • El desarrollador selecciona la ruta. • El sistema guarda la ruta seleccionada por el desarrollador. • El desarrollador presiona el botón de aceptar.

	<ul style="list-style-type: none">• El sistema guarda los cambios.
Flujo Alternativo	<ul style="list-style-type: none">• El desarrollador presiona el botón de cancelar.• El sistema no guarda los cambios.
Actores	Desarrollador
Precondiciones	Debe existir algún documento de máquina de estados abierto.
Poscondiciones	El sistema guarda una ruta donde se generara el código.

Tabla 3. 14. CU-14.

Diagrama de Casos de Uso



Figura 3. 29. Diagrama de CU-14.

Diagrama de Actividades

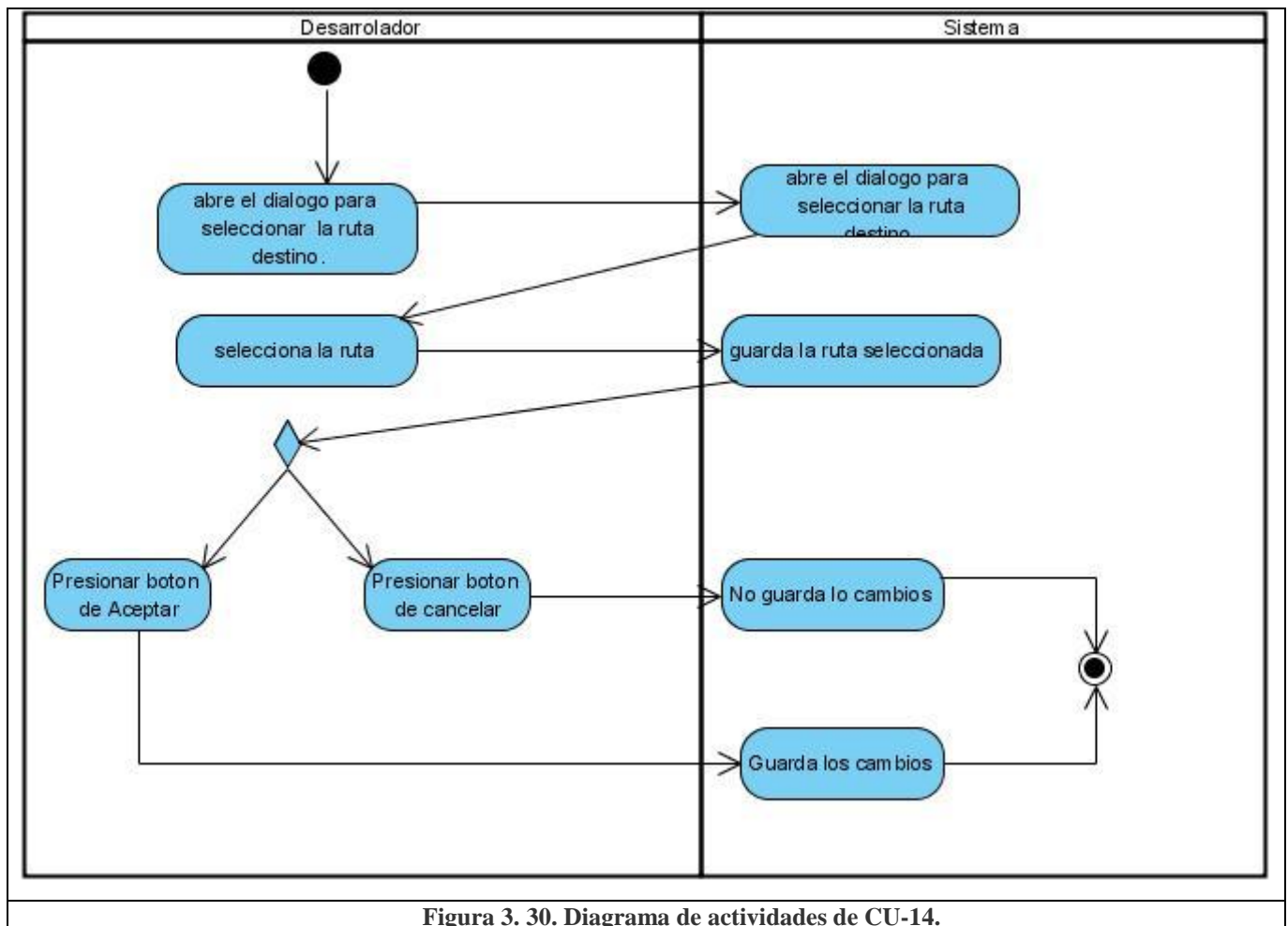


Figura 3. 30. Diagrama de actividades de CU-14.

3.4.15 Generar Código de máquina de estados CU-15

Generar Código de máquina de estados	
Numero	15
Descripción	Permite generar código de máquina de estados.
Flujo Normal	<ul style="list-style-type: none"> El desarrollador selecciona un documento de máquina de estados, y hace click derecho. El sistema muestra un menú que permite al desarrollador seleccionar la opción para generar código.

	<ul style="list-style-type: none">• El desarrollador hace click sobre la opción generar código de máquina de estados, del menú mostrado por el sistema.• El sistema genera el código de la máquina de estados seleccionada por el desarrollador.
Actores	Desarrollador
Precondiciones	Debe existir algún documento de máquina de estados.
Poscondiciones	El sistema genera el código de la máquina de estados seleccionada por el desarrollador.

Tabla 3. 15. CU-15.

Diagrama de Casos de Uso

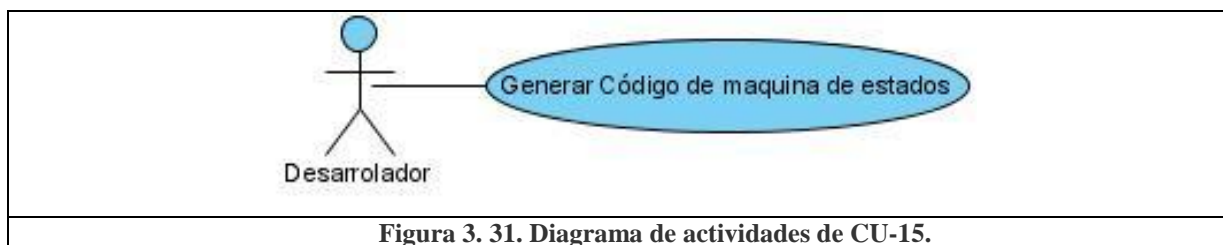
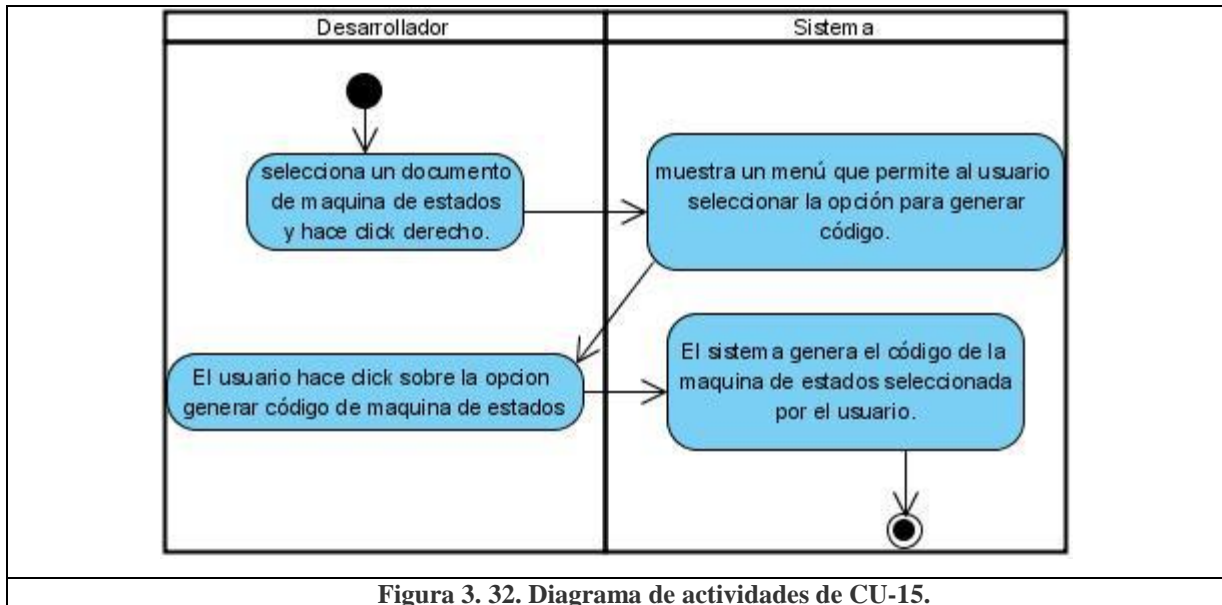


Figura 3. 31. Diagrama de actividades de CU-15.

Diagrama de Actividades



3.4.16 Seleccionar plantilla alternativa para generar código CU-16

Seleccionar plantilla alternativa para generar código	
Numero	16
Descripción	Permite seleccionar una plantilla alternativa.
Flujo Normal	<ul style="list-style-type: none"> • El desarrollador abre el dialogo para seleccionar una plantilla alternativa. • El sistema muestra el dialogo para seleccionar la plantilla. • El desarrollador selecciona la ruta de la nueva plantilla. • El sistema detecta la ruta seleccionada. • El desarrollador presiona al botón de aceptar. • El sistema guarda los cambios.
Flujo Alternativo	<ul style="list-style-type: none"> • El desarrollador presiona el botón de cancelar. • El sistema no guarda los cambios.
Actores	Desarrollador
Precondiciones	Debe existir algún documento de máquina de estados.
Poscondiciones	El sistema genera el código de la máquina de estados seleccionada por el desarrollador.

Tabla 3. 16 CU-16.

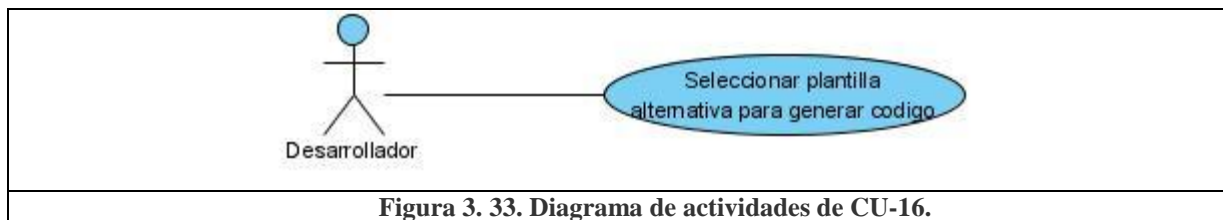
Diagrama de Casos de Uso

Figura 3. 33. Diagrama de actividades de CU-16.

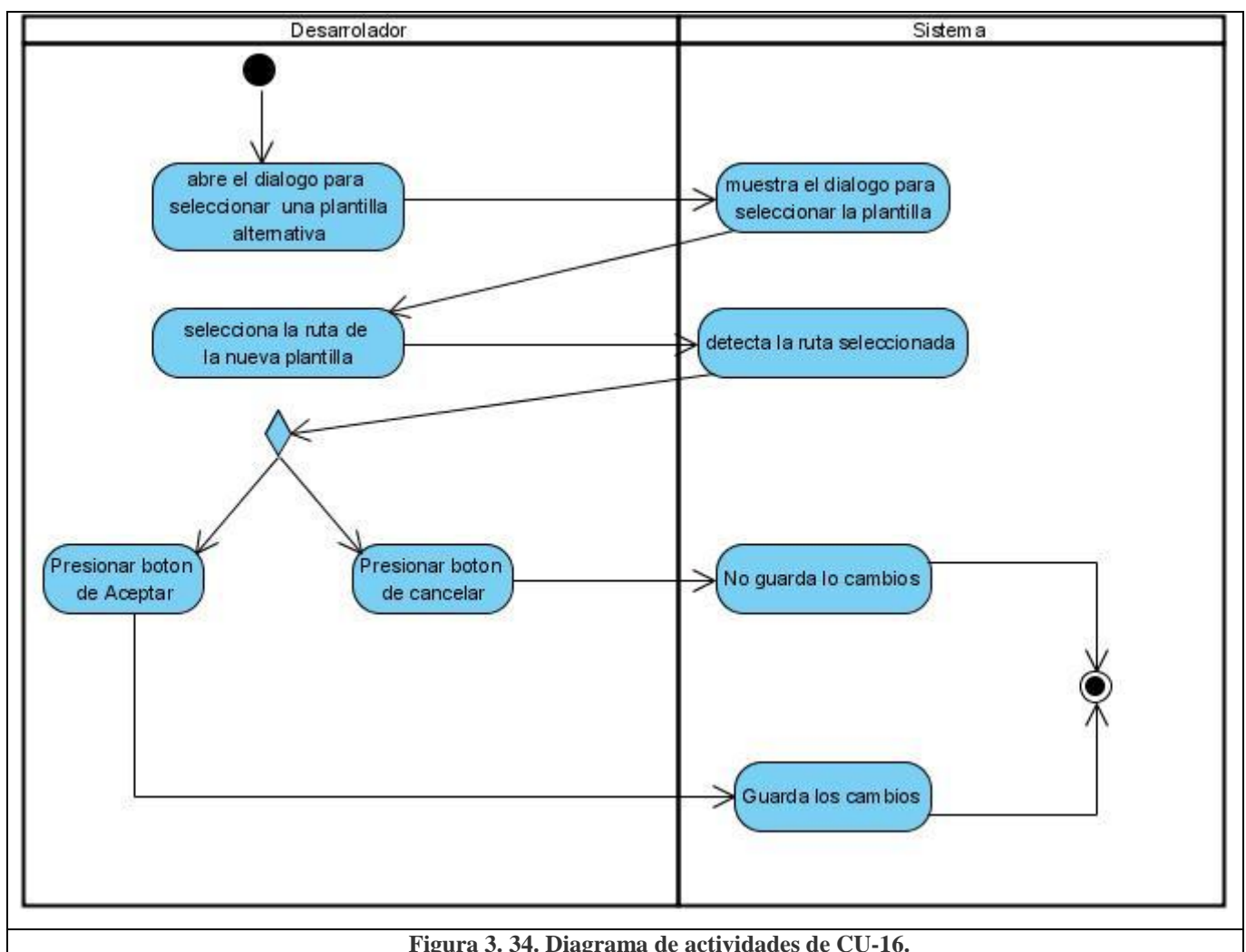
Diagrama de Actividades

Figura 3. 34. Diagrama de actividades de CU-16.

3.5 Vistas del Sistema

3.5.1 Iconos del Sistema

En esta tabla se muestra toda la notación necesaria para entender las convenciones utilizadas en este trabajo.







Notación	Nombre	Descripción
	Estado Inicial	Estado inicial de la ME, solo debe existir un solo.
	Estado Final	Estado final, puede haber uno o más estados finales.
	Estado Normal	Un estado cualquiera de la ME.
	Arco sin evento	Ocurre cuando un arco no tiene un evento asociado.
	Arco de autómata no determinista	Sucede cuando un estado puede estar en dos estados diferentes al mismo tiempo. Este editor no soporta esta propiedad.
	Arco de autómata determinista	Sucede, cuando para un estado origen existen múltiples arcos, y cada arco tiene un evento diferente.

Tabla 3. 17. Iconos del Sistema.

3.5.2 Editor

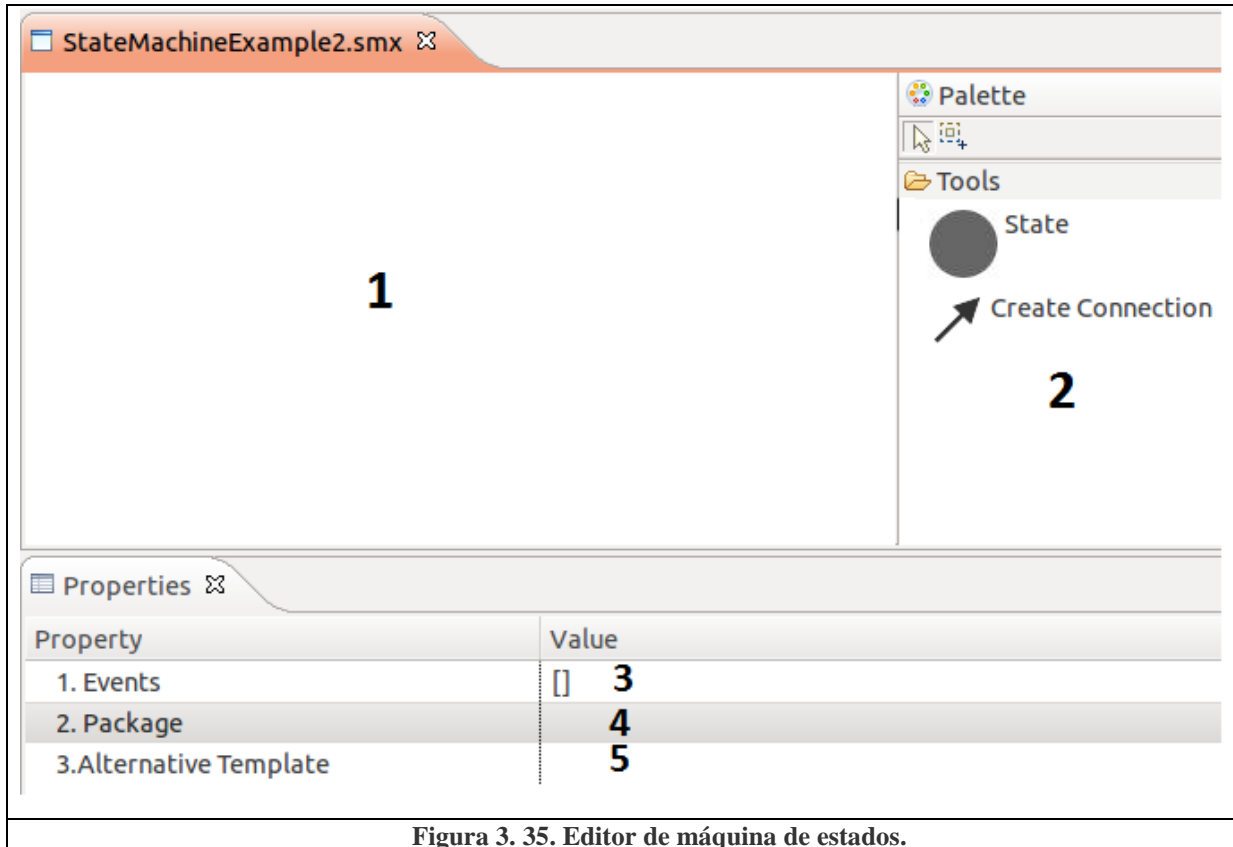


Figura 3. 35. Editor de máquina de estados.

Editor		
Numero	Descripción	Casos de Uso
1	El <i>Canvas</i> , donde el desarrollador dibuja la máquina de estados.	CU-04, CU-05, CU-06, CU-07, CU-08, CU-09, CU-13
2	Barra de herramientas lateral, que permite al desarrollador seleccionar la herramienta que desea dibujar sobre el <i>Canvas</i> .	CU-04, CU-07
3	Abre el dialogo de eventos de la máquina de estados.	CU-10, CU-11, CU-12
4	Abre el dialogo para seleccionar el paquete donde se va a generar el archivo .java.	CU-14
5	Abre el dialogo para seleccionar una plantilla alternativa.	CU-16

Tabla 3. 18. Editor.

3.5.3 Creando nuevo Documento

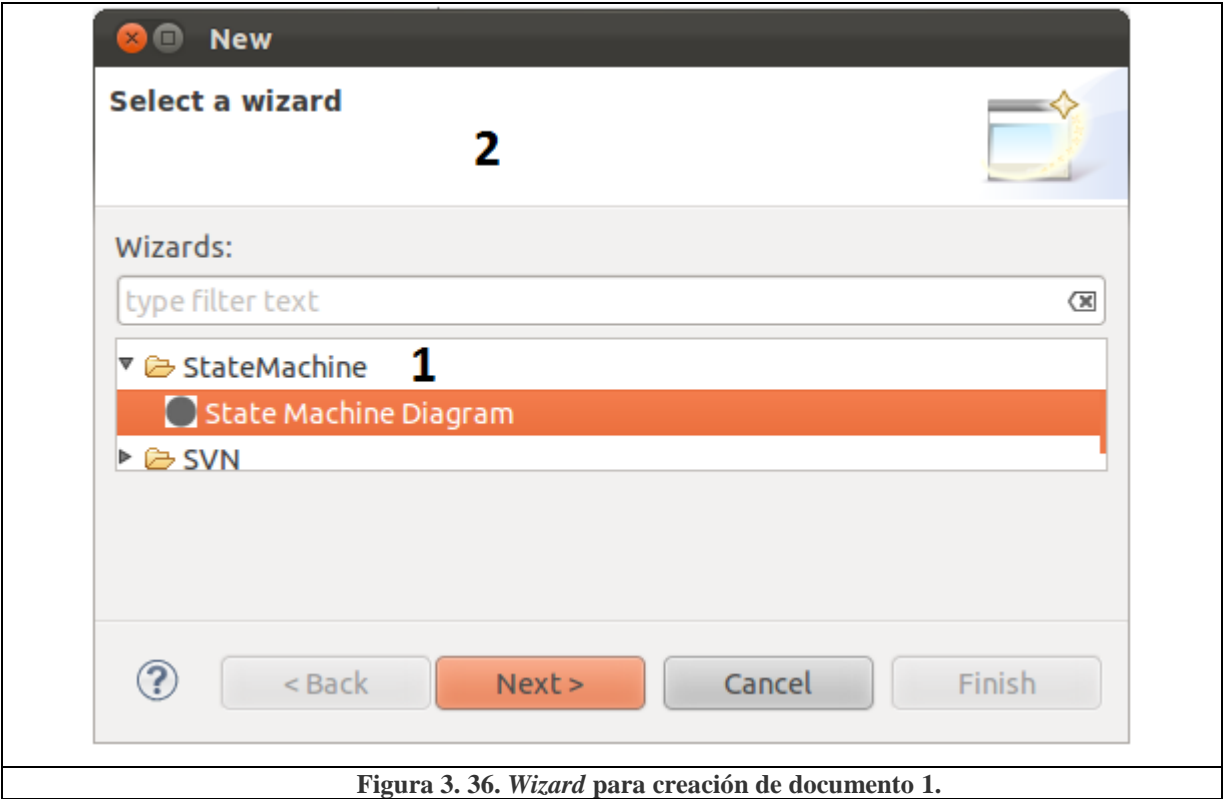


Figura 3. 36. Wizard para creación de documento 1.

Creando nuevo Documento		
Numero	Descripción	Casos de Uso
1	La pestaña <i>StateMachine</i> mostrada en el <i>wizard</i> , permite al desarrollador seleccionar un documento.	CU-01
2	El dialogo es un <i>wizard</i> que permite crea un nuevo documento.	CU-01

Tabla 3. 19. Wizard para crear nuevo documento.

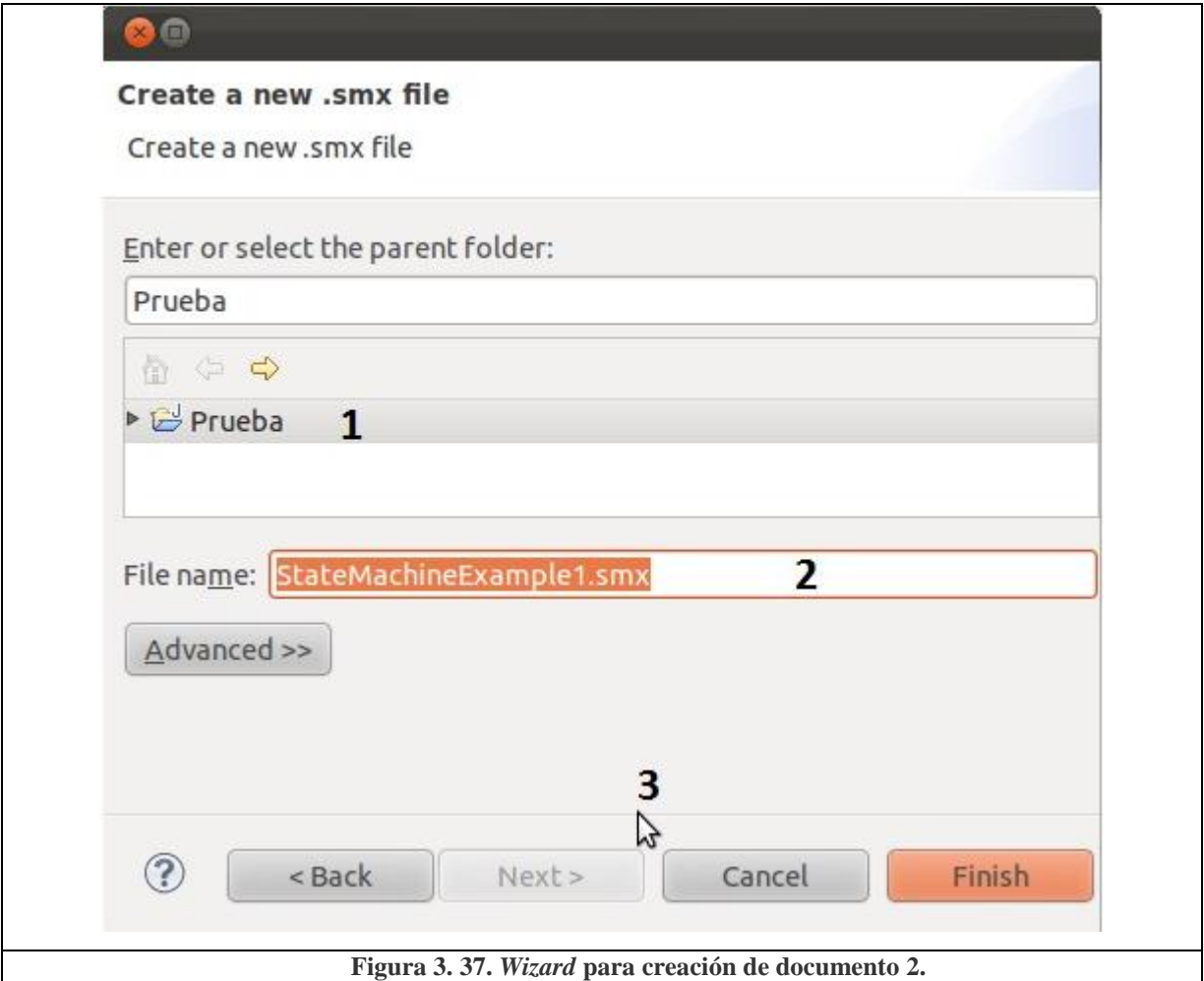


Figura 3. 37. Wizard para creación de documento 2.

