Universidad de las Ciencias Informáticas

Facultad 1

Herramienta para la generación de código de applets javacard a partir de diagramas de estado.

Trabajo de Diploma para optar por el título de Ingeniero en Ciencias Informáticas

Autores: Marisela Campos Donal.

Yoel Amed Rendón Zurbano.

Tutores: Ing. Dannier Sierra Obregón.

Ing. Vismar Fernández Santana.

Ciudad de La Habana

Junio, 2013

DECLARACIÓN DE AUTORÍA

Declaramos ser autores del presente trabajo y reconocemos a la Universidad de las Ciencias Informáticas los derechos patrimoniales del mismo, con carácter exclusivo. Para que así conste firmamos el presente a los ____ días del mes de _____ del año _____.

Marisela Campos Donal

_____

Firma del Autor

Yoel Amed Rendón Zurbano

_____

Firma del Autor

Ing. Dannier Sierra Obregón

_____

Firma del Tutor

Ing. Vismar Fernández Santana

_____

Firma del Tutor

Marisela

Agradezco a mis padres que siempre me han apoyado y ayudado en todo y me han servido de ejemplo y guía. A mi novio que siempre me apoyó en todo, mi compañero de la vida y de tesis y que siempre estuvo presente en los momento más difíciles. A mi familia y la de mi novio porque siempre me apoyaron de una forma u otra. A mis compañeras de cuarto, que resultaron ser buenas compañeras demostrando en más de una ocasión que podía contar con ellas. A mis tutores que me ayudaron a comprender y desarrollar este trabajo lo cual no fue una tarea fácil. A las personas que conocí en el laboratorio de tarjetas inteligentes, sobre todo a los profesores Ander y Susana que aunque no son mis tutores, fueron de mucha ayuda. A mis amigas por darme su ayuda cuando lo necesité. En general a todas las personas presentes en mi vida principalmente en la UCI.

Yoel Amed

Agradezco a mi mamá y a mi tía por darme apoyo durante estos 5 años de estudios y ser ejemplo y guía para alcanzar mis metas. A mi novia que siempre me ayudó en los momentos más difíciles. A mi familia y a la familia de mi novia en sentido general porque siempre me apoyaron de una forma u otra. Al tutor Dannier y a los profesores del Departamento que me ayudaron a comprender y desarrollar este trabajo lo cual no fue una tarea fácil. A mis amigos por darme apoyo siempre.

Marisela

A mi madre por ser mi amiga, mi protectora, mi gran amor de toda la vida. A mi padre por ser mi defensor, guía, amigo y sostén durante todo este tiempo. A ellos por apoyarme sin dudar nunca y por confiar en mí. A Dios por darme tan gran tesoro "mis padres".

A mi novio por ser mi mejor amigo, mi compañero de la vida y de tesis, por cuidarme, amarme, apoyarme y respetarme siempre. A mis amigas Rita y Lesyanis por su apoyo, comprensión y confianza en todo momento. A los familiares de mi novio por apoyarnos y aconsejarnos siempre.

Yoel Amed

A mi madre que me ha apoyado siempre, a mi tía Lidia por impulsarme y darme ánimos cada día, a mi prima Grisel y a mi papá donde quiera que estén, a mi familia en general. A mi novia por ser mi amiga, mi compañera de tesis, por su comprensión, apoyo y su amor.

A mis familiares y a los de mi novia, en especial a sus padres por ser como si fueran míos. A mis amigos por estar siempre cerca a pesar de la distancia.

Resumen.

La realización de un nuevo software requiere que las tareas sean completadas de forma correcta y eficiente. Las herramientas CASE (Computer Aided Software Engineering, Ingeniería de Software Asistida por Ordenador) son desarrolladas con el objetivo de automatizar los procesos del ciclo de vida de desarrollo de software. La razón para la creación de dichas herramientas fue el incremento en la velocidad de desarrollo de los sistemas. Estas herramientas proveen beneficios en el desarrollo de software como son, verificar el uso de todos los elementos en el sistema diseñado, automatizar el dibujo de diagramas, ayudar en la creación de relaciones en la base de datos y generar estructuras de código; funcionalidad muy codiciada en la actualidad, la cual se ha convertido en una necesidad para la mayoría de los programadores durante el desarrollo de software.

Las herramientas generadoras de código son de suma importancia para disminuir el tiempo de desarrollo y las probabilidades de cometer errores. El funcionamiento de las mismas se basa en la obtención de un modelo, el cual se transforma y convierte en código fuente.

La necesidad de minimizar tiempo y errores de programación en el desarrollo de aplicaciones javacard es el motivo de la realización del estudio que se presenta, para ello se hace una investigación acerca de las herramientas CASE para obtener las ventajas y desventajas de las mismas, con el objetivo de desarrollar una herramienta capaz de generar código javacard a partir de diagramas de estados.

Palabras claves: código fuente, código javacard, diagrama de estados, herramientas CASE, herramienta generadora de código.