Creando nuevo Documento		
Numero	Descripción	Casos de Uso
1	En esta sección el desarrollador puede seleccionar algún proyecto de todos los existentes para crear el nuevo documento.	CU-01
2	Este campo permite al desarrollador colocar el nombre de su preferencia para el nuevo documento.	CU-01
3	Los botones permiten al desarrollador finalizar la operación ir atrás o cancelar la operación.	CU-01

Tabla 3. 20. Wizard para crear nuevo Documento.

3.5.4 Dialogo de Eventos en barra de propiedades

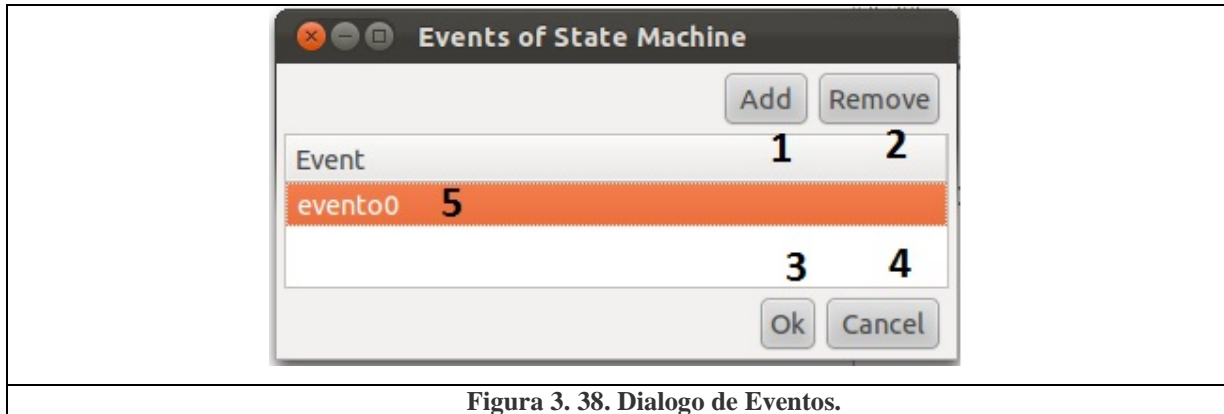


Figura 3. 38. Dialogo de Eventos.

Dialogo de Eventos en barra de propiedades		
Numero	Descripción	Casos de Uso
1	El botón <i>Add</i> permite al desarrollador añadir un nuevo evento a la lista de eventos de la máquina de estados.	CU-10
2	El botón <i>Remove</i> permite al eliminar un evento de la lista de eventos de la máquina de estados.	CU-011
3	El botón <i>Ok</i> permite al desarrollador guardar todas las modificaciones hechas en el dialogo ya sea una operación de inserción, eliminación o edición.	CU-10,CU-11,CU-12
4	El botón <i>Cancel</i> permite al desarrollador cancelar todas las operaciones hechas en el dialogo. Cuando el desarrollador vuelva a abrir el dialogo este mostrara la última versión que guardo el sistema.	CU-10,CU-11,CU-12
5	Permite al desarrollador editar un evento.	CU-12

Tabla 3. 21. Dialogo de eventos.

3.5.5 Dialogo para seleccionar una plantilla alternativa



Figura 3. 39. Dialogo para seleccionar una plantilla alternativa.

Dialogo para seleccionar una plantilla alternativa		
Numero	Descripción	Casos de Uso
1	Permite al desarrollador la ruta de la plantilla alternativa.	CU-16
2	Permite al desarrollador cancelar los cambios y mantener la plantilla por defecto.	CU-16
3	Permite al desarrollador guardar los cambios y cambiar de plantilla.	CU-16

Tabla 3. 22. Dialogo para seleccionar una plantilla alternativa.

3.5.6 Dialogo para seleccionar la ruta del archivo a generar

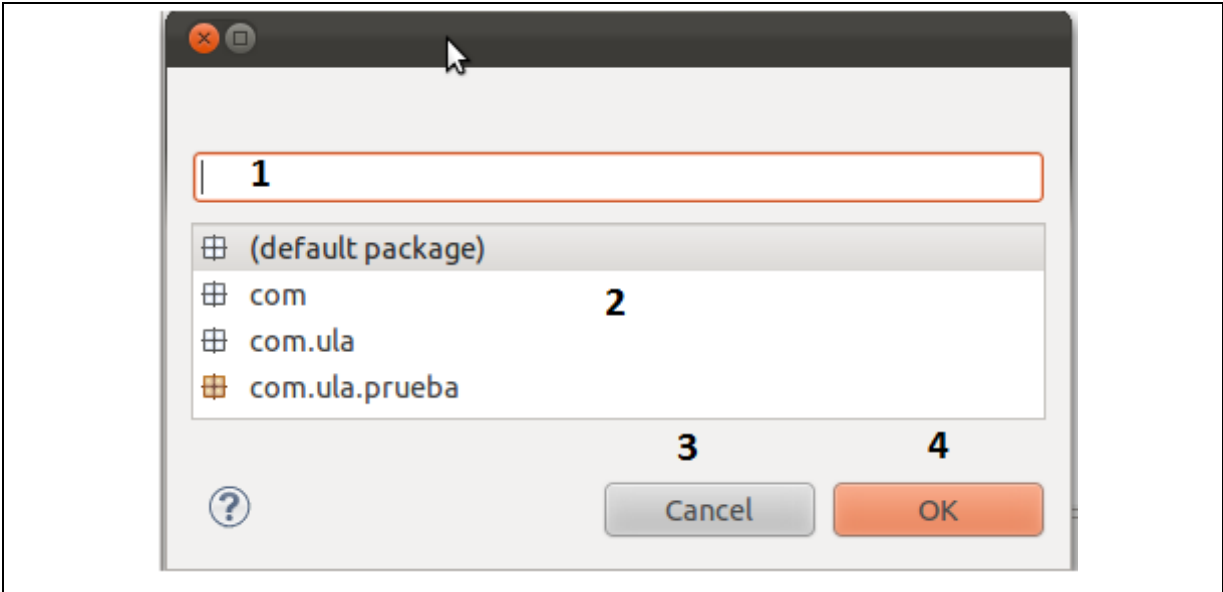
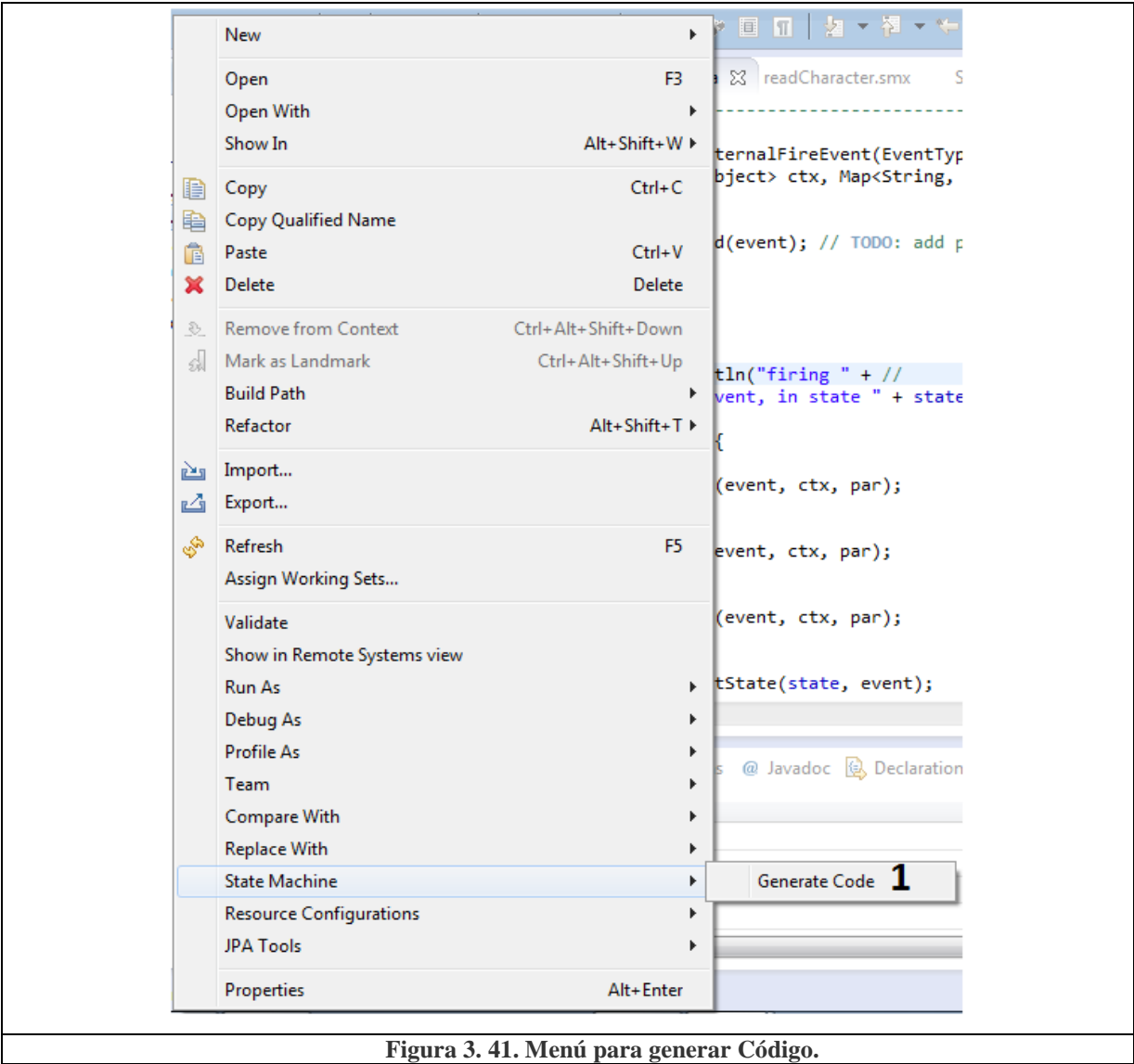


Figura 3. 40. Dialogo para seleccionar la ruta del archivo.

Dialogo para seleccionar una ruta del archivo a generar		
Numero	Descripción	Caso de Uso
1	Permite filtrar los archivos mostrados.	CU-14
2	Todos los archivos que puede seleccionar el desarrollador.	CU-14
3	Permite eliminar los cambios.	CU-14
4	Permite guardar los cambios.	CU-14

Tabla 3. 23. Dialogo para seleccionar ruta para el archivo a generar.

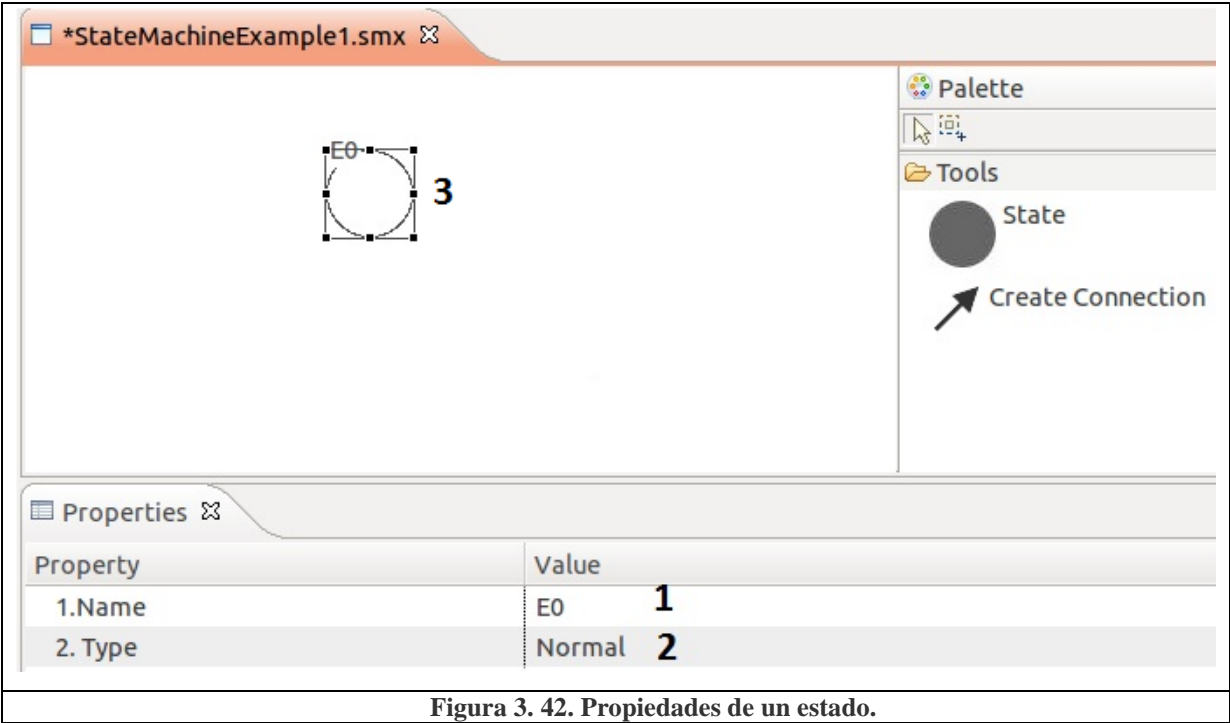
3.5.7 Menú para Generar Código



Menú para Generar código		
Numero	Descripción	Caso de Uso
1	Permite al desarrollador generar el código del documento seleccionado.	CU-15

Tabla 3. 24. Menú para generar código.

3.5.8 Propiedades y manipulación de un estado



Propiedades y manipulación de un estado		
Numero	Descripción	Caso de Uso
1	Permite al desarrollador cambiar el nombre de un estado.	CU-09
2	Permite al desarrollador seleccionar el tipo de estado que sea normal final o inicial.	CU-06
3	Con un estado seleccionado el desarrollador puede eliminarlo presionado suprimir.	CU-05

Tabla 3. 25. Propiedades de un estado.

3.5.9 Propiedades y manipulación de un arco

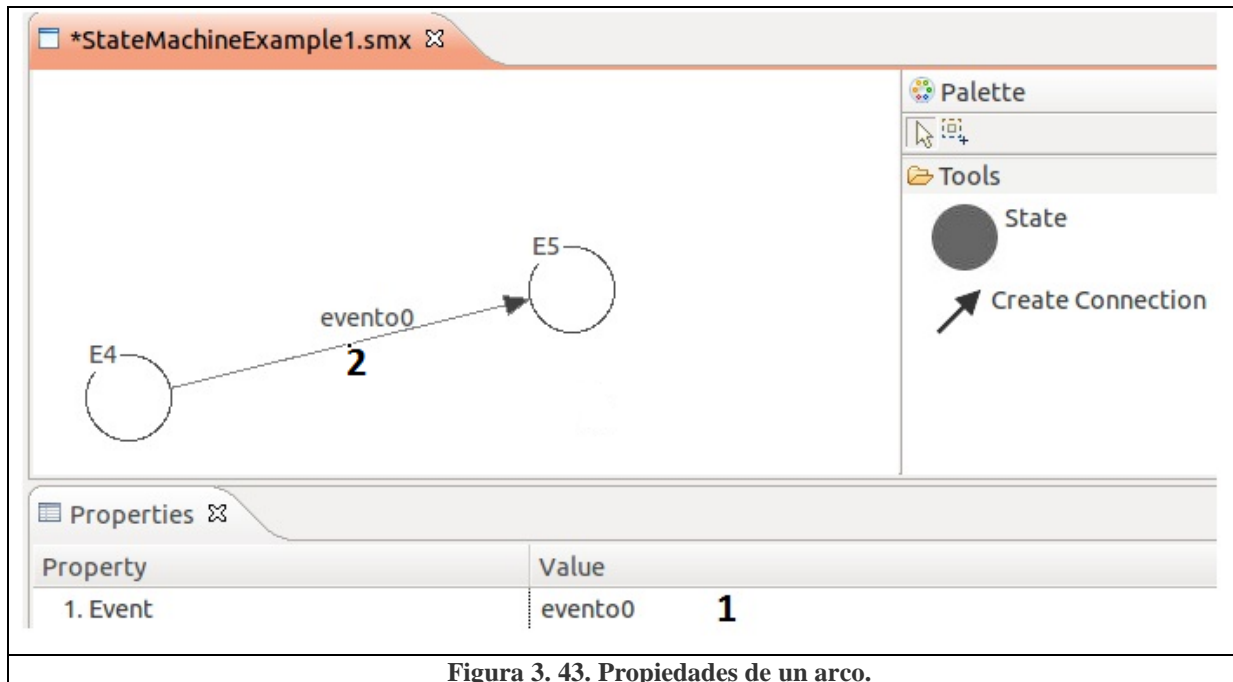


Figura 3. 43. Propiedades de un arco.

Propiedades y manipulación de un arco		
Numero	Descripción	Caso de Uso
1	Es un combo box que permite al desarrollador seleccionar de la lista de estados un evento.	CU-13
2	Con el arco seleccionado el desarrollador puede eliminarlo presionando suprimir.	CU-08

Tabla 3. 26. Propiedades de un arco.

3.5.10 Validación para Máquina de Estados Determinista

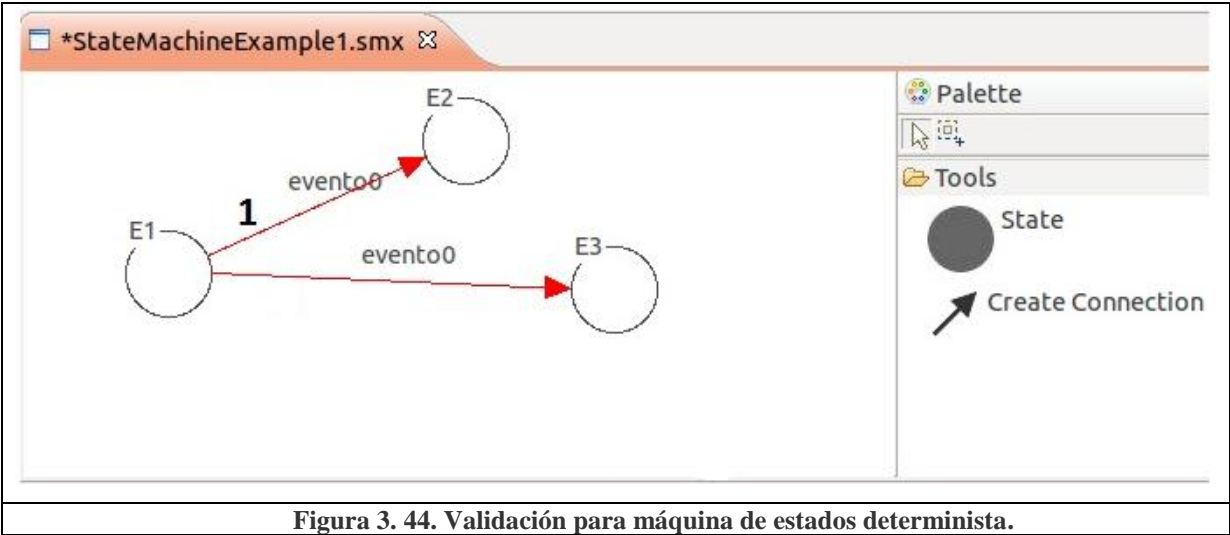


Figura 3. 44. Validación para máquina de estados determinista.

Validación para máquina de estados determinista		
Numero	Descripción	Caso de Uso
1	Arco pintada de rojo con evento0 ya existe otra transición con el mismo evento0 para el mismo estado origen.	CU-13

Tabla 3. 27. Validación para máquina de estados determinista.

Capítulo 4

Arquitectura y Desarrollo

En este capítulo se explican las tecnologías usadas y toda la arquitectura, clases y estructura del sistema utilizando diagramas de clases y diagramas bloques.

4.1 Eclipse

Es un ambiente de desarrollo integrado multilenguaje, y un sistema extensible de *Plugins*, está escrito en su mayoría en Java. Provee entornos de desarrollo para varios lenguajes de programación como lo son Java, C++, PHP, Python, Ruby, Perl, entre otros.

Eclipse dispone de un editor de texto con resaltado de sintaxis. La compilación se lleva a cabo en tiempo real. Tiene pruebas unitarias con *JUnit*, control de versiones con CVS, integración con asistentes de *wizards* para la creación de proyectos, clases test y refactorización.

Por otra parte provee *Plugins* para integración con Hibernate y otros sistemas de control de versiones como SVN y Mercurial.

La arquitectura de Eclipse está basada en *Plugins*, lo que permite a otros desarrolladores crear sus propios complementos. En la figura 4.1 tomada de Eclipse(2012), se muestran los componentes de Eclipse y cómo puede un tercero incorporar un *Plugin*.

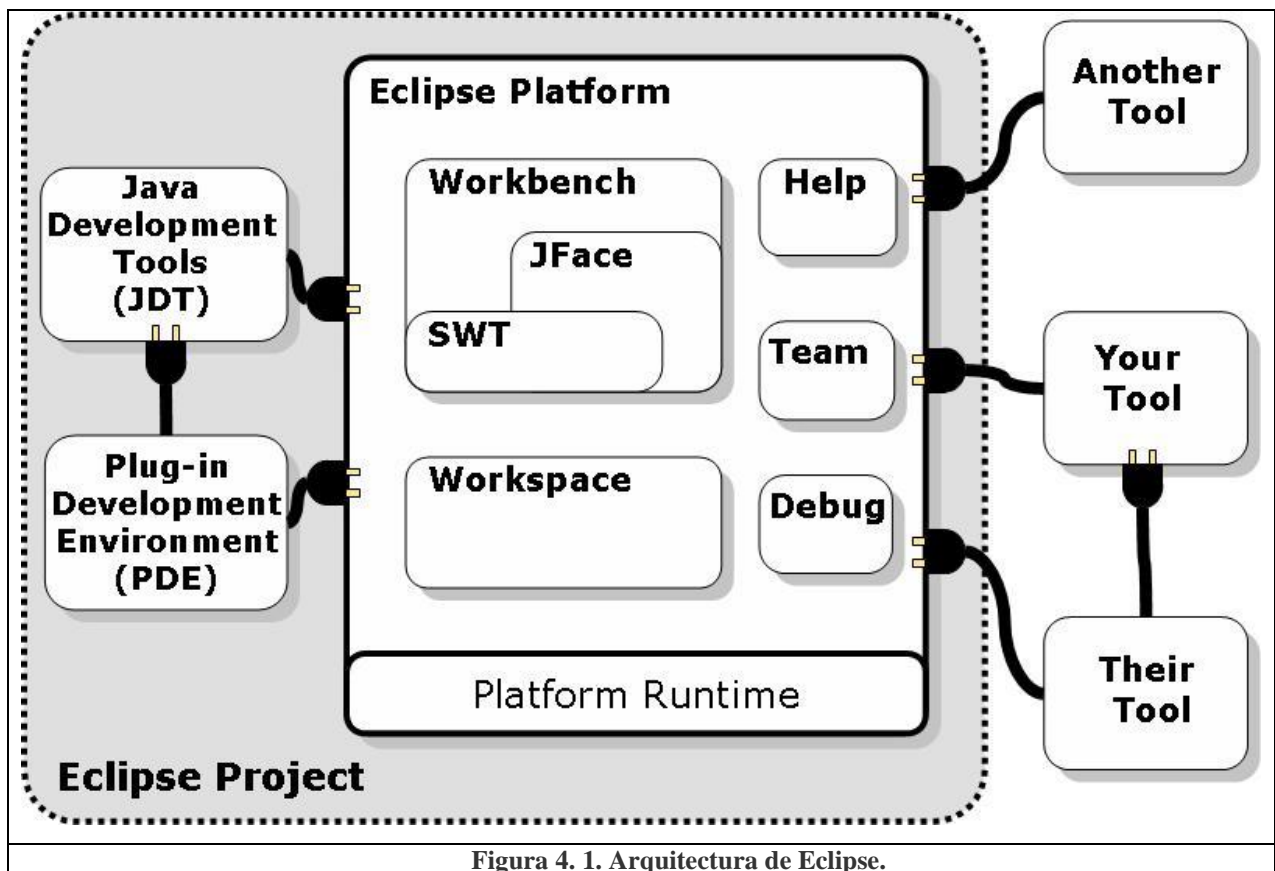


Figura 4. 1. Arquitectura de Eclipse.

El *workbench* de Eclipse, normalmente consiste de un menú principal y una barra de herramientas, así como también otros componentes como los son las vistas y los editores. La combinación de varias vistas y editores dentro del Eclipse *workbench*, son conocidas como perspectivas. Una perspectiva puede ser pensada como una página dentro del Eclipse *workbench*. Las vistas son normalmente usadas para navegar a través de los *resources* y modificar las propiedades de un *resource*. En contraste los editores son usados para modificar los *resources*. Los *resources* pueden ser cualquier tipo de archivo proyecto o carpeta. En este trabajo se utiliza el *workbench* de Eclipse para el editor de máquina de estados. (Clayberg & Rubel, 2008)

Platform Runtime, es el *kernel* que descubre al arranque del Eclipse que un *Plugin* está instalado y creado, y lleva un registro de la información de los *Plugins*. Para reducir el tiempo de arranque y los recursos usados, se encarga de que no se cargue ningún *Plugin* hasta que sea necesario. Excepto para el *Platform Runtime* todos los demás componente se implementan como un *Plugin*.

SWT es la base de toda la interfaz de usuario de Eclipse. SWT provee un conjunto muy rico de *Widgets* que pueden ser usados para crear aplicaciones de Java independientes o *plugins* de Eclipse. Se debe agregar que SWT es totalmente portable, funciona para cualquier sistema operativa sea Windows, Linux o IOS. SWT en este trabajo sirvió de utilitario para los diálogos.

4.2 Graphical Editing Framework (GEF)

GEF es el *framework* en el que se desarrolló el editor descrito en este trabajo. GEF utiliza una arquitectura MVC, la cual como se explicó en capítulo 2 sección 2.2.1 posee tres componentes: el modelo, la vista y el controlador.

En la figura 4.2 se puede observar el diagrama de secuencia que muestra las interacciones entre el modelo, la vista y el controlador en GEF. Primero se realiza una modificación al modelo (el origen de esta modificación se explica en la sección 4.2.3), esta modificación dispara un evento, que es recibido por medio de un *listener* en el *EditPart*. Luego de recibir el evento el *EditPart* hace cualquier modificación que sea necesaria en la vista para reflejar el nuevo estado del modelo.

Para que lo antes mencionado funcione, es necesario que el *EditPart* registre un *listener* para recibir las modificaciones en el modelo. Como se muestra en la figura 4.3 primero se crea un modelo, al crearse una nueva instancia de un modelo GEF transparentemente crea una instancia del *EditPart* correspondiente a dicho modelo. Por ultimo éste registra los

listeners para recibir notificaciones del modelo y construye y las vistas correspondientes que representan el modelo.

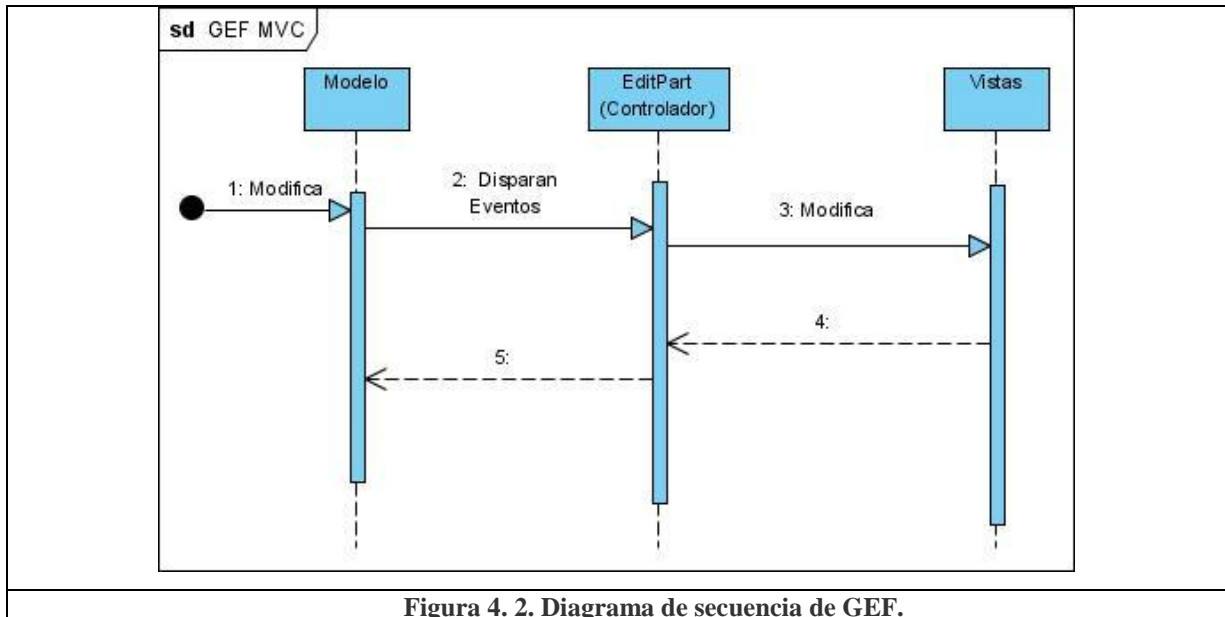


Figura 4. 2. Diagrama de secuencia de GEF.

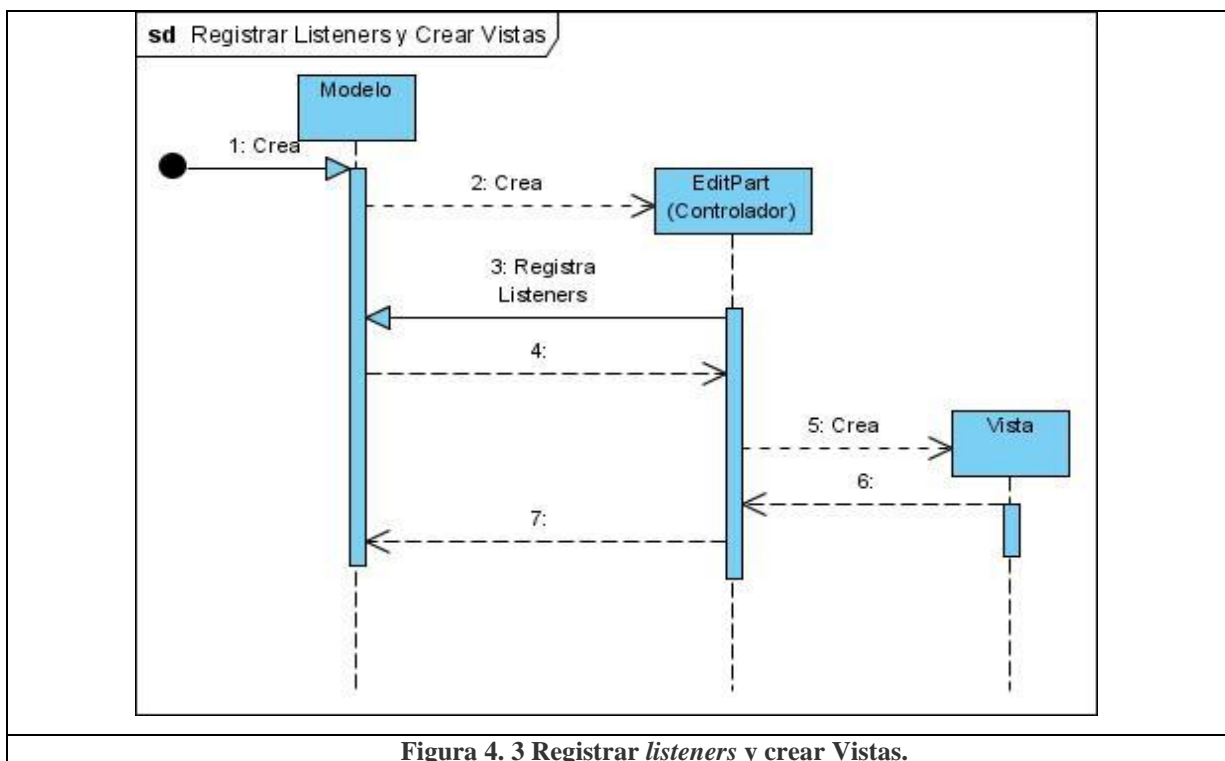
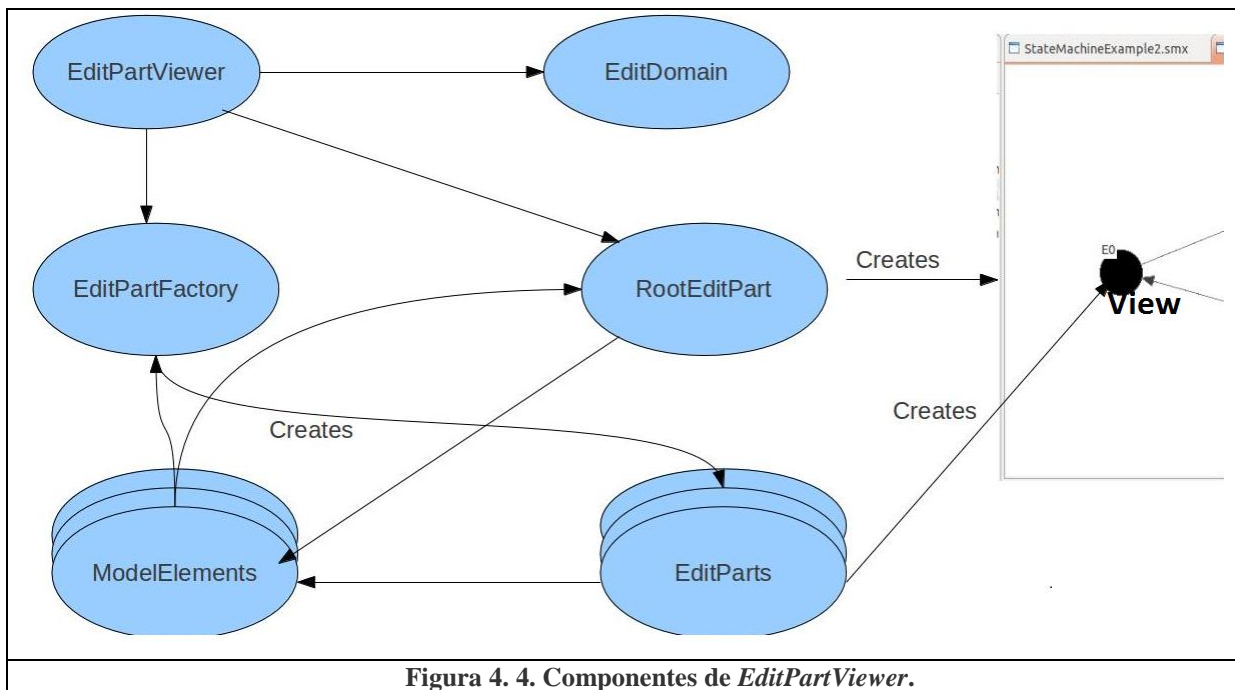


Figura 4. 3 Registrar *listeners* y crear Vistas.

En la figura 4.4, se puede observar como para cada modelo (*ModelElements*) existe un *EditPart* (controlador) asociado, y este a su vez tiene su vista (*View*) correspondiente. El *EditPartFactory* es una fábrica de *EditParts*, dependiendo del modelo que reciba el *EditPartFactory*, sabe que *EditPart* crear. La flecha que va desde *RootEditPart* hacia el *Canvas*, significa que el *RootEditPart*, es el *EditPart* correspondiente al *Canvas*. La responsabilidad de *EditPartViewer* es manejar las entradas del usuario en el editor, y enviarle estas entradas del usuario a la clase *EditDomain*, para que luego esta última construya los *Request*, en la sección 4.2.3 se explicara a detalle la responsabilidad de estas clases. En las secciones subsiguientes se explicará a detalle cada uno de los componentes de GEF, modelo, vista y controlador.



4.2.1 Modelo

Las características de un modelo de GEF son las siguientes:

- Toda la información editable por el usuario debe estar en el modelo. No debería existir información editable por el usuario ni en la vista ni en el controlador. (Rubel, Wren, & Clayberg, 2011).
- El modelo debe ser persistente, de modo que los cambios hechos por el usuario sean conservados a través de sesiones de la aplicación. (Rubel, Wren, & Clayberg, 2011).
- Toda la lógica de negocios debe estar contenida en el modelo. El modelo no debe tener referencias ni al controlador ni a la vista. (Rubel, Wren, & Clayberg, 2011).
- El modelo debe transmitir todos los cambios vía *listeners*, usando un *broadcast* para poder actualizar la vista. (Rubel, Wren, & Clayberg, 2011)
- En el modelo básicamente, se concentran todas las clases que manejan los conceptos del sistema a desarrollar. No tiene dependencia ni hereda de alguna clase del *framework*.

4.2.2 Vista

La vista en GEF, generalmente es una colección de figuras de *Draw2d*, dibujadas en un *Canvas* de SWT. *Draw2d* es un *framework* ligero construido en el tope de SWT, que provee un conjunto de figuras estándar y algoritmos gráficos de *layout*. Cuando el sistema necesita dibujar una nueva figura, el controlador obtiene la información del modelo, y este a su vez, se encarga de pasarle a la vista dicha información, y por último esta se dibuja en el *Canvas*. En la Vista las clases que escribe el desarrollador utilizan clases del *framework* en este caso *Draw2d*.

4.2.3 Controlador

Los *EditPolicy* definen todas las acciones de edición que se le puedan realizar a algún elemento, estas clases se instalan en cada *EditPart*, dependiendo de las necesidades de edición que tenga ese elemento (cambio de ubicación, tamaño, creación o eliminación) se instalarán los *EditPolicy* necesarios. El *framework* dispone de varios *EditPolicy*, si el desarrollador necesita un *EditPolicy* que no está en el *framework*, debe escribir su propio *EditPolicy* e instalarlo en el *EditPart* del objeto que necesite la acción de edición adicional.

En la figura 4.5, se puede ver un diagrama de secuencia de cómo reacciona el *framework* ante una entrada del usuario. Primero el usuario interactúa con el editor, selecciona algún elemento del *Toolbar* y luego éste se convierte en un *Tool* activo, en el paso 6 el editor le envía las entradas del usuario a *EditDomain*, para que esta construya un *Request*, dependiendo de la entrada del usuario el *Request* construido será de creación, eliminación, o edición, este *Request* se dispara en un *EditPolicy*, el *EditPolicy* que reciba ese *Request*, va a construir el comando correspondiente a la acción de edición que haya solicitado el usuario, y luego lo ejecutará para que al final este comando modifique el modelo. La modificación hecha por el comando es la entrada de la figura 4.2. Como se explicó al inicio de esta sección, ésta modificación llegará al modelo, y desde allí a través de un *listener*, se disparará un evento que será manejado en el *EditPart* (controlador).

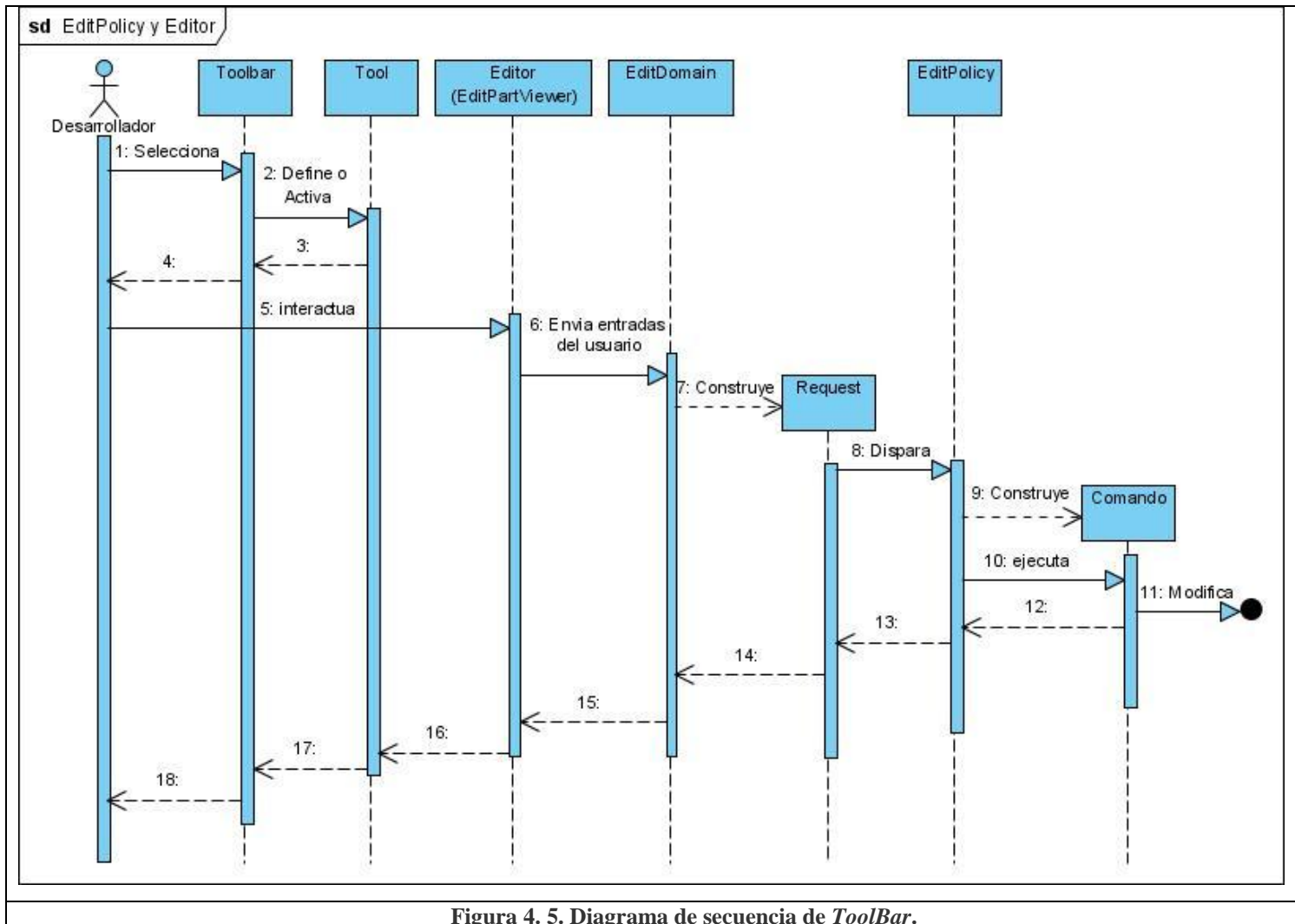


Figura 4. 5. Diagrama de secuencia de *ToolBar*.

En resumen, los *EditParts* (controladores) tienen las siguientes responsabilidades:

- Escuchar los cambios del modelo y actualizar la vista.
- Construir una figura que represente un objeto del modelo.
- Trabajar en conjunto con los *EditPolicy* para realizar la tarea de modificación del modelo, utilizando comandos.

4.3 Arquitectura del *Plugin* de Eclipse desarrollado

La figura 4.6, muestra las tres extensiones de Eclipse que se están utilizando en este trabajo. La primera es el editor de la máquina de estados; la segunda es un *Wizard* que brinda soporte para la creación de un archivo de la máquina de estados y la tercera es un submenú que brinda soporte para la generación de código. Cuando este submenú recibe un evento, ejecuta cierto código que se encuentra en la clase *GenetorCodeActionDelegate* con el fin de generar el código del archivo de la máquina de estados seleccionado por el usuario. Estas tres extensiones son completamente independientes una de la otra, pero trabajan en conjunto utilizando el API de Eclipse para dar toda la funcionalidad a la herramienta presentada en este trabajo. El *Plugin* de Eclipse se define en un archivo XML.

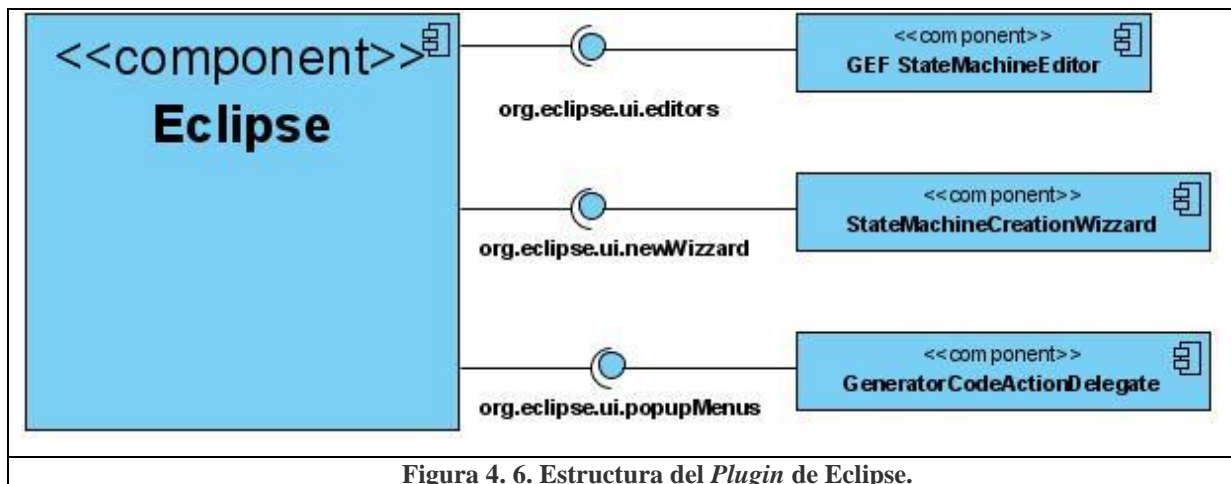


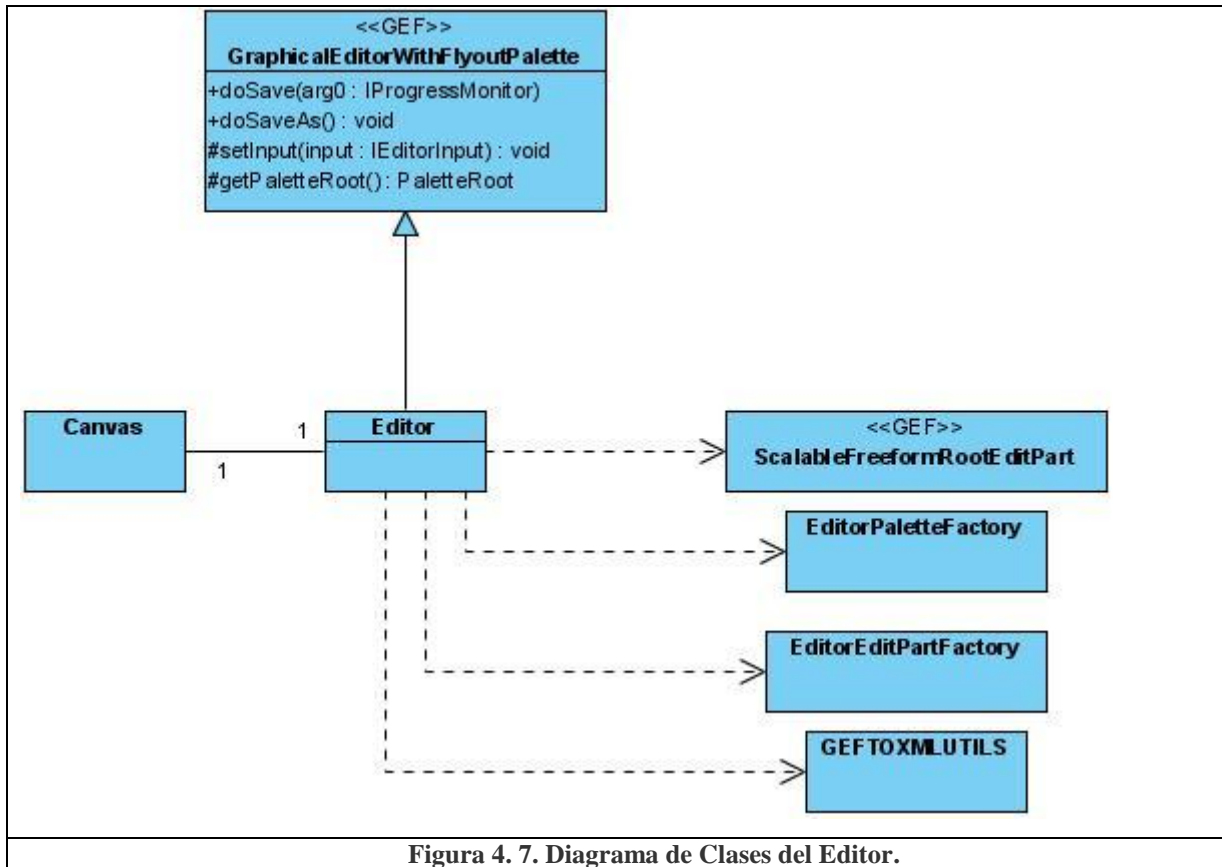
Figura 4. 6. Estructura del *Plugin* de Eclipse.

A continuación, se describirá a detalle la arquitectura de cada uno de los *Plugins* que conforman el editor de la máquina de estados.

4.4 GEF State Machine Editor

4.4.1 Editor

En la figura 4.7, se puede observar el diagrama de clases del objeto encargado de manejar el editor gráfico de GEF. La clase *Editor* hereda de la clase *GraphicalEditorWithFlyoutPalette*, que es una clase base de GEF que brinda la funcionalidad básica de un editor gráfico, que posee características tales como una barra de herramientas, que se puede ocultar para maximizar el tamaño del editor, y soporte para cargar y salvar archivos. La clase *editor* tiene una referencia a la clase *Canvas* ya que los métodos *doSave*, *doSaveAs* y *setInput* utilizan esta clase, los dos primeros para tomar el modelo de la clase *Canvas* y rellenarlo en un archivo XML y el método *setInput* lee el archivo XML y rellena el modelo de la clase *Canvas*. La dependencia con la clase *GEF2XML* es debido a que es una clase utilitaria usada para cargar y salvar el archivo XML. *EditorEditPartFactory* es una fábrica de objetos del tipo *EditPart* esta clase se explica a detalle en la sección 4.4.3 Por ultimo *EditorPaletteFactory* configura la barra de herramientas lateral. En la tabla 4.1, se muestra una lista de los métodos más importantes del diagrama de la figura 4.7.



Método	Descripción
<i>doSave(progressMonitor: IProgressMontior)</i>	Se encarga de guarda el archivo.
<i>doSaveAs()</i>	Se encarga de guardar el archivo y brinda al usuario la posibilidad de cambiar el nombre.
<i>setInput()</i>	Se encarga de cargar el archivo XML y rellenar todos los datos en la clase <i>Canvas</i> .
<i>getPaletteRoot</i>	Llama al método créate de la clase <i>createPaletteFactory</i> para cargar la barra de herramientas.

Tabla 4. 1. Métodos de la Clase Editor.