Índice

Índice de figuras

Índice de tablas

Introducción

El uso de la informática en la actualidad ha crecido considerablemente. Las empresas cada día se interesan más por mantenerse actualizadas tecnológicamente e ir al ritmo del desarrollo científico tecnológico para mantenerse en competencia.

La utilización de software en la automatización de los procesos empresariales es clave para la eficiencia de las empresas, factor que ha permitido un vertiginoso crecimiento en la demanda de producción de aplicaciones informáticas en el mercado. El ahorro de tiempo por parte de las empresas y la calidad requerida por el usuario son puntos importantes para el avance en materias de producción, así como el mantenimiento en un mercado cada vez más exigente y competitivo.

La relación tiempo-producción muchas veces sufre un desbalance debido a los errores que se puedan cometer y el gran volumen de código que deben realizar los programadores en el desarrollo de software.

Escoger la herramienta adecuada para el desarrollo de software es clave para minimizar la cantidad de errores que se puedan cometer, y garantizar un menor tiempo de desarrollo. La capacidad de generar código es una de las funcionalidades más codiciadas en la actualidad, pues facilita el trabajo en cualquiera de las fases de un proyecto. Ejemplos de herramientas con estas características se tiene a Rational Rose, Visual Paradigm, Enterprise Architect entre otras, las cuales tienen la capacidad de generar código en varios lenguajes, pero ninguna de ellas genera código javacard.

La construcción manual de código javacard requiere de mucho tiempo y se convierte en una tarea tediosa por lo repetitiva que puede ser al inicio del ciclo de desarrollo. La argumentación anterior constituye, sin dudas, un freno al proceso de desarrollo de applets javacard, trayendo como consecuencia la pérdida de tiempo y esfuerzo por parte de los desarrolladores.

Por tanto, surge la necesidad de dar solución a la situación anteriormente expuesta, lo cual conlleva al **problema investigativo**: ¿Cómo facilitar el proceso de desarrollo de aplicaciones javacard, reduciendo el tiempo empleado y los errores cometidos por los programadores?

La investigación enmarca su **objeto de estudio** en el proceso de generación automática de código.

Se define como **objetivo general:** Desarrollar una herramienta que permita generar arquetipos[1] de aplicaciones javacard usando diagramas de estado.

Los **objetivos específicos** definidos son:

[1] Un arquetipo en el presente contexto hace referencia a una plantilla de código javacard que puede ser editado.

- ✓ Caracterizar las herramientas generadoras de código mediante diagramas de estado.
- ✓ Diseñar la herramienta para la generación de arquetipos de aplicaciones javacard a partir de diagramas de estado.
- ✓ Implementar la herramienta para la generación de arquetipos de aplicaciones javacard a partir de diagramas de estado.
- ✓ Probar las funcionalidades definidas para la herramienta de generación de arquetipos de aplicaciones javacard a partir de diagramas de estado.

Como **tareas de la investigación** se proponen las siguientes:

- ✓ Análisis de las herramientas generadoras de códigos existentes.
- ✓ Análisis de la sintaxis del lenguaje javacard a tener en cuenta para la generación de código.
- ✓ Descripción de las tecnologías, metodologías y estándares más adecuados para el desarrollo de la solución.
- ✓ Definición de plantillas de diagramas de estado para los principales estándares de comunicación y gestión de información de tarjetas.
- ✓ Diseño del prototipo de interfaz de usuario.
- ✓ Validación con especialistas del Departamento del prototipo de interfaz diseñado.
- ✓ Definición y refinamiento de los requisitos con la participación de especialistas del Departamento.
- ✓ Implementación de los requerimientos definidos para la solución haciendo uso de patrones de diseño.
- ✓ Sincronización del código generado.
- ✓ Realización de pruebas de unidad a la herramienta desarrollada.
- ✓ Realización de pruebas de aceptación para la herramienta con especialistas del Departamento y el Centro.

Entre los **métodos científicos teóricos** se utilizaron:

- ➢ Histórico – Lógico.

Mediante este método científico, se realizó un estudio del estado del arte, sobre las principales herramientas generadoras de código, principalmente las realizadas en la universidad. Permitió analizar la evolución de estas herramientas en materia de arquitectura y diseño, y las tendencias actuales.

- ➢ Analítico – Sintético.

El uso del método científico analítico – sintético permitió realizar un estudio por separado de cada una de las herramientas generadoras de código que más se utilizan en la actualidad, se definió qué particularidades presentaban en común y se estableció una series de parámetros, atendiendo principalmente a las características relacionadas con sus objetivos fundamentales, para establecer una comparación entre ellas y tomar los resultados arrojados por dicha comparación, como datos de gran interés para la actual investigación.

- Inducción – Deducción.

  Mediante la aplicación del mismo se desarrolló un estudio con los principales generadores código, atendiendo a sus características propias, y basado en ello se definieron las características o cualidades que debe tener o cumplir la herramienta que se propone en el trabajo.