En la figura número 4.8, se encuentra el diagrama de clases de *EditActionBarContributor*, esta clase es la encarga de generar los *Actions* para deshacer y

rehacer. Como se ve en la figura 4.3, estos *Actions* se transforman en *Request* para llamar a los comandos. En la sección 4.4.4 se muestra como todos los comandos sobrescriben los métodos *Undo* y *Redo* de la clase base de GEF *Command*. En la tabla 4.2, se muestra una lista de los métodos más importantes de la clase *EditorActionBarContributor*.

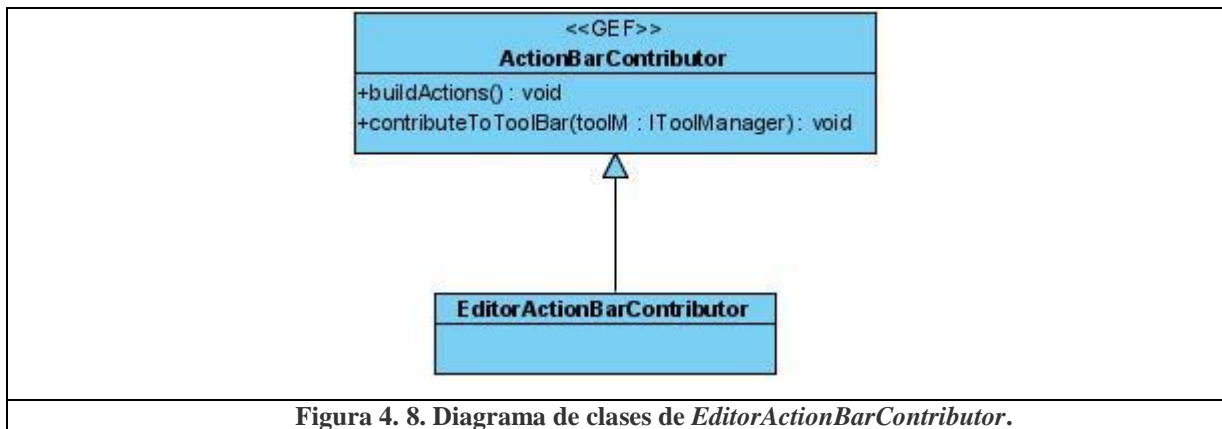


Figura 4. 8. Diagrama de clases de *EditorActionBarContributor*.

Método	Descripción
<i>buildActions()</i>	Se encarga de manejar las acciones de rehacer y deshacer.
<i>contributeToToolBar(toolM: IToolManager)</i>	Se encarga de manejar las acciones de rehacer y deshacer.

Tabla 4. 2. Métodos de la clase *EditorActionBarContributor*.

4.4.2 Modelo

El diagrama de clases de la figura 4.9, muestra la estructura general del modelo usado en el editor de máquina de estados. En estas clases es donde está toda la lógica de negocios del sistema, y toda la información editable por el usuario, en algunos casos estas clases son simplemente JavaBeans, es decir contienen métodos *getters* y *setters*. Por otra parte estas clases generalmente son las que contienen la información a guardar, en el caso de GEF en un archivo XML. Como se puede observar la clase *Canvas* está compuesta por las clases *Element* y *ArcShape*, este objeto es el encargado de llevar una lista de *Elements* de *ArcShape* y de *Events*, es decir, desde este objeto se puede navegar por todos los objetos del modelo, ya

que desde la clase *Canvas* se pueden obtener también los objetos que no son una composición directa como *LabelEvent* y *LabelState*, debido a que la clase *ArcShape* y *State* están compuestas por los tres objetos mencionados anteriormente.

La clase *ArcShape* es la encargada de manejar el concepto de Arco o conexión. Como se puede observar en la figura 4.9, este objeto tiene dos referencias a la clase *State*, debido a que para que exista un arco debe existir un estado origen y uno destino. Por el concepto de máquina de estados, un arco debe tener asociado un evento, por este motivo el objeto *ArcShape* debe tener una referencia a las clases *Event* y *LabelEvent*.

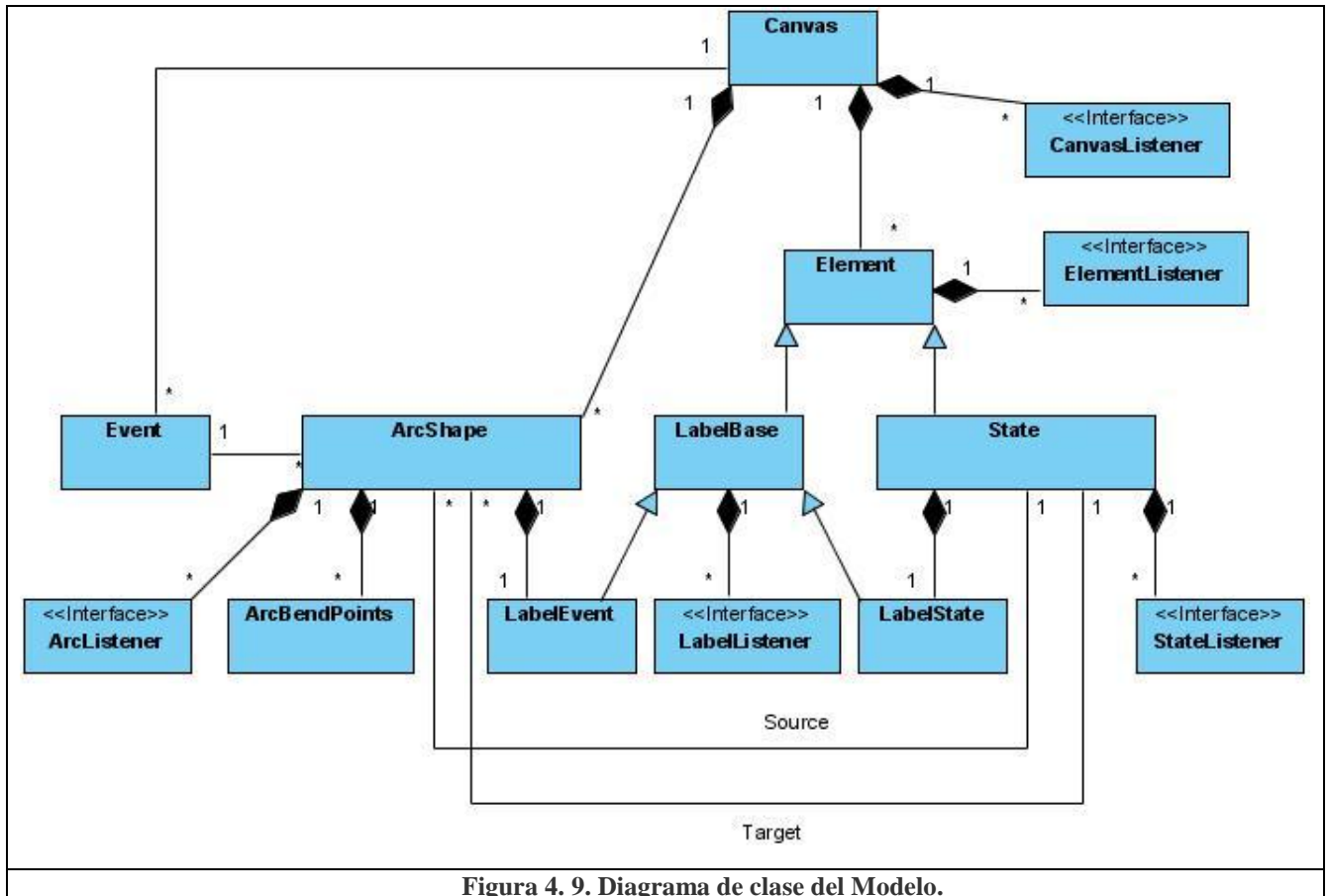
La clase *Element* es la encargada de manejar la generalización de un elemento cualquiera pintado en el área de dibujo, es decir cualquier elemento que posea coordenadas y un tamaño. Los dos objetos que heredan de esta clase son *State* y *LabelBase*. *State* está encargado de manejar el concepto de estado por lo tanto tiene dos listas de *ArcShape*, una representa los arcos salientes de un estado y la otra los entrantes, como todo estado de una máquina de estados debe tener un nombre y por este motivo un *State* está compuesto por un *LabelState*.

Event está encargado de representar el concepto de evento de una máquina de estados, como se puede observar en la figura 4.9, esta clase tiene una lista de *ArcShape* dicha lista representa los arcos donde existe un evento asociado.

El objeto *BaseLabel* está encargado de manejar toda la data de una etiqueta pintada en el área de dibujo, y es la clase padre para *LabelEvent* y *LabelState*.

Por otra parte todas estas clases, en algunos métodos particulares, disparan eventos, (*listeners*) para que el controlador refleje en la vista los cambios hechos en el modelo. Todas las clases que manejan los eventos son interfaces de Java. En las tablas 4.3, y 4.4, se

encuentran una lista de los métodos y atributos más importantes de todas las clases de la figura 4.9.



Método	Descripción
Canvas	
<i>addElement(element: Element)</i>	Se encarga de agregar un elemento a la lista de elementos de la clase <i>Canvas</i> y disparar el evento para que se pinte el nuevo elemento en el área de dibujo.
<i>removeElement(element: Element)</i>	Se encarga de eliminar un elemento de la lista de elementos de la clase <i>Canvas</i> y disparar el evento para que se refleje dicho cambio en la vista.
CanvasListener	
<i>elementAddedListener(element: Element)</i>	Es implementado en la clase <i>CanvasEditPart</i> explicada más adelante. Se encarga de manejar el evento que se dispara cuando se agregar un elemento al <i>Canvas</i> , generalmente en el método <i>addElement</i> .
<i>elementRemovedListener(element: Element)</i>	Es implementado en la clase <i>CanvasEditPart</i> explicada más adelante. Se encarga de manejar el evento que se dispara cuando se elimina un elemento del <i>Canvas</i> , generalmente desde el método <i>removeElement</i> .
Element	
<i>setLocation(newX:int, newY:int)</i>	Se encarga de asignar una nueva posición a un elemento y dispara el evento para que se pinte el cambio en la vista.
<i>setSize(newWidth:int, newHeight:int)</i>	Se encarga de asignar un nuevo tamaño a un elemento y dispara el evento para que se pinte el cambio en la vista.
ElementListener	
<i>changeLocationListener(x:int, y:int)</i>	Se encarga de manejar el evento disparado en el método <i>setLocation</i> y es implementado en la clase <i>ElementEditPart</i> explicado más adelante.
<i>changeSizeListener(w:int, h:int)</i>	Se encarga de manejar el evento disparado en el método <i>setSize</i> e implementado en <i>ElementEditPart</i> .
State	
<i>addConnection(conn: ArcShape)</i>	Se encarga de agregar un nuevo arco a un estado y dispara el evento para que se agregue una nueva conexión en la vista.
<i>removeConnection(conn: ArcShape)</i>	Se encarga de eliminar un arco de un estado y dispara el evento para que se elimine la conexión de la vista.
StateListener	
<i>addSourceListener(source: State)</i>	Se encarga de manejar el evento disparado en el método <i>addConnection</i> y es implementado en la clase <i>StateEditPart</i> explicada más adelante.
<i>addTargetListener(target: State)</i>	Se encarga de manejar el evento disparado en el método <i>addConnection</i> y es implementado en la clase <i>StateEditPart</i> explicada más adelante.
<i>addLoopListener(state: State)</i>	Se encarga de manejar el evento disparado en el método <i>addConnection</i> y es implementado en la clase <i>StateEditPart</i> explicada más adelante.

<i>removeSourceArcListener(arc: ArcShape)</i>	Se encarga de manejar el evento disparado en el método <i>removeConnection</i> y es implementado en la clase <i>StateEditPart</i> explicada más adelante.
<i>removeSourceArcListener(arc: ArcShape)</i>	Se encarga de manejar el evento disparado en el método <i>removeConnection</i> y es implementado en la clase <i>StateEditPart</i> explicada más adelante.
<i>ArcShape</i>	
<i>insertBendPoint(index:int,point:BendPoint)</i>	Se encarga de agregar un nuevo <i>BendPoint</i> a la lista de <i>BendPoints</i> de <i>ArcShape</i> .
<i>removeBendPoint(index: int)</i>	Se encarga de eliminar un <i>BendPoint</i> de la lista de <i>BendPoints</i> de <i>ArcShape</i> .
<i>reconnect()</i>	Se encarga de agregar un nuevo arco entre un estado origen y uno destino.
<i>disconnect()</i>	Se encarga de eliminar un arco entre un estado origen y uno destino.
<i>ArcListener</i>	
<i>bendPointChangeListener()</i>	Se encarga de manejar los eventos disparados en los métodos <i>insertBendPoint</i> y <i>removeBendPoint</i> .

Tabla 4. 3. Tabla de Métodos de los modelos.

Atributo	Descripción
<i>Canvas</i>	
<i>targetPath: string</i>	Representa la ruta del archivo a generar.
<i>pathPackage: string</i>	Representa el paquete de java donde se va a generar el archivo .java.
<i>Element</i>	
<i>x: int</i>	Representa la coordenada “x” dentro del área de dibujo.
<i>y: int</i>	Representa la coordenada “y” dentro del área de dibujo.
<i>w: int</i>	Representa el ancho de una figura.
<i>h: int</i>	Representa el alto de una figura.
<i>State</i>	
<i>type: int</i>	Representa el tipo de estado (inicial, normal, final).
<i>LabelBase</i>	
<i>text: string</i>	Representa el texto de una etiqueta.

Tabla 4. 4. Atributos de los modelos.

4.4.3 Controlador (*EditParts*)

En el diagrama de la figura 4.10, se muestran los *EditParts* que heredan de la clase *ElementEditPart*, este objeto hereda de la clase base de GEF *AbstractGraphicalEditPart* que provee toda la funcionalidad para poder comunicarse con las clases que manejan las figuras de Draw2d, manejar conexiones y comunicarse con los modelos. Se debe recordar que los *EditParts* en GEF son el controlador, por lo tanto estas clases deben implementar todos los *listeners* para reflejar en la vista los cambios en el modelo. En GEF los *EditParts* trabajan en conjuntos con los *EditPolicy* y estos a su vez deben generar nuevas instancias de comandos, por lo tanto es desde los *EditParts* que se generan los nuevos comandos para generar cambios en los modelos.

La herencia que se maneja en la figura 4.10, es para poder manejar las etiquetas igual que los estados. Por herencia un *StateEditPart* y un *BaseLabelEditPart* también son un *AbstractGraphicalEditPart*. *StateEditPart* es el controlador para la clase *State* del modelo. La clase *State* implementa las interfaces *StateListener* y *NodeEditPart*, *StateListener* tiene todos los métodos para agregar un arco o eliminarlo, *NodeEditPart* sirve para instanciar a los métodos de GEF que se encargan de anclar un arco a una figura, en el caso de este trabajo a un estado. La dependencia con la clase *ComponentEditPolicy* sirve para generar nuevas instancias del comando *DeleteElementCommand* (Ver sección 4.4.4). La otra dependencia con *GraphicalNodeEditPolicy* es para manejar los arcos y esta se encarga de generar una nueva instancia para el comando *CreateConnectionCommand* (Ver sección 4.4.4).

LabelBaseEditPart es el controlador para la clase *LabelBase* de los modelos. Esta clase implementa la interfaz *LabelListener* que tiene un método que se encarga de reflejar en la vista un cambio de texto de la etiqueta.

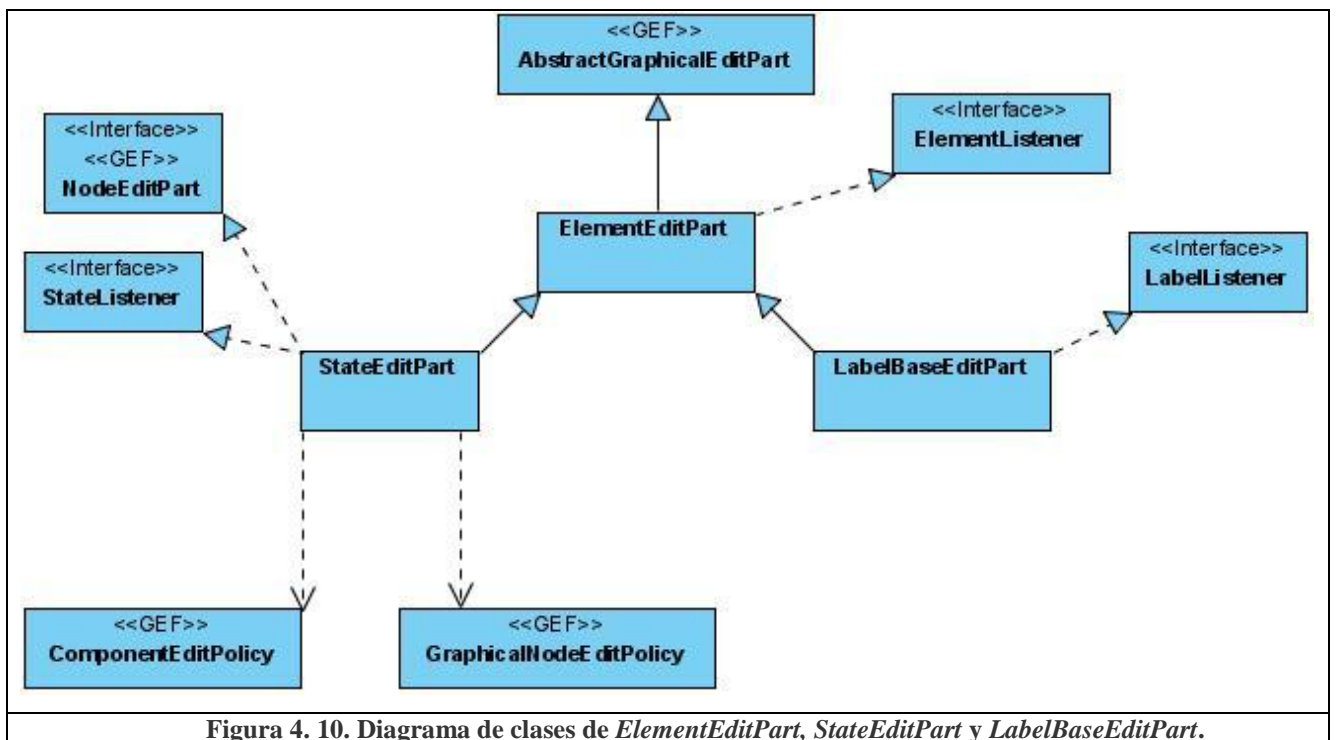
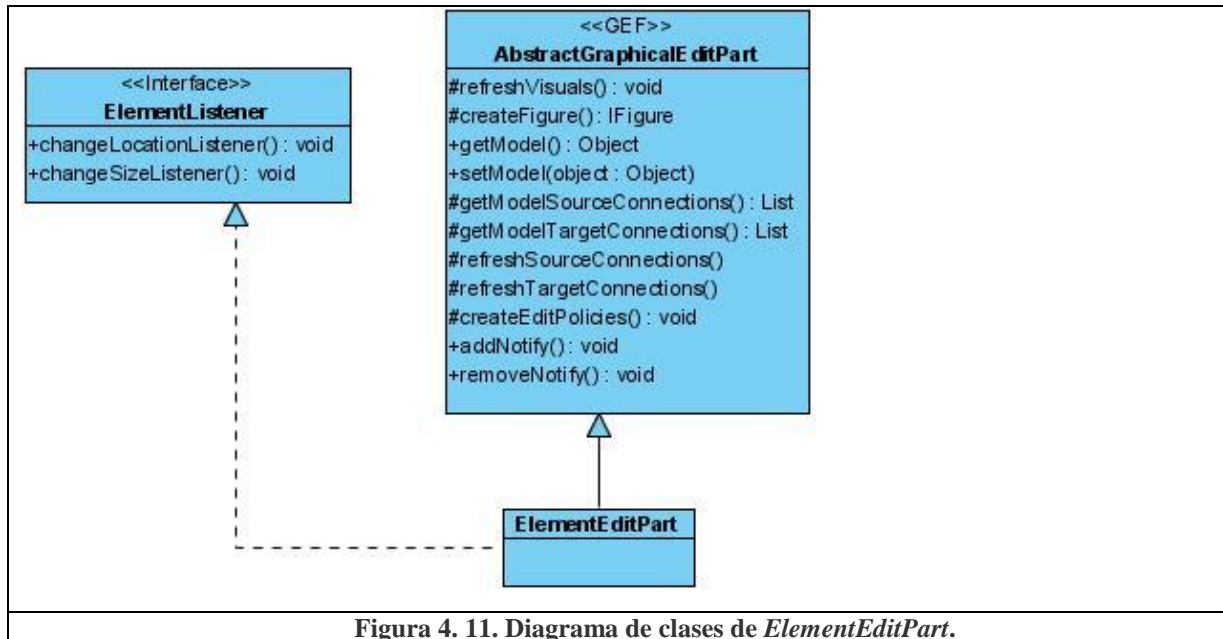


Figura 4. 10. Diagrama de clases de *ElementEditPart*, *StateEditPart* y *LabelBaseEditPart*.

A continuación se explicaran a más detalle cada uno de los *EditParts* mostrados en la figura 4.10.

En la figura 4.11, se muestra el diagrama de clases de *ElementEditPart*, este objeto implementa la interfaz *ElementListener* que tiene dos métodos que se encargan de reflejar un cambio de ubicación o tamaño en la vista. En la tabla 4.5, se muestran métodos más importantes de la clase *ElementEditPart*.

Figura 4. 11. Diagrama de clases de *ElementEditPart*.

Método	Descripción
<i>ElementEditPart</i>	
<i>changeLocationListener(x: int, y: int)</i>	Maneja un evento disparado desde el modelo y refleja el cambio en la vista. En su implementación llama a <i>refreshVisual</i> .
<i>changeSizeListener(w: int, h: int)</i>	Maneja un evento disparado desde el modelo y refleja el cambio en la vista. En su implementación llama a <i>refreshVisual</i> .
<i>AbstractGraphicalEditPart</i>	
<i>refreshVisual()</i>	Este método se encarga de reflejar en la vista un cambio de tamaño o de localización.
<i>createFigure()</i>	Se encarga de crear una figura para una instancia de un <i>EditPart</i> .
<i>getModel()</i>	Se encarga de retornar el modelo correspondiente a un <i>EditPart</i> .
<i>setModel(model: Object)</i>	Es llamado en el constructor de cada <i>EditPart</i> y asigna el modelo correspondiente a dicha clase.
<i>getModelSourceConnections()</i>	Es llamado cuando se instancia un editor para poder dibujar los arcos existentes en el modelo.
<i>getModelTargetConnections()</i>	Es llamado cuando se instancia un editor para poder dibujar los arcos existentes en el modelo.
<i>createEditPolicy()</i>	Este método se encarga de crear todos los <i>EditPolicy</i> asociados a un <i>EditPart</i> .
<i>addNotify()</i>	Este método se encarga de agregar los <i>listeners</i> correspondientes a un <i>EditPart</i> .

<code>removeNotify()</code>	Este método se encarga de eliminar los <i>listeners</i> correspondientes a un <i>EditPart</i> .
-----------------------------	---

Tabla 4. 5. Métodos de la clase *ElementEditPart*.

La figura 4.12, muestra el diagrama de clases de *StateEditPart* debemos recordar que esta clase hereda de *ElementEditPart* y por lo tanto como se ve la figura 4.10, *StateEditPart* también es un *AbstractGraphicalEditPart* y sobrescribe varios métodos de esa clase. En la tabla 4.6, se encuentra una lista de los métodos más importantes de la clase *StateEditPart*.

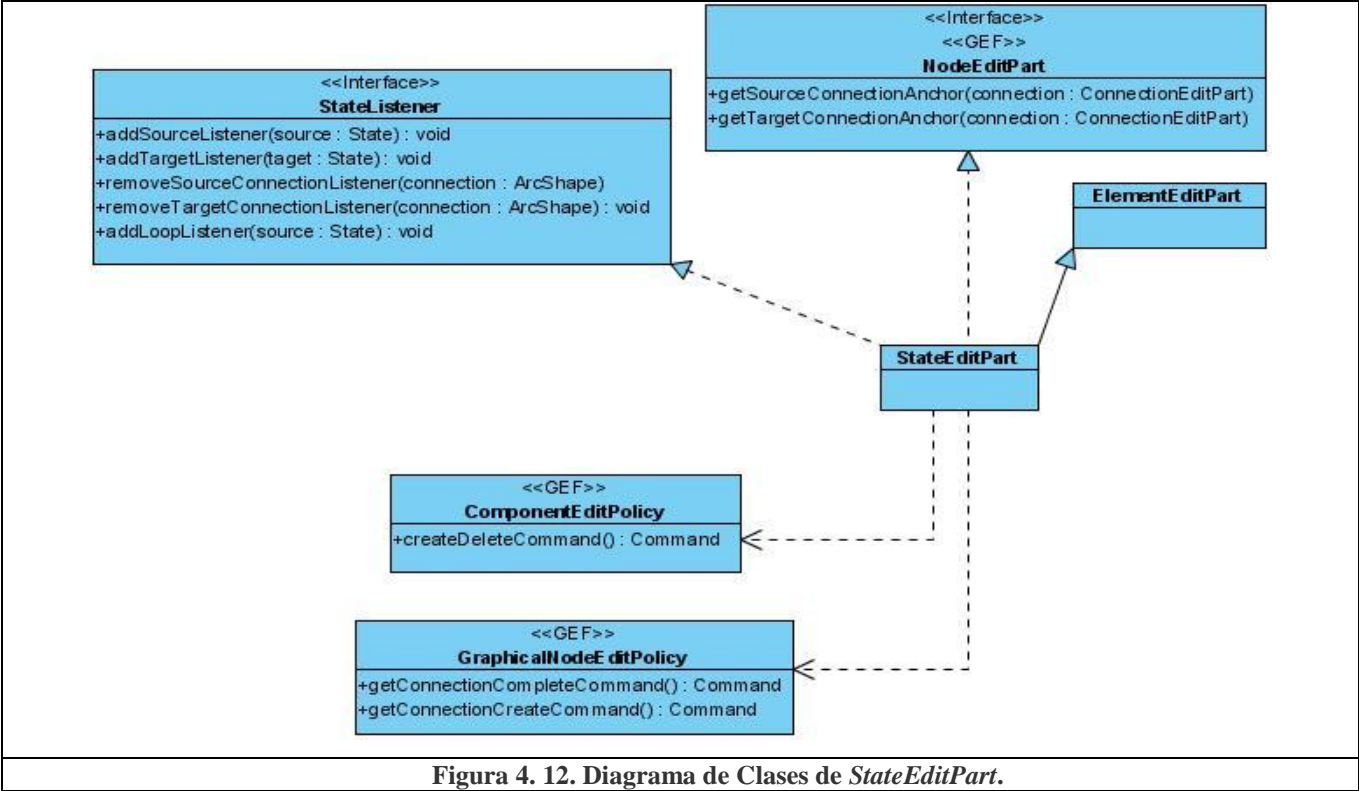


Figura 4. 12. Diagrama de Clases de *StateEditPart*.

Método	Descripción
<code>addSourceListener(state: State)</code>	Maneja un evento disparado desde el modelo, es implementado en la clase <i>StateEditPart</i> y se encarga de llamar al método de GEF <i>refreshSourceConnections</i> .
<code>addTargetListener(state: State)</code>	Maneja un evento disparado desde el modelo, es implementado en la clase <i>StateEditPart</i> y se encarga de llamar al método de GEF <i>refreshTargetConnections</i> .
<code>removeSourceConnectionListener()</code>	Maneja un evento disparado desde el modelo, es implementado en <i>StateEditPart</i> y se encarga de llamar al método de GEF <i>refreshSourceConnections</i> .

<i>removeTargetConnectionListener()</i>	Maneja un evento disparado desde el modelo, es implementado en <i>StateEditPart</i> y se encarga de llamar al método de GEF <i>refreshTargetConnections</i> .
<i>addLoopListener</i>	Maneja un evento disparado desde el modelo, es implementando en <i>StateEditPart</i> y se encarga de llamar los métodos de GEF <i>refreshSourceConnections</i> y <i>refreshTargetConnections</i> .
<i>refreshSourceConnections()</i>	Método de GEF que sirve para actualizar cualquier modificación ya se de inserción o eliminación de un arco saliente a una figura específica.
<i>refreshTargetConnections()</i>	Método de GEF que sirve para actualizar cualquier modificación ya se de inserción o eliminación de un arco entrante a una figura específica.

Tabla 4. 6. Métodos de la clase *StateEditPart*.

En la figura 4.13, se encuentra el diagrama del controlador de la clase *LabelBase* del modelo, *LabelBaseEditPart*. Esta clase implementa la interfaz *LabelListener* que tiene un método que se encarga de reflejar en la vista un cambio del texto de la etiqueta. Esta clase según la figura 4.10, también hereda de *ElementEditPart* y por lo tanto también es un *AbstractGraphicalEditPart*.

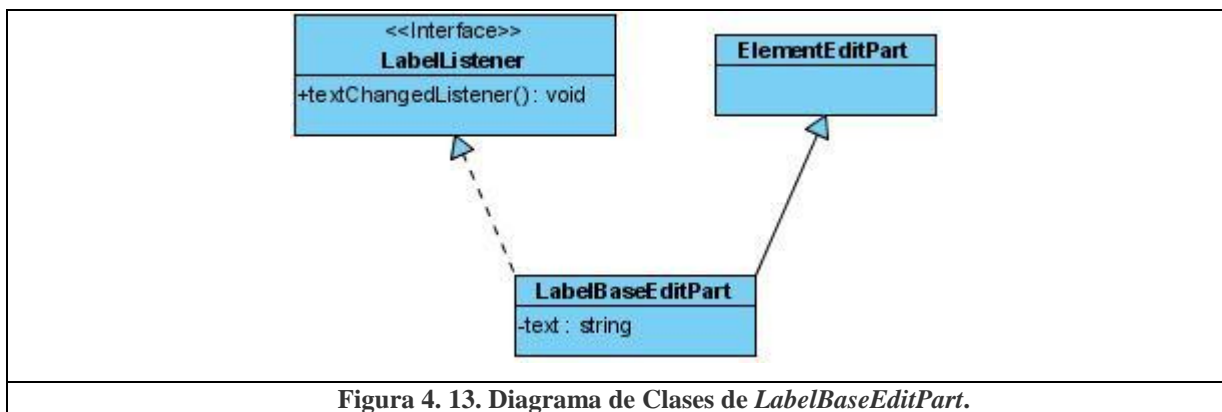


Figura 4. 13. Diagrama de Clases de *LabelBaseEditPart*.

En la figura 4.14, se muestra el diagrama de clase de *ArcShapeEditPart* esta clase hereda de *AbstracConnectionEditPart* que es un clase de GEF que provee toda la funcionalidad para manejar conexiones entre elementos. *ArcShapeEditPart* implementa *ArcListener*, esta interfaz es utilizada para actualizar la lista de *BendPoints*, para manejar las validaciones de si un arco no tiene evento, y para validar si la máquina de estados es determinista. La dependencia con la clase *ConnectionEditPolicy* se debe a que esta clase tiene un método *getDeleteCommand* que retorna una nueva instancia del comando

DeleteConnectionCommand (Ver sección 4.4.4). La clase *ArcShapeEditPolicy* hereda de *BendPointEditPolicy* que es una clase base de GEF que brinda tres métodos para el manejo del algoritmo de ruteo, cada uno de estos métodos debe retornar la instancia del comando correspondiente. En la tabla 4.7, se encuentra una lista de los métodos más importantes.

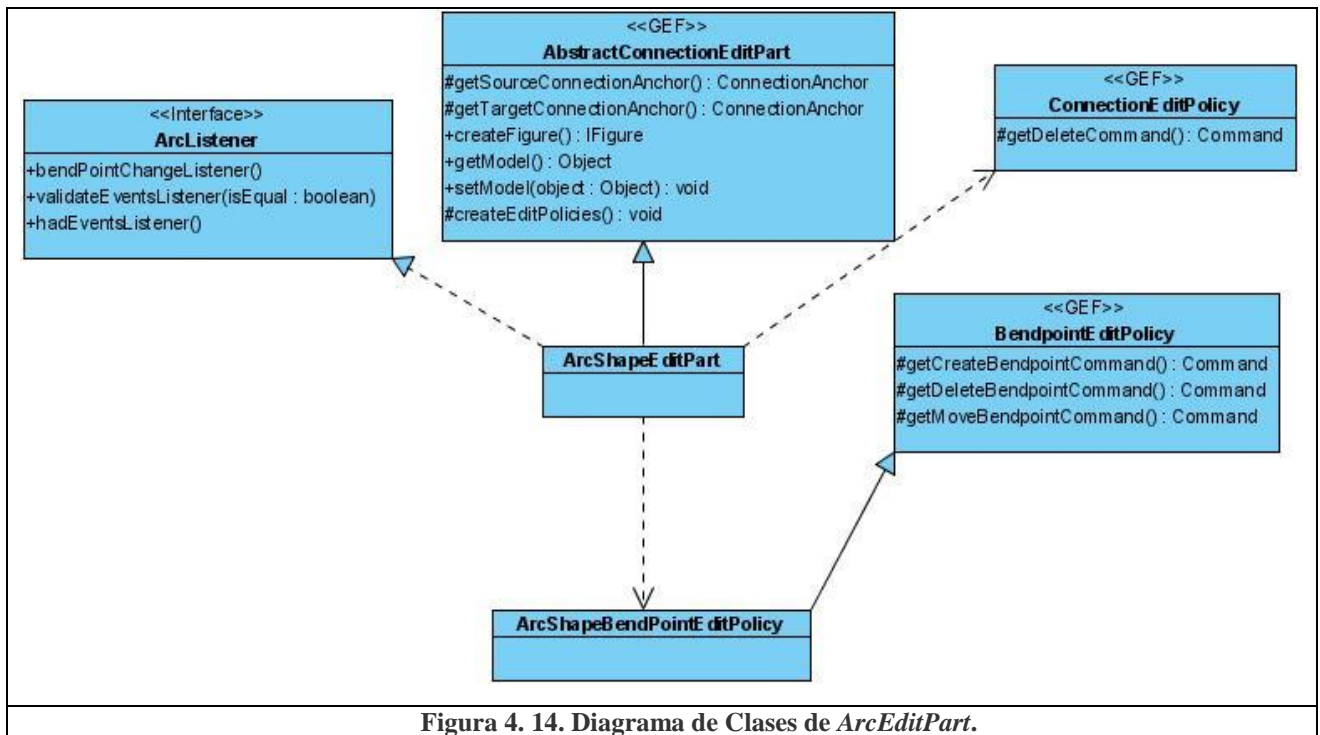


Figura 4. 14. Diagrama de Clases de *ArcEditPart*.

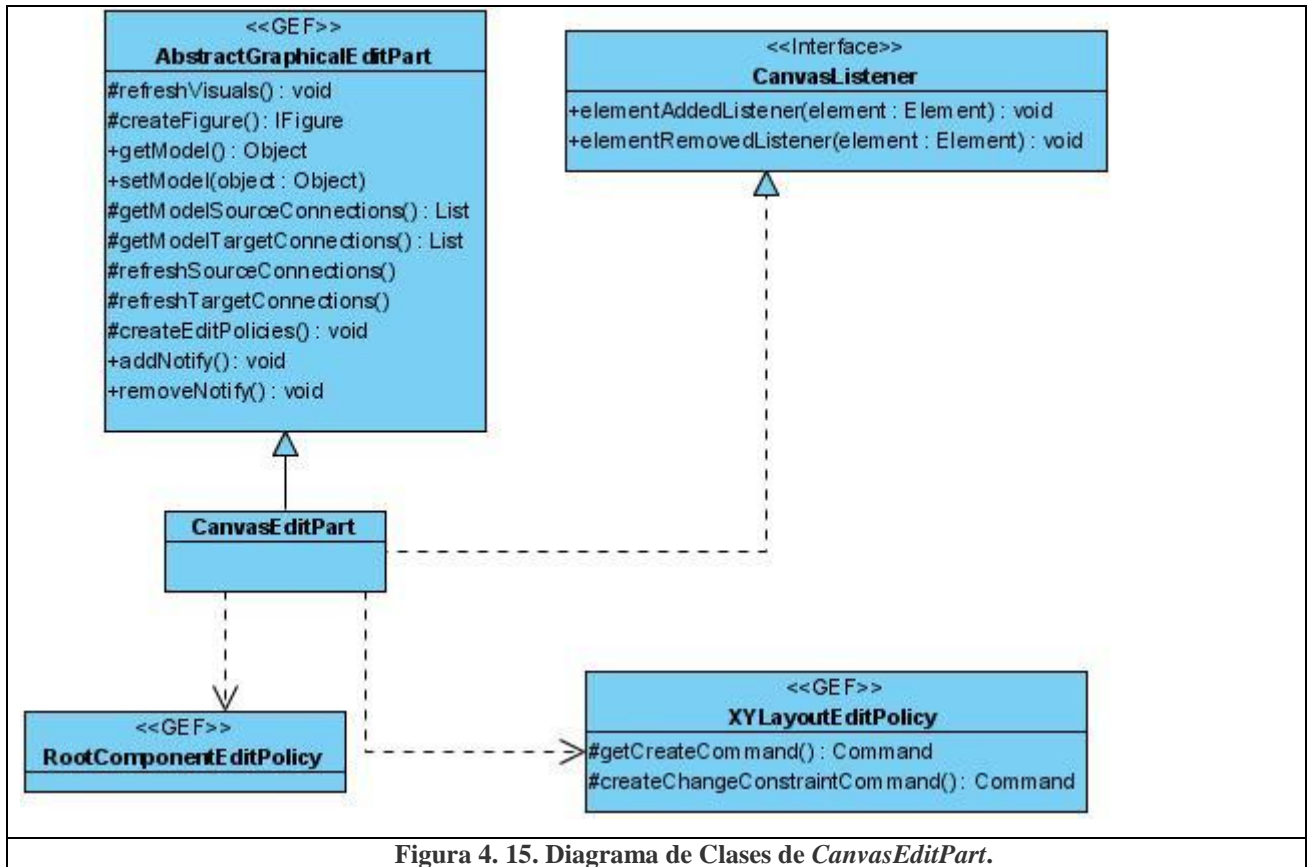
Método	Descripción
<i>ArcShapeEditPart</i>	
<i>getSourceConnectioAnchor()</i>	Se encarga de llamar al método de GEF para anclar la flecha del arco a un círculo(estado)
<i>getTargetConnectionAnchor()</i>	Se encarga de llamar al método de GEF para anclar la flecha del arco a un círculo(estado)
<i>createFigure()</i>	Crea un nuevo arco en el área de dibujo
<i>getModel()</i>	Retorna el modelo correspondiente a <i>ArcShapeEditPart</i> (<i>ArcShape</i>)
<i>setModel(object: Object)</i>	Es llamado en el constructor de <i>ArcShapeEditPart</i> y asigna el modelo correspondiente a dicha clase
<i>createEditPolicy()</i>	Se encarga de crear las clases <i>ConnectionEditPolicy</i> y <i>ArcShapeBendPointEditPolicy</i>
<i>ArcListener</i>	
<i>bendPointChangeListener()</i>	Maneja un evento disparado desde el modelo y refleja el cambio en la vista. Es implementado en <i>ArcShapeEditPart</i> y se encarga de actualizar la lista de <i>BendPoints</i> .

<i>validateEvents(isEqual: boolean)</i>	Maneja un evento disparado desde el modelo y Se encarga de validar que la máquina de estados dibujada sea determinista pitando los arcos de color rojo.
<i>hadEventListener()</i>	Maneja un evento disparado desde el modelo Se encarga de validar que un arco tenga un evento asociado pitando el arco de color azul.
<i>ArcShapeBendPointEditPolicy</i>	
<i>getCreateBendPointCommand()</i>	Se encarga de retornar nuevas instancias del comando <i>createBendPointCommand</i> .
<i>getMoveBendPointCommand()</i>	Se encarga de retornar nuevas instancias del comando <i>MoveBendPointCommand</i> .
<i>getDeleteBendPointCommand()</i>	Se encarga de retornar nuevas instancias del comando <i>DeleteBendPointCommand</i> .

Tabla 4. 7. Métodos de la figura 4.14.

En la figura 4.15, se muestra el diagrama de *CanvasEditPart* esta clase hereda de *AbstractGraphicalEditPart* y como se explicó en la sección al principio de esta sección es una clase base de GEF que provee toda la funcionalidad de un *EditPart*. La clase *CanvasEditPart* es el controlador de la clase *Canvas* del modelo, esta clase implementa la interfaz *CanvasListener*. En la implementación de *CanvasListener* se agrega o se elimina un elemento del área de dibujo dependiendo del método que se haya invocado (*elementAddedListener* o *elementRemovedListener*).

Por otra parte la clase *RootComponentEditPolicy* indica a GEF que el objeto *CanvasEditPart* es el encargado de manejar todos los elementos que se van a pintar en el área de dibujo. La clase *XYLayoutEditPolicy* se encarga de recibir un *Request* y dependiendo del *Request* recibido va a retornar nueva una instancia de *CreateElementCommand* o *MoveAndResizeCommand*. En la tabla 4.8, se listan los métodos más importantes de este diagrama.

Figura 4. 15. Diagrama de Clases de *CanvasEditPart*.

Método	Descripción
<i>CanvasEditPart</i>	
<i>createFigure()</i>	Se encarga de configurar y darle un color de fondo a el área de dibujo.
<i>getModel()</i>	Se encarga de retornar el modelo correspondiente a <i>CanvasEditPart</i> (<i>Canvas</i>).
<i>setModel(object: Object)</i>	Se encarga de asignar en el constructor de <i>CanvasEditPart</i> el modelo correspondiente a dicha clase.
<i>createEditPolicy()</i>	Se encarga de crear las nuevas instancias de las clases <i>RootComponentEditPolicy</i> y <i>XYLayoutEditPolicy</i> .
<i>CanvasListener</i>	
<i>elementAddedListener()</i>	Maneja un evento disparado desde el modelo, es implementado en la clase <i>CanvasEditPart</i> y se encarga de agregar un nuevo elemento al área de dibujo.
<i>elementRemovedListener()</i>	Maneja un evento disparado desde el modelo, es implementado en la clase <i>CanvasEditPart</i> y se encarga de eliminar un elemento del área de dibujo.
<i>XYLayoutEditPolicy</i>	
<i>getCreateCommand()</i>	Se encarga de generar una nueva instancia del comando <i>CreateElementCommad</i> .

<code>createChangeConstraintComm and()</code>	Se encarga de generar una nueva instancia del comando <i>MoveAndResizeCommand</i> .
---	---

Tabla 4. 8. Métodos de la figura 4.15.

El diagrama de la figura 4.16, es el diagrama de clases para la clase *EditPartFactory*. Esta clase es la encargada de generar una nueva instancia de un *EditPart* para un modelo dado. *EditPartFactory* tiene un solo método *createEditPart* este método es sobrescrito, y básicamente su comportamiento es que recibe un modelo y dependiendo del modelo que reciba, el método retorna una nueva instancia del *EditPart* asociado a el modelo recibido. El fragmento de código de la 4.17 ilustra el funcionamiento del método *createEditPart*.

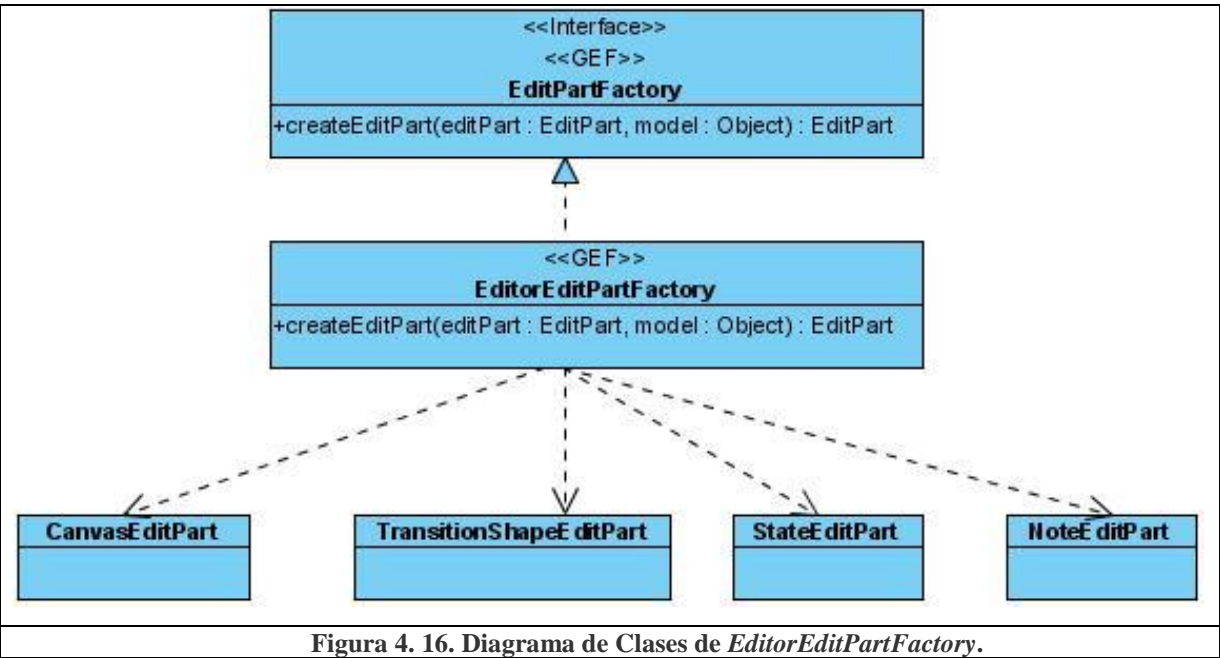


Figura 4. 16. Diagrama de Clases de *EditorEditPartFactory*.

```
public EditPart createEditPart(EditPart arg0, Object model) {  
    if(model instanceof State){  
        return new StateEditPart((State) model);  
    }  
  
    if(model instanceof Canvas){  
        return new CanvasEditPart((Canvas) model);  
    }  
}
```



```

    }
    // ... otros edit parts ...

```

Figura 4. 17. Código de método *createEditPart*.

4.4.4 Comandos

En la figura 4.18, se encuentra el diagrama de clases de todos los comandos del Editor. Cada comando realiza cambios a los modelos dependiendo de una entrada hecha por el usuario. En los comandos no deben existir referencias a los *EditPart* ni llamar a alguna función de estas clases, porque se rompe la arquitectura MVC del *framework*.

Algunos de estos comandos fueron mencionados en las secciones 4.4.1, 4.4.2 y 4.4.3. En esta sección se explicara a detalle cada uno de ellos.

Command (GEF) es la clase de la que heredan todos los comandos, brinda soporte para rehacer y deshacer. Cada comando que herede de esta clase debe sobrescribir los métodos *undo* y *execute*. Por defecto el *framework* tiene una implementación para el método *redo*, que es llamar al método *execute*, si el desarrollador desea cambiar este comportamiento debe sobrescribir el método *redo*. En la tabla 4.9, se listan los métodos de la clase *Command* y su descripción.

Método	Descripción
<i>execute()</i>	Método que se encarga de hacer los cambios en el modelo cuando el comando es llamado.
<i>undo()</i>	Método que se encarga de revertir los cambios hechos en el método <i>execute</i> .
<i>redo()</i>	Por defecto llama al método <i>execute</i> . Su tarea es volver al estado en que se encontraba el editor antes de llamar al método <i>undo</i> .

Tabla 4. 9. Métodos de la figura 4.18.

CreateElementCommand este es comando es el encargado de crear nuevos elementos y añadirlos a la clase *Canvas* (modelo). El constructor del comando recibe el elemento a construir, en una posición dentro del área de dibujo, y una referencia a la clase *Canvas* para poder agregar elementos a la lista de elementos de dicha clase.

DeleteElementCommand este comando se encarga de eliminar elementos de la lista de la clase *Canvas*. Esta clase recibe en su constructor el elemento a eliminar y una referencia a la clase *Canvas*. El comando tiene en sus atributos dos listas de *sourceConnections* y *targetConnections*, debido a que cuando se elimina un estado del área de dibujo se deben borrar también sus conexiones, y al tratar de deshacer la ejecución de este comando es necesario restablecer dichas conexiones. De esta forma, estas listas se están utilizando como una copia de respaldo para brindar soporte al método *undo*.

MoveAndResizeCommand es el comando encargado de manejar todo lo relacionado con los cambios de localización y tamaño. El constructor del comando recibe como parámetros el elemento a modificar, y sus nuevas coordenadas y tamaño.

CreateConnectionCommand y *DeleteConnectionCommand*, son los comandos encargados de crear una nueva conexión o eliminarla. En estos comandos es donde se hace uso del método *reconnect* que sirve para crear una nueva conexión o el método *disconnect* que sirve para eliminar una conexión.

CreateBendPointCommand, *MoveBendPointCommand* y *DeleteBendPointCommand* son los comandos encargados de crear mover o eliminar un *Bendpoint* respectivamente modificando la lista de *Bendpoints* de la clase *ArcShape*.

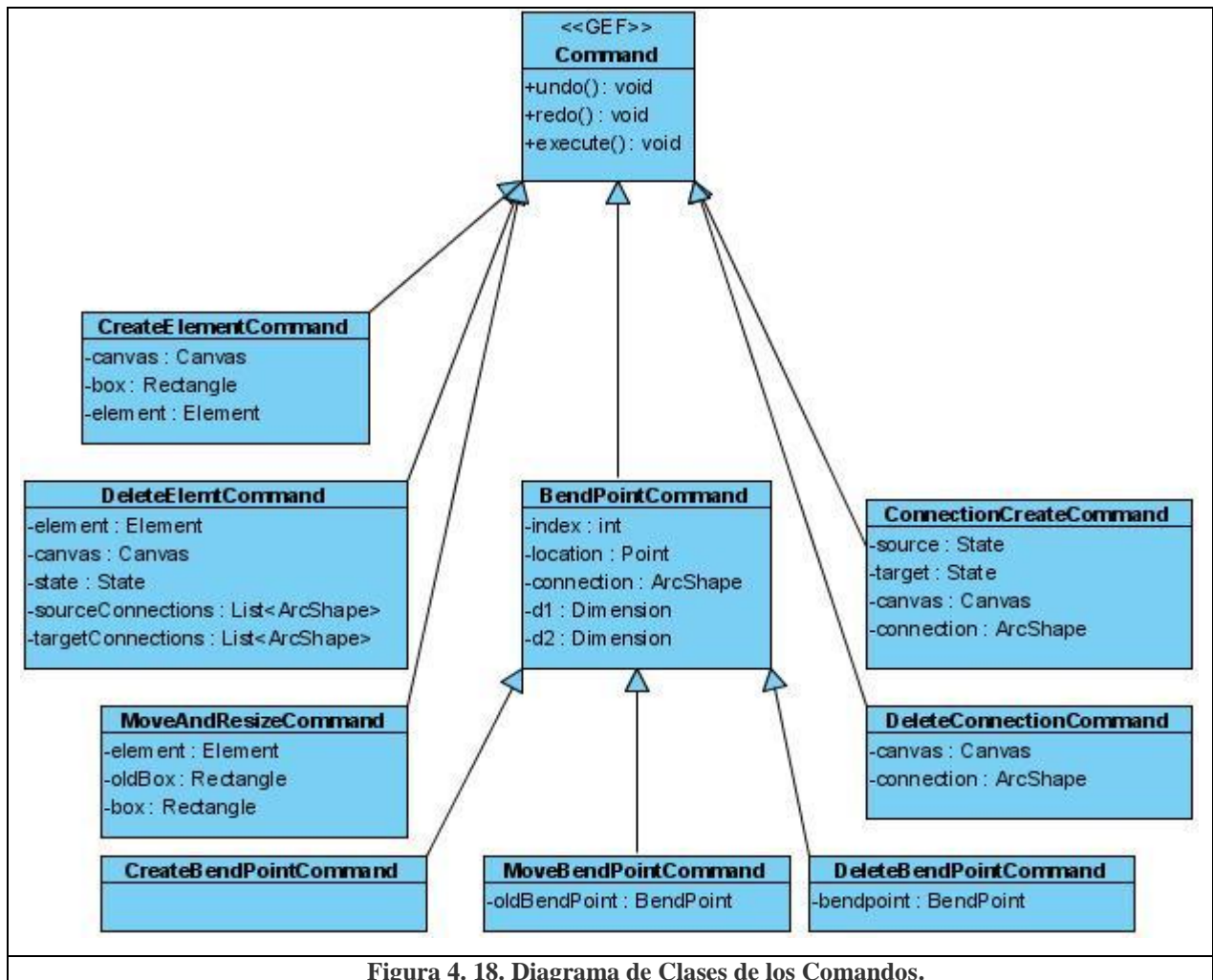


Figura 4. 18. Diagrama de Clases de los Comandos.

4.4.5 Vistas

La vista en GEF son bastante sencillas, como se explicó anteriormente utilizan Draw2d. La figura 4.19 ilustra cómo funciona la vista para el caso de un estado, la clase *StateFigure* hereda de una clase base de Draw2d llamada *Ellipse*, que brinda todo el soporte para dibujar elipses en un *Canvas* de SWT. El constructor de la clase recibe un modelo de *State*, y dependiendo del tipo de estado le da un color a la elipse.

```
public class StateFigure extends Ellipse {  
    public StateFigure(State state) {  
        setOpaque(true);  
  
        if (state.getType() == State.NORMAL) {  
            setBackgroundColor(ColorConstants.white);  
        }  
        if (state.getType() == State.FINAL){  
            setBackgroundColor(ColorConstants.red);  
        }  
        if (state.getType() == State.INITIAL){  
            setBackgroundColor(ColorConstants.black);  
        }  
    }  
}
```

Figura 4. 19. Código de la clase *StateFigure*.

4.5 Generator Code Action Delegate

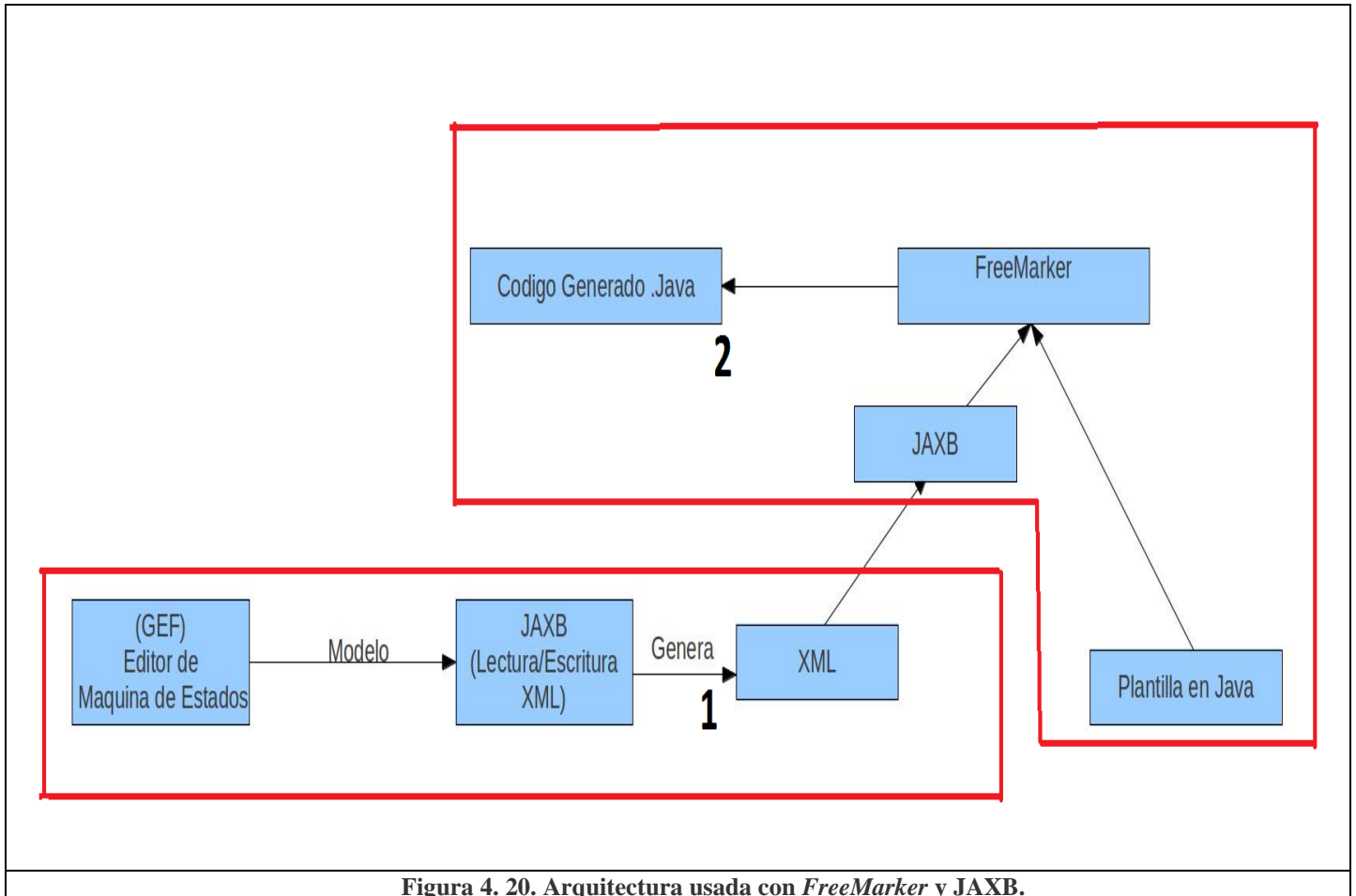


Figura 4. 20. Arquitectura usada con *FreeMarker* y *JAXB*.

En la figura 4.20, se puede observar la arquitectura usada para integrar *JAXB*, *FreeMarker* y el editor de máquina de estados. La sección 1 de la figura 4.20, ilustra como el editor de máquina de estados le provee a *JAXB* un modelo, y con este modelo *JAXB* genera un archivo XML. En la sección 2 de la figura 4.20, es donde el *Plugin GeneratorCodeActionDelegate*, se encarga de utilizar *FreeMarker* para generar el código de la máquina de estados en Java. *FreeMarker* hace uso de *JAXB* para leer el archivo XML, y de una plantilla, en este caso escrita en Java, con el fin de generar el código ejecutable. El *Plugin GeneratorCodeActionDelegate*, es un submenú que se agrega en Eclipse, este se incorporó a la herramienta para poder integrar el editor gráfico con el generador de código.

4.6 State Machine Creation Wizard

En la figura 4.21, se encuentra el diagrama de clases de *StateMachineCreationWizard*, esta clase como se mostró en la sección 4.3, es un *Plugin* de Eclipse, y su responsabilidad es brindar el soporte de un *wizard* para la creación de un nuevo documento de máquina de estados en Eclipse. La clase *StateMachineCreationWizard*, hereda de la clase *Wizard* de Eclipse, esta última brinda toda la funcionalidad para incorporar un nuevo *wizard* en Eclipse, y también implementa la interfaz *INewWizard* de Eclipse, que sirve para inicializar el nuevo *Wizard*. La dependencia con la clase *CreationPage*, es debido a que esta clase es la encargada de configurar una página del *wizard*. Para el caso de este trabajo se utiliza una sola página, *CreationPage*, que hereda de *WizardNewFileCreationPage*, esta última brinda todos los métodos necesarios para ir agregando nuevas páginas en un *wizard*. En la tabla 4.10, se encuentran los métodos más importantes de la clase *StateMachineCreationWizard* y *CreationPage*.

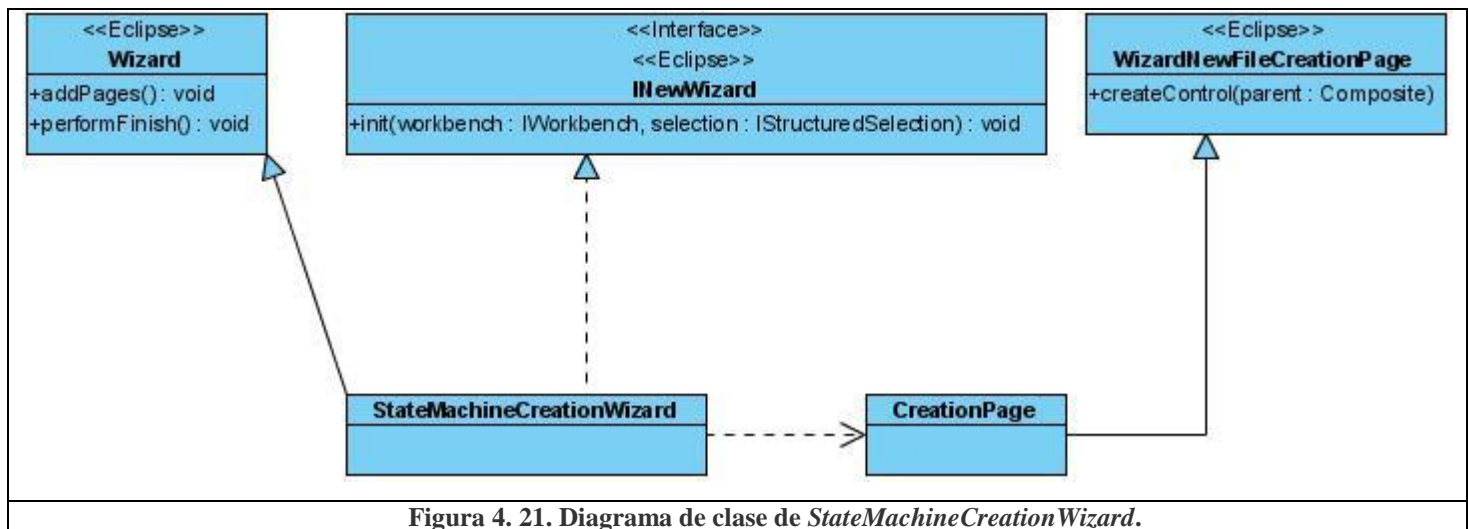


Figura 4. 21. Diagrama de clase de *StateMachineCreationWizard*.

Método	Descripción
Wizard	
<i>init(workbench: IWorkbench, selection: IStructureSelection)</i>	Método que se encarga de inicializar un <i>wizard</i> .
<i>addPages()</i>	Método para agregar una nueva página al <i>wizard</i> .
<i>performFinish()</i>	Método para saber si se presionó el botón finalizar en el <i>wizard</i> .

CreationPage	
<i>createControl(parent: Composite)</i>	Método para configurar una nueva página del wizard.

Tabla 4. 10. Métodos de las clases *Wizard* y *CreationPage*.

4.7 Definición de Máquina de Estados en Java

Como se explicó en el capítulo 2 en la 2.1.2 las máquinas de estados que se están utilizando en este trabajo son las de *Moore* y *Mealy*. Ya a nivel de implementación se necesita darle al desarrollador la suficiente flexibilidad como para que pueda enganchar (*hook*) el código que desea ejecutar, en cualquiera de las salidas mostradas en la figura 4.22. En la tabla 4.11, se explica a detalle cuales son las salidas que le puede dar el desarrollador a la máquina de estados.

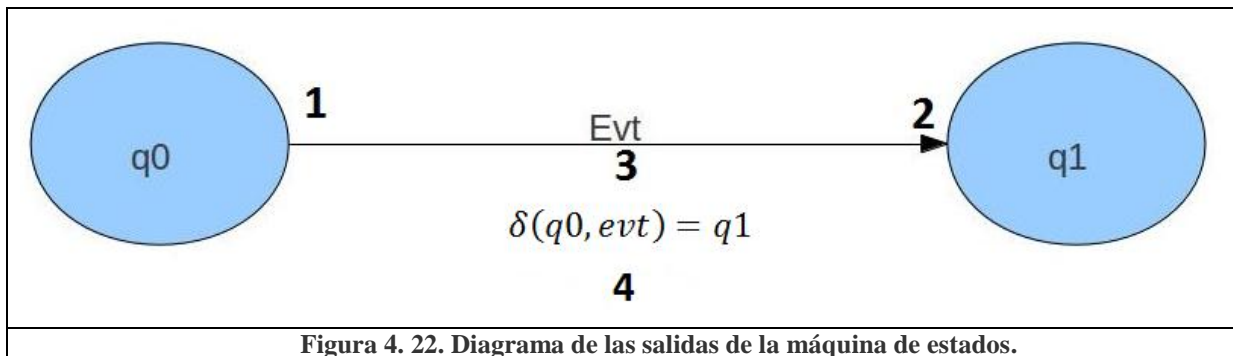


Figura 4. 22. Diagrama de las salidas de la máquina de estados.

Hook	Método	Descripción
1	<i>extStateChange(currentState, targetState, currentEvent)</i>	Este es el método para ejecutar el código a la salida de un estado.
2	<i>entStateChange(currentState, targetState, currentEvent)</i>	Este es el método para ejecutar el código a la entrada de un estado.
3	<i>evtStateChange(currentState, targetState, currentEvent)</i>	Este es el método para ejecutar el código cuando sucede un evento.
4	<i>sourceState_CurrentEvent_targetState()</i>	Este es el método para ejecutar el código cuando hay una transición.

Tabla 4. 11 Hooks de máquina de estados generada.

A continuación, se explicara un pequeño ejemplo para entender cómo se estructura, la máquina de estados generada por la herramienta desarrollada. Básicamente son los estados en los que puede estar una persona, Neutro, Feliz y Triste, los eventos son una Mala Noticia, una Buena Noticia, una Muy Buena o una Muy Mala Noticia. El grafo se muestra en la figura 4.23.

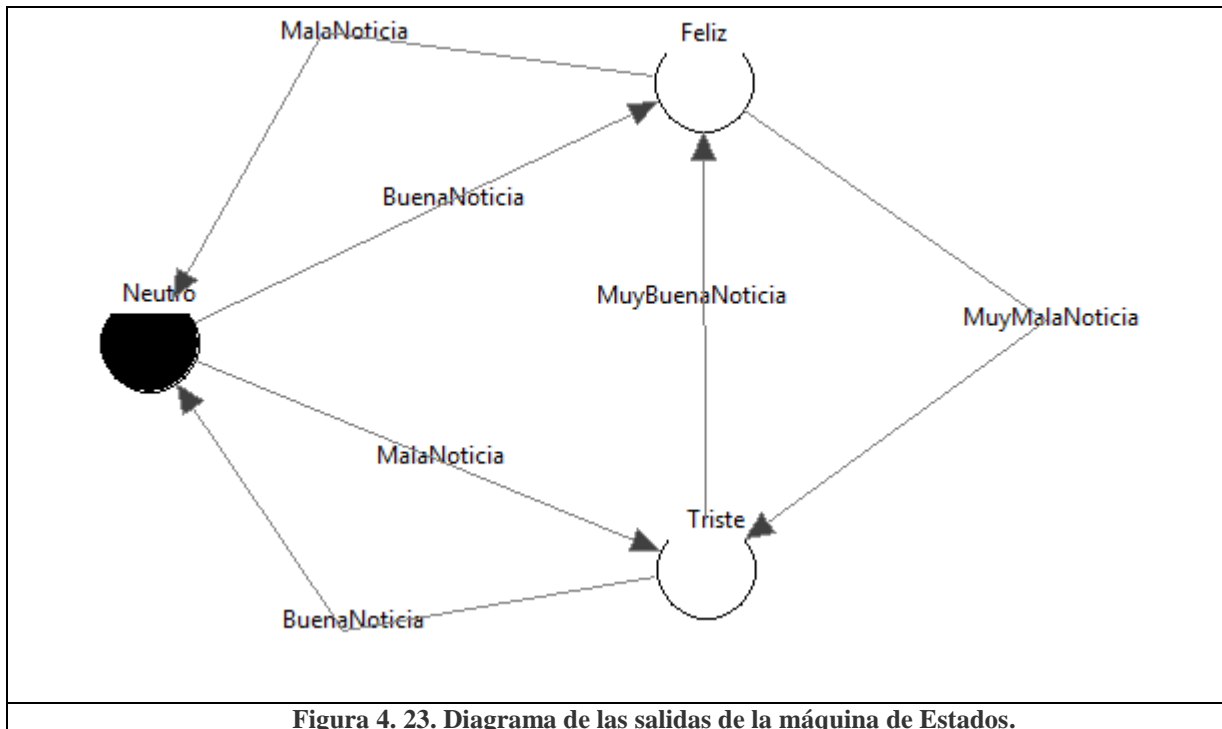


Figura 4. 23. Diagrama de las salidas de la máquina de Estados.

Los estados del ejemplo mostrado en la figura 4.23 se representan con un enumerado como se muestra en la figura 4.24:

```
public enum StateType {  
    Neutro, //  
    Feliz, //  
    Triste, //  
}
```

Figura 4. 24. Código de enumerado de estados.

Los eventos se representan del mismo modo, tal como se ilustra en la figura 4.25:

```
public enum EventType {  
    MuyBuenaNoticia, //  
    MuyMalaNoticia, //  
    BuenaNoticia, //  
    MalaNoticia, //  
}
```

Figura 4. 25. Código de Enumerado de eventos.

Para almacenar el estado actual se tiene una variable de la instancia como se muestra en la figura 4.26:

```
private StateType state = StateType.Neutro;
```

Figura 4. 26. Código de variable para manejar estados.

Esta variable debe empezar en el estado inicial, que en este caso y tal como se muestra en la figura 4.23, el estado inicial es Neutro.

La función de transición que maneja los estados se ilustra en la figura 4.27:

```
public synchronized void fireEvent(EventType event,  
    Map<String, Object> ctx, Map<String, Object> par) {  
  
    if (checkPreFireEvent(event, ctx, par)) {  
        internalFireEvent(event, ctx, par);  
    }  
}
```

Figura 4. 27. Código de función de transición.

Por razones de implementación se tiene un método *internalFireEvent* que es la que realmente hace el trabajo, mostrado en la figura 4.28.

```
protected void internalFireEvent(EventType event,  
    Map<String, Object> ctx, Map<String, Object> par) {  
  
    switch (state) {  
        case Neutro :  
            stateNeutro(event, ctx, par);  
            break;  
        case Feliz :  
            stateFeliz(event, ctx, par);  
            break;  
    }
```



```

        case Triste :
            stateTriste(event, ctx, par);
            break;
        default :
            throwDefaultState(state, event);
            break;
    }
}

```

Figura 4. 28. Código de método *internalFireEvent*.

Podemos ver que el método *internalFireEvent* dependiendo del estado actual tiene ciertas funciones que se encargan de procesar el estado correspondiente. La variable *state* que recibe el *switch* es una variable de la instancia que empieza con el estado inicial. En el código de la figura 4.29 se muestra el método que procesa el estado Neutro. Si llega el evento *MuyBuenaNoticia* el estado no sabe qué hacer y arroja una excepción. El estado sabe procesar los eventos *BuenaNoticia* o *MalaNoticia*. Como se puede ver los métodos explicados en la tabla 4.11 son los que se muestran en este código para cualquiera de los eventos que se pueden manejar desde el estado Neutro.

```

private void stateNeutro(EventType event,
Map<String, Object> ctx, Map<String, Object> par) {
    switch (event) {
        case MuyBuenaNoticia :
            throwIllegalState(state, event);
            break;
        case MuyMalaNoticia :
            throwIllegalState(state, event);
            break;
        case BuenaNoticia :
            stateChangeMsg(StateType.Feliz, event, ctx, par);

            extStateChange(this.state, StateType.Feliz, event, ctx, par);
            evtStateChange(this.state, StateType.Feliz, event, ctx, par);

            Neutro_BuenaNoticia_Feliz();

            entStateChange(this.state, StateType.Feliz, event, ctx, par);

            stateChangeSet(StateType.Feliz, event, ctx, par);
            break;
        case MalaNoticia :

```

```

stateChangeMsg(StateType.Triste, event, ctx, par);

extStateChange(this.state, StateType.Triste, ctx, par);
evtStateChange(this.state, StateType.Triste, ctx, par);

Neutro_MalaNoticia_Triste();

entStateChange(this.state, StateType.Triste, ctx, par);
stateChangeSet(StateType.Triste, event, ctx, par);
break;
default :
    throwDefaultState(state, event);
    break;
}
}

```

Figura 4. 29. Código de método *StateNeutro*.

El *hook* para dar una salida a una transición en el caso del estado *stateNeutro* es el método *Neutro_BuenaNoticia_Feliz* como se muestra en la figura 4.30 este método se encuentra totalmente vacío y se encuentra protegido para ser sobrescrito por el desarrollador.

```

protected void Neutro_BuenaNoticia_Feliz(ctx,par) {
    // Empty
}

```

Figura 4. 30. Código de *hook* para la transición.

Los métodos *extStateChange*, *evtStateChange* y *entStateChange* básicamente tienen la misma estructura, utilizando un *switch*. Existe un método particular para manejar cada caso, por ejemplo, para el caso de *extStateChange* el método se ilustra en la figura 4.31:

```

private void extStateChange(StateType currState, StateType nextState, //
    EventType event, Map<String, Object> ctx, Map<String, Object> par) {

    switch (currState) {
        case Neutro :
            ext_Neutro(currState, nextState, event, ctx, par);
            break;
        case Feliz :
            ext_Feliz(currState, nextState, event, ctx, par);
            break;
        case Triste :
            ext_Triste(currState, nextState, event, ctx, par);
            break;
    }
}

```

```
        default :  
            throwDefaultState(currState, event, ctx, par);  
            break;  
    }  
}
```

Figura 4. 31. Código de método *extStateChange*.

Los métodos *ext_Neutro*, *ext_Feliz*, *ext_Triste* son los métodos que manejan el código que se ejecuta a la salida de un estado y al igual que el método *Neutro_BuenaNoticia_Feliz* se encuentra vacíos y son protegidos. Cada uno de estos métodos los debe sobrescribir el desarrollador dependiendo de sus necesidades. Como se explicó anteriormente los métodos *entStateChange* y *evtStateChange* tienen la misma estructura que *extStateChange*. A continuación se muestra los trozos de código en las figuras 4.32 y 4.33:

```
private void entStateChange(StateType currState, StateType nextState, //  
    EventType event, Map<String, Object> ctx, Map<String, Object> par) {  
  
    switch (nextState) {  
        case Neutro :  
            ent_Neutro(currState, nextState, event, ctx, par);  
            break;  
        case Feliz :  
            ent_Feliz(currState, nextState, event, ctx, par);  
            break;  
        case Triste :  
            ent_Triste(currState, nextState, event, ctx, par);  
            break;  
        default :  
            throwDefaultState(currState, event, ctx, par);  
            break;  
    }  
}
```

Figura 4. 32. Código de método *entStateChange*.

```
private void evtStateChange(StateType currState, StateType nextState, //  
    EventType event, Map<String, Object> ctx, Map<String, Object> par) {  
  
    switch (event) {  
        case MuyBuenaNoticia :  
            evt_MuyBuenaNoticia(currState, nextState, event, ctx, par);  
            break;  
    }  
}
```

```
case MuyMalaNoticia :  
    evt_MuyMalaNoticia(currState, nextState, event, ctx, par);  
    break;  
case BuenaNoticia :  
    evt_BuenaNoticia(currState, nextState, event, ctx, par);  
    break;  
case MalaNoticia :  
    evt_MalaNoticia(currState, nextState, event, ctx, par);  
    break;  
default :  
    throwDefaultState(currState, event, ctx, par);  
    break;  
}  
}
```

Figura 4. 33. Código de método *evtStateChange*.

Como se puede observar en el método *evtStateChange*, el *switch* depende del evento actual. El usuario puede rellenar cualquiera de los métodos que se encuentra en la función *evtStateChange*. Se debe recordar que esta función es la que maneja los *hooks* para cuando sucede se recibe un evento en la máquina de estados. La tabla hash “ctx” que se encuentra en todo el código generado, es una variable opcional para manejar el contexto de la máquina de estados, y la variable “par” es para brindarle al desarrollador un parámetro adicional que necesite utilizar. En la sección 5.2 se puede observar el uso de estas variables.

4.7.1 Eventos internos y externos

Los eventos externos, provienen de clases que se encuentran fuera de la máquina de estados. Por ejemplo en los casos de estudio que se verán en el capítulo 5 en las secciones 5.1 y 5.2, los eventos externos en el caso de la calculadora provienen de las interacciones con la interfaz de usuario, en el caso de MagicRoot provienen de la clase partida, dependiendo de los mensajes que reciba la clase a través de la red, se disparan eventos.

Los eventos internos son un ardid para no perder la consistencia de la máquina de estados, ya que sin ellos se tendría que manipular el estado dentro de cada *hook*. Un ejemplo de cómo funcionan los eventos internos se puede ver en la figura 4.34, los eventos externos están marcados en azul, y los internos en rojo, como se puede observar el evento marcado en

color azul “juegaRojo”, manda a la máquina de estados al estado “ValidarJugadaRojo”. En el *hook* del estado “TurnoRojo”, se encola un evento, ese evento encolado puede ser “cambioTablero”, o “finPartida”. El estado “ValidarJugadaRojo” sabe cómo reaccionar ante los eventos “cambioTablero” o “finPartida”, encolados en el estado “TurnoRojo”, y hará el cambio de estado a “EligiendoTableroRojo”, o “FinPartida” dependiendo de cuál evento fue encolado.

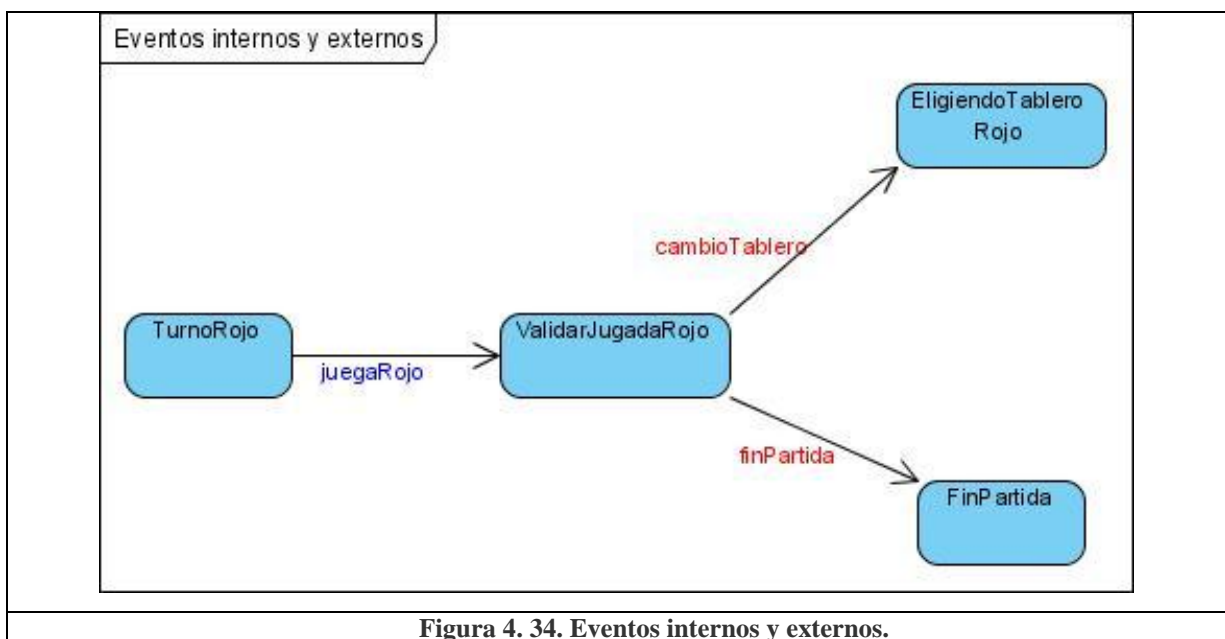


Figura 4. 34. Eventos internos y externos.

Capítulo 5

Casos de estudio

En este capítulo se explicarán dos casos de estudio desarrollados con la herramienta presentada en este trabajo.

5.1 Calculadora

Una calculadora es una de los casos de estudio realizados al editor de máquina de estados, en la figura 5.1 se ilustra la máquina de estados que se encarga de manejar las operaciones y el acumulador de la calculadora. Realizar una calculadora como la que brindan los sistemas operativos Linux o Windows parece una tarea sencilla, pero cuando se toman en cuenta todos los eventos que se pueden dar, se empieza a generar código largo y difícil de entender. En la tabla 5.1, se muestran los estados de la calculadora, y en la tabla 5.2 todo el conjunto de eventos que puede recibir la calculadora.

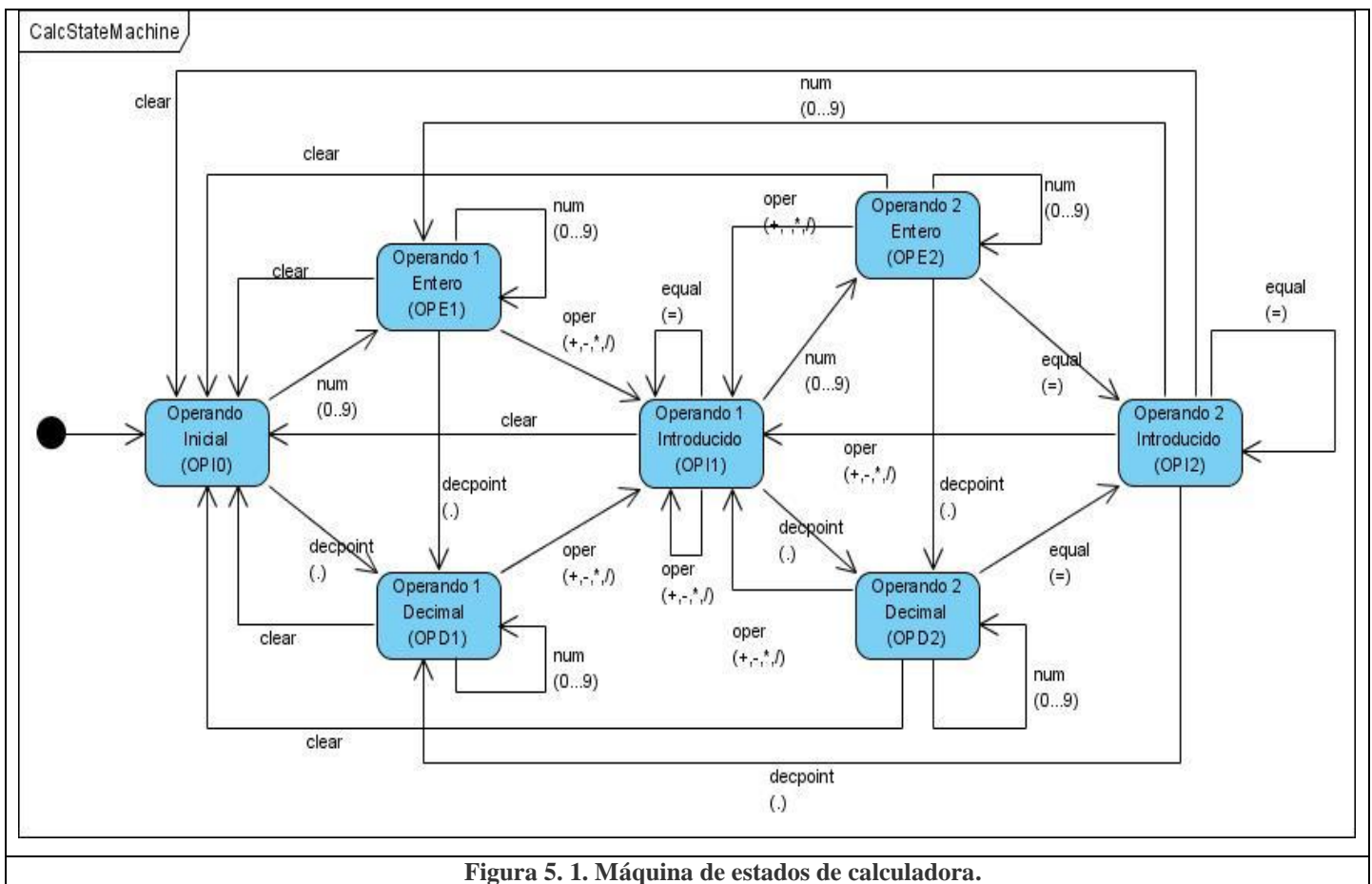


Figura 5. 1. Máquina de estados de calculadora.

Nombre del Estado	Descripción
Operando Inicial (OPI0)	Estado para esperar una entrada del usuario a la calculadora.
Operando 1 Entero (OPE1)	Estado para recibir el primer operador entero de uno o más dígitos.
Operando 1 Decimal (OPD1)	Estado para recibir el primer operador decimal de uno o más dígitos.
Operando Introducido (OPI1)	Cuando se ha introducido el primer operador y se introduce una operación (+, -, *, /).
Operando 2 Entero (OPE2)	Estado para recibir el segundo operador entero de uno o más dígitos.
Operando 2 Decimal (OPD2)	Estado para recibir el segundo operador decimal de uno o más dígitos.
Operando 2 Introducido (OPI2)	Estado para manejar la entrada del evento <i>equal (=)</i> y continuar con el acumulador.

Tabla 5. 1. Estado de la máquina de estados de una calculadora.

Nombre del Evento	Descripción
num	Representa la entrada de un número entre 0 y 9.
oper	Representa una operación de suma, resta, multiplicación o división
decpoint	Representa la entrada de un punto para introducir un numero decimal
clear	Representa la entrada del usuario para borrar la pantalla.

Tabla 5. 2. Eventos de la máquina de estados de una calculadora.

La figura 5.2 ilustra cómo se implementó la máquina de estados de la calculadora, utilizando en editor de máquina de estados. La clase *StateMachineCalculator*, se generó con el editor de máquina de estados, y es la que se encarga de manejar toda la lógica de la máquina de estados como tal, es decir la función de transición, y brindar los *hooks*, para darle las salidas a la máquina de estados.

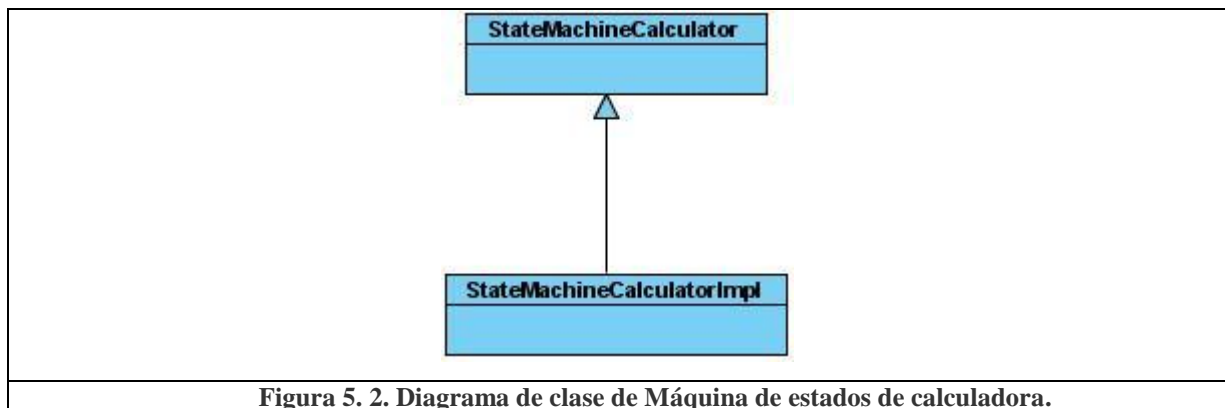


Figura 5. 2. Diagrama de clase de Máquina de estados de calculadora.

El código de la figura 5.3, muestra cómo se procesa el estado Operando 1 Entero (OPE1), el estado tiene aceptación para casi todos los eventos menos para un evento *equal*. El evento *num* se encarga de agregar otros dígitos al operador, y es un bucle a sí mismo como se puede ver en la figura 5.1. El evento *decpoint*, es cuando se introduce un punto en la calculadora, y como se observa en la figura 5.1, se pasa al estado Operando 1 Decimal (OPD1) para manejar números decimales. El evento *oper* es para recibir alguna operación, sea suma, resta, multiplicación o división, y como se puede observar en la figura 5.1, envía a la máquina de estados al estado Operando Introducido (OPI1), por último el evento *clear*, se

encarga de enviar a la calculadora al estado operando inicial (OPI0) para reiniciar el acumulador.

```
private void stateOPE1(EventType event, Map<String, Object> ctx, Map<String,
Object> par) {
    switch (event) {
        case num :
            stateChangeMsg(StateType.OPE1, event, ctx, par);

            extStateChange(this.state, StateType.OPE1, event, ctx, par);
            evtStateChange(this.state, StateType.OPE1, event, ctx, par);

            OPE1_num_OPE1(ctx, par);

            entStateChange(this.state, StateType.OPE1, event, ctx, par);

            stateChangeSet(StateType.OPE1, event, ctx, par);
            break;
        case decpoint :
            stateChangeMsg(StateType.OPD1, event, ctx, par);

            extStateChange(this.state, StateType.OPD1, event, ctx, par);
            evtStateChange(this.state, StateType.OPD1, event, ctx, par);

            OPE1_decpoint_OPD1(ctx, par);

            entStateChange(this.state, StateType.OPD1, event, ctx, par);

            stateChangeSet(StateType.OPD1, event, ctx, par);
            break;
        case oper :
            stateChangeMsg(StateType.OPI1, event, ctx, par);

            extStateChange(this.state, StateType.OPI1, event, ctx, par);
            evtStateChange(this.state, StateType.OPI1, event, ctx, par);

            OPE1_oper_OPI1(ctx, par);

            entStateChange(this.state, StateType.OPI1, event, ctx, par);

            stateChangeSet(StateType.OPI1, event, ctx, par);
            break;
        case equal :
            throwIllegalState(state, event);
            break;
        case clear :
            stateChangeMsg(StateType.OPI0, event, ctx, par);
```

```

    extStateChange(this.state, StateType.OPI0, event, ctx, par);
    evtStateChange(this.state, StateType.OPI0, event, ctx, par);

    OPE1_clear_OPI0(ctx, par);

    entStateChange(this.state, StateType.OPI0, event, ctx, par);

    stateChangeSet(StateType.OPI0, event, ctx, par);
    break;
default :
    throwDefaultState(state, event);
    break;
}
}

```

Figura 5. 3. Código de método StateOPE1.

La figura 5.4, ilustra el método para dar la salida a la transición para el estado operando 1 entero (OPE1), cuando llega el evento *num*. La idea es ir agregando cualquier cantidad de dígitos que introduzca el usuario. Como muestra la figura 5.5 el método *addnum* se encarga de concatenar números, si lo que se encuentra en la pantalla de la calculadora es diferente de 0, en caso de no ser diferente de 0, agrega el número introducido por el usuario a la pantalla de la calculadora.

```

protected void OPE1_num_OPE1(Map<String, Object> ctx, Map<String, Object> par) {
    addNum(false, ctx, par);
}

```

Figura 5. 4. Hook para el estado Operando 1 Entero (OPE1).

```

private void addNum(boolean clearDisplay, //
    Map<String, Object> ctx, Map<String, Object> par) {

    if (clearDisplay) {
        clearDisplay();
    }

    if (!txtDisplay.getText().equals("0")) {
        txtDisplay.setText(txtDisplay.getText() + par.get(PARAMETER_KEY));
    }
}

```

```
    } else {  
        txtDisplay.setText(par.get(PARAMETER_KEY).toString());  
    }  
}
```

Figura 5. 5. Método *addNum*.

En el código de la figura 5.6 se ilustra el método que dispara los eventos externos de la máquina de estados. Dependiendo del evento que llegue desde la interfaz de usuario se llamara al método *fireEvent*. La variable “par” se está utilizando para guardar la operación que se desea realizar, sea suma resta multiplicación o división. Este método se encuentra en la clase, que se encarga del manejo de la interfaz de usuario de la calculadora.

```
private void handleEvent(ActionEvent evt) {  
    String com = evt.getActionCommand().substring(0, 4);  
    String par = evt.getActionCommand().substring(5, 6);  
  
    Map<String, Object> parMap = new HashMap<String, Object>();  
    parMap.put(CalcImpl.PARAMETER_KEY, par);  
  
    try {  
        /* */if (com.equals("numb")) {  
            calcImpl.fireEvent(EventType.num, null, parMap);  
        } else if (com.equals("oper")) {  
            calcImpl.fireEvent(EventType.oper, null, parMap);  
        } else if (com.equals("decp")) {  
            calcImpl.fireEvent(EventType.decpoint, null, parMap);  
        } else if (com.equals("equa")) {  
            calcImpl.fireEvent(EventType.equal, null, parMap);  
        } else if (com.equals("clea")) {  
            if (par.equals("A")) {  
                calcImpl = new CalcImpl(txtDisplay);  
                txtDisplay.setText("");  
                return;  
            }  
            calcImpl.fireEvent(EventType.clear, null, parMap);  
        }  
    } catch (IllegalStateException e) {  
        System.err.println(e.getMessage());  
    }  
}
```

Figura 5. 6. Metodo *handleEvent*.

5.2 MagicRoot

MagicRoot es uno de los casos de estudio realizados a la herramienta presentada en este trabajo. MagicRoot es un juego de cartas multijugador para *Android* desarrollado en la tesis de (Zamora, 2012).

La figura 5.7 muestra la arquitectura general MagicRoot. Existe un cliente que es la interfaz de usuario desarrollado en *swing*, y se encarga de recibir comandos que vienen desde el servidor a través de la red, dependiendo del comando que reciba, actualizará la interfaz de usuario. El servidor es el encargado de manejar a cada cliente que se conecta y de acceder a la base de datos. En el servidor existe una clase Partida, que tiene como responsabilidad manejar toda la lógica de una partida entre dos jugadores. La clase Tablero, se encarga de llevar toda la lógica del juego como tal. La máquina de estados es la encargada de manejar los turnos en una partida, es decir, a quien le corresponde jugar en que momento, y dependiendo del estado en que se encuentre, hace llamadas a la clase tablero para que ésta realice cierta acción en el juego.

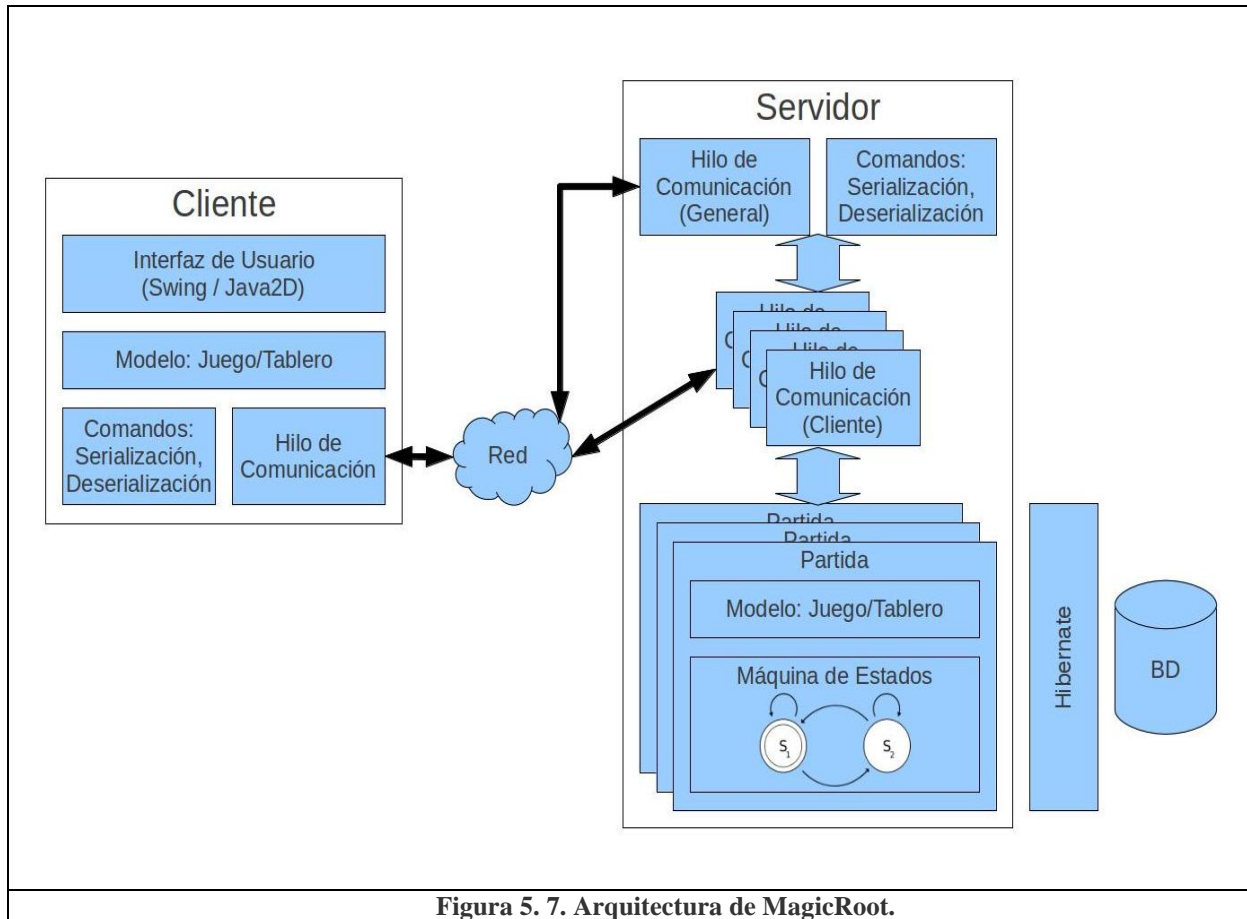


Figura 5. 7. Arquitectura de MagicRoot.

La máquina de estados de MagicRoot se encuentra en la figura 5.8. Los turnos entre los jugadores los manejan los estados “Turno Rojo” y “Turno Azul”, los estados “Validar jugada Rojo” y “Validar jugada Azul” se utilizan para validar una jugada realizada por un jugador. Los estados “Elegiendo tablero Azul” y “Elegiendo tablero Rojo”, se encargan de manejar un cambio de tablero, y luego de que el jugador escoja el tablero nuevo, asignar el turno al jugador correspondiente. En cada turno existe un evento *Time Out*, que es un *timer* para validar que un jugador no deje a otro esperando mucho tiempo.

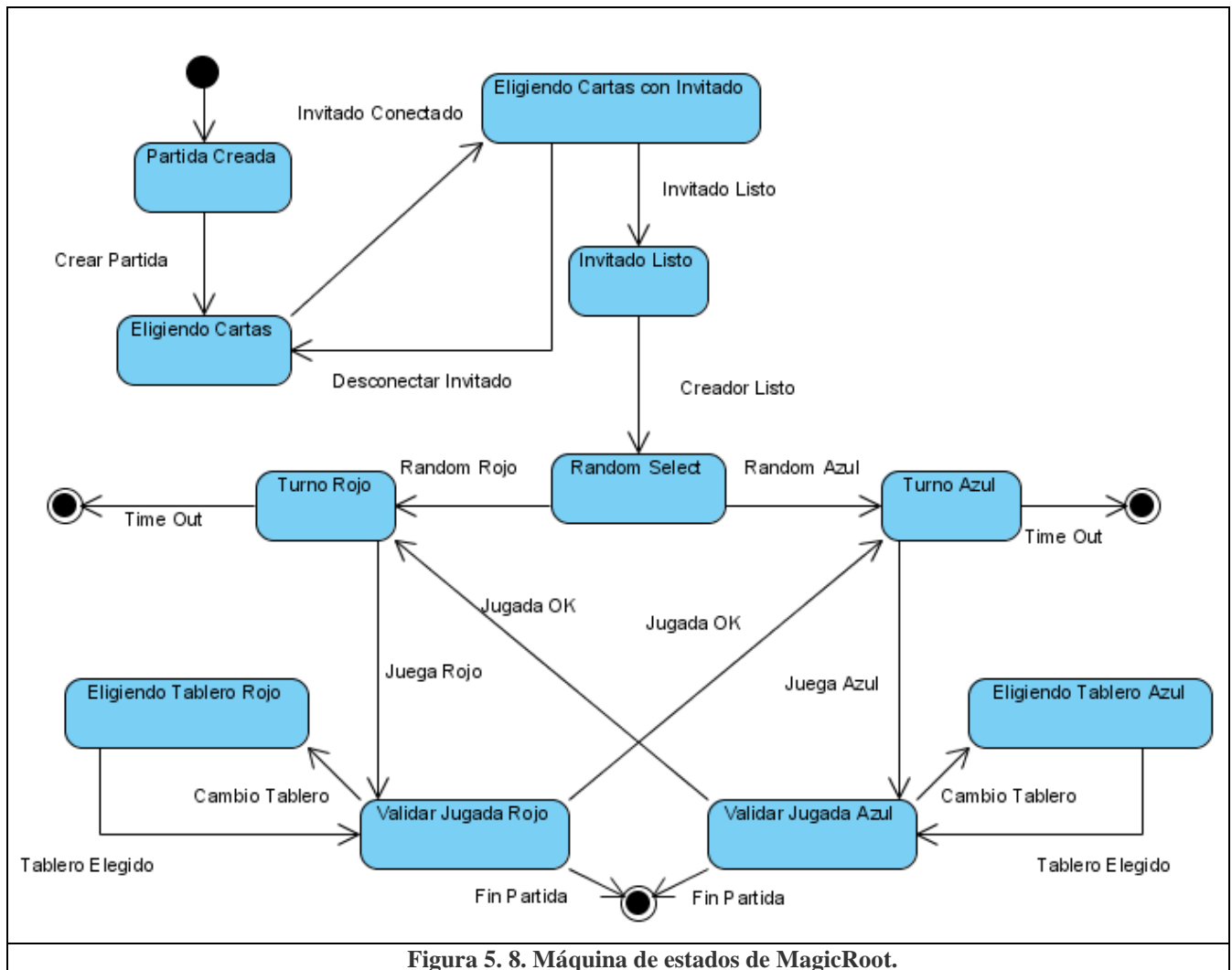


Figura 5. 8. Máquina de estados de MagicRoot.

La función de transición de la máquina de estados de MagicRoot sin usar el editor de máquina de estados, básicamente es la misma que la mostrada en el capítulo 4 sección 4.7. El código de la figura 5.9, ilustra el método para procesar el estado “turno rojo” sin usar el editor de máquina de estados, todo el método fue codificado explícitamente. El método se encarga de recibir el evento actual y dependiendo de eso ejecutará un determinado código, finalmente actualizará el estado actual. El evento “juega rojo” es un evento externo a la máquina de estados, que llega desde la clase partida, cuando llega el evento juega rojo, el método llama a la clase tablero para procesar una jugada, envía un comando a través de la

partida para que luego el cliente refleje un cambio, y por ultimo dependiendo del estado del tablero, dispara un evento interno en la máquina de estados. El evento “fin partida” se encarga de mandar a la máquina de estados a un estado final donde termina la partida, disparando un evento interno. Por último el evento *time out* se utiliza para manejar el *timer*, en caso de ser disparado el *timer*, éste evento envía a la máquina de estados a un estado final.

```
private void estadoTurnR(TipoDeEvento evento, Map<String, Object> ctx,
    Map<String, Object> par) throws UnknowEventForState,
    IdCreadorInvalido {
    switch (evento) {
    case finPartida:
        disparaEvento(TipoDeEvento.finPartida, null, null);
        estado = TipoDeEstado.ESTADO_FINAL;
        break;
    case juega_r:
        int x = (Integer) par.get(Card.ROW);
        int y = (Integer) par.get(Card.COLUMN);
        Card carta = new Card();
        carta = (Card) par.get(Card.CARD);
        if (getTablero(ctx).setCard(x, y, carta)) {
            Command cartaComando =
                getTablero(ctx).crearCartaCommand(carta);

            Command comando = new JugarCommand(getPartida(ctx)
                .getIdHiloAzul(), carta.getIdCarta(), x, y,
                cartaComando);
            getPartida(ctx).enviarMsjJugEspera(comando);

            if (getTablero(ctx).isFull()) {
                disparaEvento(TipoDeEvento.finPartida, ctx, null);
            } else if (getTablero(ctx).cambio()) {
                disparaEvento(TipoDeEvento.cambioTablero, ctx, null);
            } else {
                disparaEvento(TipoDeEvento.jugadaOK, null, null);
            }
        }
        estado = TipoDeEstado.VALIDAJUGADA_R;
        break;
    case timeOut:
        disparaEvento(TipoDeEvento.timeOut, ctx, par);
        estado = TipoDeEstado.ROJO_GANADOR;
    default:
        getPartida(ctx).enviar(
            "Error!!!! accion(evento) no valida...(estado) ",
            getPartida(ctx).getIdHiloRojo());
    }
```

```
        estadoIllegal(TipoDeEstado.TURNR, evento);  
        break;  
    }  
}
```

Figura 5. 9. Código de método estadoTurnoR.

En la figura 5.10, se muestra como se estructuró la máquina de estados de MagicRoot utilizando el editor de máquina de estados. La clase *StateMachineMagicRoot*, es la generada por la herramienta presentada en este trabajo, y se encarga de llevar toda la lógica de la máquina de estados. La clase *StateMachineMagicRootImpl*, es la que se encarga de sobrescribir los *hooks* necesarios, para darle una salida a la máquina de estados de MagicRoot, en MagicRoot las salidas se encuentran asociadas a las transiciones y los estados.

El código de la figura 5.11, ilustra el funcionamiento del método para procesar el estado “turno rojo” de la máquina de estados de MagicRoot, usando el editor/generador de máquina de estados. Todo el método es código generado, y dependiendo del evento actual, éste ofrece los *hooks* mencionados en el capítulo 4 en la sección 4.7. En el caso de este método se utilizaran los *hooks* asociados a las transiciones para los eventos “juega rojo”, “fin partida” y “time out”, el código que ejecuta cada uno de estos *hooks* son es exactamente igual al que se encuentran en el método de la figura 5.9. El código de la figura 5.11 no debe ser editado por el desarrollador ya que éste es código generado. Si llega algún evento que no sabe procesar el estado “turno rojo”, el método arroja una excepción, la tabla hash “ctx” que recibe el método, como se explicó en el capítulo 4 sección 4.7, es una variable para manejar el contexto en el que se va a usar en la máquina de estados, en el caso particular de MagicRoot es la clase partida. La variable “par” se utiliza para manejar otro parámetro adicional que se quiera usar, en este caso de estudio son todas las cartas.

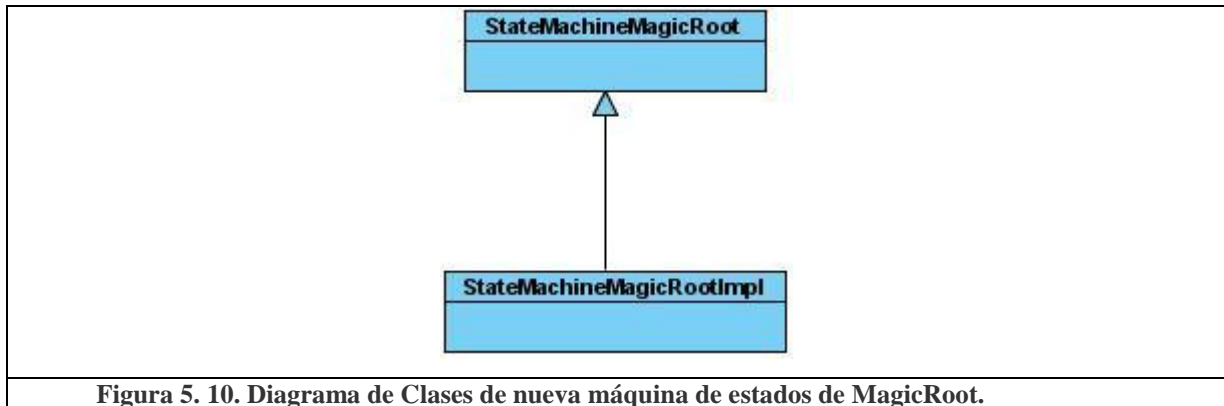


Figura 5. 10. Diagrama de Clases de nueva máquina de estados de MagicRoot.

```

private void stateTURNO_ROJO(EventType event, Map<String, Object> ctx, Map<String,
Object> par) {
    switch (event) {
        case CrearPartida :
            throwIllegalState(state, event);
            break;
        case CreadorListo :
            throwIllegalState(state, event);
            break;
        case InvitadoListo :
            throwIllegalState(state, event);
            break;
        case RandomRojo :
            throwIllegalState(state, event);
            break;
        case RandomAzul :
            throwIllegalState(state, event);
            break;
        case JugadaOk :
            throwIllegalState(state, event);
            break;
        case JuegoRojo :
            stateChangeMsg(StateType.VALIDAR_JUGADA_ROJO, event, ctx, par);

            extStateChange(this.state, StateType.VALIDAR_JUGADA_ROJO, event, ctx, par);
            evtStateChange(this.state, StateType.VALIDAR_JUGADA_ROJO, event, ctx, par);

            TURNO_ROJO_JuegoRojo_VALIDAR_JUGADA_ROJO(ctx, par);

            entStateChange(this.state, StateType.VALIDAR_JUGADA_ROJO, event, ctx, par);

            stateChangeSet(StateType.VALIDAR_JUGADA_ROJO, event, ctx, par);
    }
}

```

```

        break;
    case JuegaAzul :
        throwIllegalState(state, event);
        break;
    case TableroElegido :
        throwIllegalState(state, event);
        break;
    case CambioTablero :
        throwIllegalState(state, event);
        break;
    case FinPartida :
        stateChangeMsg(StateType.ESTADO_FIN_PARTIDA, event, ctx, par);

        extStateChange(this.state, StateType.ESTADO_FIN_PARTIDA, event, ctx, par);
        evtStateChange(this.state, StateType.ESTADO_FIN_PARTIDA, event, ctx, par);

        TURNO_ROJO_FinPartida_ESTADO_FIN_PARTIDA(ctx, par);

        entStateChange(this.state, StateType.ESTADO_FIN_PARTIDA, event, ctx, par);

        stateChangeSet(StateType.ESTADO_FIN_PARTIDA, event, ctx, par);
        break;
    case TimeOut :
        stateChangeMsg(StateType.AZULGANADOR, event, ctx, par);

        extStateChange(this.state, StateType.AZULGANADOR, event, ctx, par);
        evtStateChange(this.state, StateType.AZULGANADOR, event, ctx, par);

        TURNO_ROJO_TimeOut_AZULGANADOR(ctx, par);

        entStateChange(this.state, StateType.AZULGANADOR, event, ctx, par);

        stateChangeSet(StateType.AZULGANADOR, event, ctx, par);
        break;
    case Inicio :
        throwIllegalState(state, event);
        break;
    case LlegaInvitado :
        throwIllegalState(state, event);
        break;
    case invitadoDesconectado :
        stateChangeMsg(StateType.AZULGANADOR, event, ctx, par);

        extStateChange(this.state, StateType.AZULGANADOR, event, ctx, par);
        evtStateChange(this.state, StateType.AZULGANADOR, event, ctx, par);

        TURNO_ROJO_invitadoDesconectado_AZULGANADOR(ctx, par);

        entStateChange(this.state, StateType.AZULGANADOR, event, ctx, par);

```

```

        stateChangeSet(StateType.AZULGANADOR, event, ctx, par);
        break;
    case creadorDesconectado :
        stateChangeMsg(StateType.ROJOGANADOR, event, ctx, par);

        extStateChange(this.state, StateType.ROJOGANADOR, event, ctx, par);
        evtStateChange(this.state, StateType.ROJOGANADOR, event, ctx, par);

        TURNO_ROJO_creadorDesconectado_ROJOGANADOR(ctx, par);

        entStateChange(this.state, StateType.ROJOGANADOR, event, ctx, par);

        stateChangeSet(StateType.ROJOGANADOR, event, ctx, par);
        break;
    case jugadaInvalida :
        throwIllegalState(state, event);
        break;
    default :
        throwDefaultState(state, event);
        break;
    }
}

```

Figura 5. 11. Código de método *stateTurnoRojo*.

En la figura 5.12 se ilustra el código que se ejecuta asociado a la transición entre el estado “turno rojo”, y el evento “juega rojo”, enviando la máquina de estados a el estado “validar jugada rojo”. Como se puede ver, es el mismo código que se ejecuta en la figura 5.10 para el evento juega rojo, la ventaja principal, es que el desarrollador ahora en la clase *StateMachineImpl*, tiene todos los *hooks* que va a utilizar para las salidas asociadas a un estado o a una transición, sin tener que preocuparse por el manejo de la lógica de la máquina de estados, ya que eso es responsabilidad de la clase generada por el editor.

```

@Override
    protected void TURNO_ROJO_JuegaRojo_VALIDAR_JUGADA_ROJO(
        Map<String, Object> ctx, Map<String, Object> par) {
        super.TURNO_ROJO_JuegaRojo_VALIDAR_JUGADA_ROJO(ctx, par);
        int x = (Integer) par.get("row");
        int y = (Integer) par.get("column");
        Card carta = new Card();
        carta = (Card) par.get("carta");

        if (getTablero(ctx).setCard(x, y, carta)) {
            Command cartaComando = getTablero(ctx).crearCartaCommand(carta);

```

```

        Command comando = new JugarCommand(getPartida(ctx).getIdHiloAzul(),
            carta.getIdCarta(), x, y, cartaComando);
        getPartida(ctx).enviarMsjJugEspera(comando);
        if (getTablero(ctx).isFull()) {
            fireEvent(EventType.FinPartida, ctx, null);
        } else if (getTablero(ctx).cambio()) {

            fireEvent(EventType.CambioTablero, ctx, null);

        } else {

            fireEvent(EventType.JugadaOk, null, null);

        }

    } else {
        getPartida(ctx).enviarHiloRojo("jugadaInvalidaCommand");
        fireEvent(EventType.jugadaInvalida, ctx, par);
    }
}

```

Figura 5. 12. Hook implementado en la clase StateMachineMagicRootImpl.

El código de la figura 5.13, es el método de la clase partida que se encarga de disparar un evento externo a la máquina de estados, cuando el jugador invitado está listo para empezar una partida. La clase partida tiene una referencia a la clase *StateMachineMagicRootImpl* llamada máquina, utilizando esta referencia se llama al método *fireEvent* pasándole como parámetro el evento “InvitadoListo”.

```

private void initListo(Command comando) {

    if (comando.getId() == getIdHiloRojo()) {
        maquina.fireEvent(EventType.InvitadoListo, createCtx(), null);
    } else {
        enviar("#2Wrong ID ", getIdHiloAzul());
        enviar("#2Wrong ID", getIdHiloRojo());
    }

}

```

Figura 5. 13. Método initListo.

Para concluir el capítulo se puede decir que cada caso de estudio tiene su particularidad. Con la calculadora queda demostrado, lo sencillo que puede ser codificar un problema de este estilo utilizando una máquina de estados. Codificando este problema de la manera clásica, probablemente el código de la calculadora va a ser muy difícil de mantener y de poca calidad. Como se puede apreciar en la sección 5.1, lo que realmente tuvo que codificar el desarrollador, fueron algunos métodos para darle una salida a la máquina de estados. Toda la lógica de la máquina de estados se encapsula en la clase generada por el editor de máquina de estados.

Por otra parte, con el caso de estudio MagicRoot, quedo demostrado que el uso de herramientas como la presentada en este trabajo, permite a los desarrolladores concentrarse en el problema que están resolviendo, y no en cómo implementar una máquina de estados. Además el uso de este tipo de herramientas, permite a los desarrolladores hacer cambios fácilmente en el código, dándole mayor flexibilidad y mantenibilidad a los productos desarrollados.

Con los casos de estudio implementados se realizaron las pruebas caja negra add hoc. Básicamente se diseñaron las máquinas de estados para los casos de estudio, y se fueron realizando las pruebas sobre las máquinas de estados generadas por la herramienta presentada en este trabajo, y a las aplicaciones que hacen uso de dichas máquinas de estados, es decir la calculadora y MagicRoot.

Capítulo 6

Conclusiones y Recomendaciones

En este capítulo se darán las conclusiones y recomendaciones de este proyecto de grado.

6.1 Conclusiones

El uso de los formalismos correctos, y de herramientas que permitan incorporar de manera directa estos formalismos, a productos de software que implementen sistemas a eventos discretos, puede reducir los tiempos y costos de desarrollo, y mejorar la mantenibilidad y la calidad de los productos desarrollados. Con este proyecto, se logra desarrollar un editor gráfico de máquina de estados, que genera el código ejecutable en Java de dicha máquina de estados, por medio de la integración de las tecnologías JAXB, *FreeMarker*, y el *framework* GEF.

De igual manera, se consolida la funcionalidad del editor gráfico, con el desarrollo de los dos casos de estudio (MagicRoot y la calculadora), confirmando con estos, que el uso de los formalismos correctos, mejora la calidad y mantenibilidad de los productos desarrollados. Se debe agregar, que el producto desarrollado abre una puerta en la Facultad de Ingeniería ULA, para el desarrollo de *Plugins* de Eclipse, específicamente editores gráficos, debido a que con este trabajo se consolida una base teórica bastante completa para el uso del *framework* GEF.

6.2 Recomendaciones y Trabajos Futuros

Este trabajo es una primer versión del editor de máquina de estados, existen muchas mejoras que se le pueden agregar a este trabajo, actualmente el editor solo genera código ejecutable en Java, en un trabajo futuro se puede desarrollar una versión más genérica que brinde soporte para diferentes lenguajes sea C++, PHP, Python entre otros. Ya existe un dialogo que permite al desarrollador agregar un plantilla de *FreeMarker* diferente a la que se encuentra por defecto, pudiendo esta plantilla estar escrita en otro lenguaje.

De igual manera, otra mejora que se le puede realizar al editor de máquina de estados, es que a un arco se le puedan agregar varios eventos, esta primera versión soporta un solo evento para un arco, esta decisión de diseño se tomó por razones de interfaz de usuario.

También se podría trabajar en agregar a la herramienta un simulador de autómatas, donde el usuario pueda especificar una cadena de entrada, y pueda observar la salida para realizar pruebas mucho más rápido.

Por otra parte, se recomienda el desarrollo de un manual para el uso de la herramienta, así como la incorporación de un módulo de ayuda a la misma.

En próximos trabajos, se puede incorporar soporte para otros tipos de autómatas, como lo son los autómatas de pila, los autómatas finitos no deterministas, entre otros. Ya que la herramienta solo tiene soporte para autómatas finitos deterministas.

Referencias

- Clayberg, E., & Rubel, D. (2008). *eclipse Plug-ins*. Boston: Addison Wesley.
- Darovsky. (2011). *FSME*. Recuperado el 12 de Octubre de 2012, de <http://sourceforge.net/projects/fsme/files/fsme/fsme-1.0.6/>
- Duffner, S., & Strobel, R. (2001). *QFSM*. Recuperado el 12 de Octubre de 2012, de <http://www.qfsm.sourceforge.net>
- Eclipse. (2012). *help Eclipse*. Recuperado el 5 de Diciembre de 2012, de <http://help.eclipse.org/juno/index.jsp>
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2003). *Patrones de Diseño*. Madrid: Addison Wesley.
- Goldberg, A., & Robson, D. (1983). *Smalltalk-80. The Language and Its Implementation*. MA: Addison Wesley.
- Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2002). *Introducción a la teoría de autómatas, lenguajes y computación*. Madrid: Addison Wesley.
- Moral, S. (2001). Recuperado el 15 de Noviembre de 2012, de <ftp://decsai.ugr.es/pub/utai/other/smc/docencia/automata.pdf>
- Rodger. (2009). *Jflap*. Recuperado el 12 de Octubre de 2012, de <http://www.jflap.org>
- Rubel, D., Wren, J., & Clayberg, E. (2011). *The Eclipse Graphical Editor Framework (GEF)*. Indiana: Addison Wesley.
- Sommerville, I. (2007). *Ingeniería del software*. Madrid: Addison Wesley.
- Wikipedia. (2012). *Wikipedia*. Recuperado el 15 de Noviembre de 2012, de http://en.wikipedia.org/wiki/Automata-based_programming
- Wirfs-Brock, R., & Johnson, R. E. (1990). Surveying current research in object-oriented design. *Comm.ACM*, 104-24.
- Zamora, A. (2012). *Desarrollo de Aplicaciones Móviles para Android: MagicRoot, un caso de estudio*. Mérida, Venezuela: Universidad de los Andes.

Apéndice A

XSD utilizado para el modelo de la máquina de estados

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.statemachine.org/state_machine.xsd"
  xmlns="http://www.statemachine.org/state_machine.xsd">

  <xs:element name="state_machine">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="eventSM" maxOccurs="unbounded" />
        <xs:element ref="state" maxOccurs="unbounded" />
        <xs:element ref="initial" />
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required" />
      <xs:attribute name="package" type="xs:string" use="required" />
    </xs:complexType>
  </xs:element>

  <xs:element name="initial">
    <xs:complexType>
      <xs:attribute name="name" type="xs:string" use="required" />
      <xs:attribute name="type" type="xs:string" use="required" />
      <xs:attribute name="x" type="xs:int" use="required" />
      <xs:attribute name="y" type="xs:int" use="required" />
      <xs:attribute name="w" type="xs:int" use="required" />
      <xs:attribute name="h" type="xs:int" use="required" />
    </xs:complexType>
  </xs:element>

  <xs:element name="state">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Arc" maxOccurs="unbounded" />
        <xs:element ref="labelState" />
      </xs:sequence>
      <xs:attribute name="type" type="xs:string" use="required" />
      <xs:attribute name="x" type="xs:int" use="required" />
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```

        <xs:attribute name="y" type="xs:int" use="required" />
        <xs:attribute name="w" type="xs:int" use="required" />
        <xs:attribute name="h" type="xs:int" use="required" />
    </xs:complexType>
</xs:element>

<xs:element name="Arc">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="source" />
            <xs:element ref="target" />
            <xs:element ref="eventArc" />
            <xs:element ref="bendPoints" maxOccurs="unbounded" />
        </xs:sequence>
    </xs:complexType>
</xs:element>

<xs:element name="source">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="state" />
        </xs:sequence>
    </xs:complexType>
</xs:element>

<xs:element name="target">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="state" />
        </xs:sequence>
    </xs:complexType>
</xs:element>

<xs:element name="eventSM">
    <xs:complexType>
        <xs:attribute name="name" type="xs:string" use="required" />
    </xs:complexType>
</xs:element>

<xs:element name="eventArc">
    <xs:complexType>
        <xs:attribute name="name" type="xs:string" use="required" />
        <xs:attribute name="x" type="xs:int" use="required" />
        <xs:attribute name="y" type="xs:int" use="required" />
    </xs:complexType>
</xs:element>

```

```
<xs:element name="LabelState">
  <xs:complexType>
    <xs:attribute name="name" type="xs:string" use="required" />
    <xs:attribute name="x" type="xs:int" use="required" />
    <xs:attribute name="y" type="xs:int" use="required" />
  </xs:complexType>
</xs:element>

<xs:element name="bendPoints">
  <xs:complexType>
    <xs:attribute name="d1W" type="xs:int" use="required" />
    <xs:attribute name="d1H" type="xs:int" use="required" />
    <xs:attribute name="d2W" type="xs:int" use="required" />
    <xs:attribute name="d2H" type="xs:int" use="required" />
  </xs:complexType>
</xs:element>

</xs:schema>
```

Figura A.1. XSD utilizado para el modelo de la máquina de estados.

Plantilla de FreeMarker

```
[#ftl]
// -----
--
// Generated code, do not edit
// -----

package ${package};

import java.util.ArrayList;
import java.util.List;
import java.util.Map;

public class ${name} {

    // -----

    public enum StateType {
        [#list state as currentState]
        ${currentState.labelState.name}, //
        [/#list]
    }

    // -----

    public enum EventType {
        [#list eventSM as currentEvent]
        ${currentEvent.name}, //
        [/#list]
    }

    // -----

    private StateType state = StateType.${initial.name};

    // -----
    protected List<EventType> eventQueue = new ArrayList<EventType>();

    protected boolean firing;

    // -----

    //Funcion de Transicion
```

```

    public synchronized void fireEvent(EventType event, //
        Map<String, Object> ctx, Map<String, Object> par) {

        if (checkPreFireEvent(event, ctx, par)) {
            internalFireEvent(event, ctx, par);
        }
    }
    // -----

    public StateType getCurrentState(){
        return state;
    }

    // -----

    protected void internalFireEvent(EventType event, //
        Map<String, Object> ctx, Map<String, Object> par) {

        if (firing) {
            eventQueue.add(event); // TODO: add par to queue
            return;
        }

        firing = true;

        System.err.println("firing " + //
            event + " event, in state " + state);

        switch (state) {
            [#list state as currentState]
            case ${currentState.labelState.name} :
                state${currentState.labelState.name}(event, ctx, par);
                break;
            [/#list]
            default :
                throwDefaultState(state, event);
                break;
        }

        firing = false;

        if (!eventQueue.isEmpty()) {
            EventType eventType = eventQueue.remove(0);
            fireEvent(eventType, ctx, null);
        }
    }

    // -----
    protected boolean checkPreFireEvent(EventType event, //
        Map<String, Object> ctx, Map<String, Object> par) {

```

```

    return true;
}

[#list state as currentState]
// -----
private void state${currentState.labelState.name}(EventType event, Map<String,
Object> ctx, Map<String, Object> par) {
    switch (event) {
        [#list eventSM as currentEvent]
        [#assign eventFound=false]
        [#list currentState.arc as currentArc]
        [#if currentArc.eventArc.name == currentEvent.name]
        [#assign eventFound=true]
        case ${currentArc.eventArc.name} :
            stateChangeMsg(StateType.${currentArc.target.state.labelState.name},
event, ctx, par);

            extStateChange(this.state,
StateType.${currentArc.target.state.labelState.name}, event, ctx, par);
            evtStateChange(this.state,
StateType.${currentArc.target.state.labelState.name}, event, ctx, par);

            ${currentState.labelState.name}_${currentArc.eventArc.name}_${currentArc.target.s
tate.labelState.name}(ctx, par);

            entStateChange(this.state,
StateType.${currentArc.target.state.labelState.name}, event, ctx, par);

            stateChangeSet(StateType.${currentArc.target.state.labelState.name},
event, ctx, par);
            break;
        [/#if]
        [/#list]
        [#if eventFound == false]
        case ${currentEvent.name} :
            throwIllegalState(state, event);
            break;
        [/#if]
        [/#list]
        default :
            throwDefaultState(state, event);
            break;
    }
}

[/#list]
[#list state as currentState]
// -----

```

```

    protected void ent_${currentState.labelState.name}(StateType currState,
StateType nextState, //
        EventType event, Map<String, Object> ctx, Map<String, Object> par) {
        // Empty
    }

    protected void ext_${currentState.labelState.name}(StateType currState,
StateType nextState, //
        EventType event, Map<String, Object> ctx, Map<String, Object> par) {
        // Empty
    }

[/#list]
[#list eventSM as currentEvent]
// -----

    protected void evt_${currentEvent.name}(StateType currState, StateType
nextState, //
        EventType event, Map<String, Object> ctx, Map<String, Object> par) {
        // Empty
    }

[/#list]
[#list state as currentState]
    [#list currentState.arc as currentArc]
    // -----
    protected void
${currentState.labelState.name}_${currentArc.eventArc.name}_${currentArc.target.s
tate.labelState.name}(Map<String, Object> ctx, Map<String, Object> par) {
        // Empty
    }

[/#list]
[/#list]
// -----

private void extStateChange(StateType currState, StateType nextState, //
    EventType event, Map<String, Object> ctx, Map<String, Object> par) {

    switch (currState) {
        [#list state as currentState]
        case ${currentState.labelState.name} :
            ext_${currentState.labelState.name}(currState, nextState, event, ctx,
par);
            break;
        [/#list]
        default :
            throwDefaultState(currState, event);
            break;
    }
}

```

```

}

// -----
private void entStateChange(StateType currState, StateType nextState, //
    EventType event, Map<String, Object> ctx, Map<String, Object> par) {

    switch (nextState) {
        [#list state as currentState]
        case ${currentState.labelState.name} :
            ent_${currentState.labelState.name}(currState, nextState, event, ctx,
par);
            break;
        [/#list]
        default :
            throwDefaultState(currState, event);
            break;
    }
}

// -----

private void evtStateChange(StateType currState, StateType nextState, //
    EventType event, Map<String, Object> ctx, Map<String, Object> par) {

    switch (event) {
        [#list eventSM as currentEvent]
        case ${currentEvent.name} :
            evt_${currentEvent.name}(currState, nextState, event, ctx, par);
            break;
        [/#list]
        default :
            throwDefaultState(currState, event);
            break;
    }
}

// -----

private void stateChangeMsg(StateType state, EventType event, //
    Map<String, Object> ctx, Map<String, Object> par) {

    System.err.println( //
        "Curr state is: " + this.state + //
        "; event is: " + event + "; next state is: " + state);
}

private void stateChangeSet(StateType state, EventType event, //
    Map<String, Object> ctx, Map<String, Object> par) {

    this.state = state;
}

```



```
}

// -----
protected void throwIllegalState(StateType state, EventType event) {
    System.err.println( //
        "IllegalStateException: " + state + " for event " + event);

    throw new IllegalStateException( //
        /**/state + " for event " + event);
}

protected void throwDefaultState(StateType state, EventType event) {
    String msg = "default : this is for sure a bug in the state machine for state
" + //
        state + " and event " + event;

    System.err.println(msg);

    throw new RuntimeException(msg);
}
}
```

Figura A.2. Plantilla de *FreeMarker*.