  Entre los **métodos empíricos** se utilizaron:

- Observación.

  Mediante el método científico de la observación, se prestó atención a la situación actual existente en el departamento de Tarjetas Inteligentes en cuanto a las dificultades a la hora de generar código de applets javacard, así como la necesidad de crear una herramienta con este objetivo.

La realización de este trabajo facilitará el proceso de desarrollo de software principalmente en el departamento de Tarjetas Inteligentes permitiendo desarrollar aplicaciones de mayor calidad en el tiempo establecido para ello, además de permitir visualizar de manera más clara cada uno de los diferentes eventos y estados por los cuales transita una aplicación javacard durante su ciclo de vida.

CAPÍTULO 1: Fundamentación teórica.

1.1. Introducción.

En este capítulo se sustenta el problema científico y el propósito de este trabajo mediante la fundamentación teórica, formada por un grupo de referencias seleccionadas, análisis y valoraciones de los temas tratados. Para ello, se presentan los aspectos fundamentales relacionados con los generadores de código; características y conceptos asociados a la generación de código para la propuesta de una solución. Se aborda acerca del proceso actual y tendencias de sistemas generadores de código atendiendo principalmente a las tendencias históricas de los mismos. Se hace una valoración de programas existentes que permiten generar código. Se especifican y argumentan las principales técnicas y herramientas usadas para la construcción de dicho software, además de la metodología a usar para la realización de la aplicación y los principios a tener en cuenta. También se hace un análisis de la idoneidad del software en el departamento de Tarjetas Inteligentes y fuera del mismo.

1.2. Generación de código.

La generación de código es usada para construir programas de una manera automática evitando su escritura a mano, constituyendo un ahorro de tiempo en el desarrollo de proyectos y aplicaciones. Actualmente, es de suma importancia, pues el ahorro de tiempo, la eficiencia en la programación y la estandarización de código son pilares fundamentales para la construcción de un proyecto. ((HERRINGTON, 2006))

Las herramientas generadoras de código son hechas para programadores ayudándoles a facilitar su trabajo ya que permiten obtener, en un menor tiempo, el mismo código que un programador obtendría de forma manual, constituyendo un paso de avance en el mejoramiento de sus condiciones de trabajo, y permitiéndoles disminuir la posibilidad de cometer errores. El funcionamiento de estas herramientas se basa en la obtención de un modelo y la transformación del mismo en código fuente para el desarrollo de una aplicación. Los modelos pueden ser un diseño de base de datos, un diagrama de clases, un diagrama de estados, un modelo creado con la misma herramienta y otros.

1.3. Características de los generadores de código.

Los generadores de código se caracterizan por su lenguaje generado los cuales pueden ser un lenguaje estándar o un lenguaje propietario, por la portabilidad del código generado que es la capacidad para poder ejecutarlo en diferentes plataformas físicas y/o lógicas, por la generación del esqueleto del programa o del programa completo; si únicamente genera el esqueleto será necesario completar el resto mediante

programación, por la posibilidad de modificación del código generado y por la generación del código asociado a las plantillas e informes de la aplicación mediante el cual se obtendrá la interfaz de usuario de la aplicación.

1.4. Tipos de generación de código.

Los tipos de generación existentes son:

Templating: Se genera un esbozo de código fuente no funcional para ser editado, con el que se evita tener que escribir la parte más repetitiva del código (generalmente poco compleja). (Cepeda.v, 2007)

Parcial: Se genera código fuente que implementa parcialmente la funcionalidad requerida, pero que el programador usará como base para modificar, integrar y/o adaptar a sus necesidades. (Cepeda.v, 2007)

Total: Se genera código fuente funcionalmente completo pero que no va a ser modificado por el programador, sino que si es necesario se vuelve a regenerar. Por lo general tampoco suele ser un código excesivamente complejo. (Cepeda.v, 2007)

1.5. Diagrama de estados.

Un diagrama de estados muestra la secuencia de estados por los que pasa un caso de uso o un objeto a lo largo de su vida, indicando qué eventos hacen que pase de un estado a otro y cuáles son las respuestas y acciones que genera. (Xavier Ferré Grau)

Los diagramas de estados se representan en forma de grafo donde los nodos son los estados y los arcos son las transiciones, las cuales son disparadas por eventos.

Para la representación de los estados se utiliza una caja redondeada con el nombre de dicho estado en su interior y las transiciones son flechas que van desde un estado a otro. Los estados también pueden contener operaciones de entrada o salida del mismo o una acción interna.

Un diagrama de estados puede representar ciclos continuos o bien una vida finita, en la que hay un estado inicial de creación y un estado final de destrucción (del caso de uso o del objeto). El estado inicial se muestra como un círculo sólido y el estado final como un círculo sólido rodeado por una circunferencia. En realidad, los estados inicial y final son pseudoestados, pues un objeto no puede estar en esos estados, pero sirven para saber cuáles son las transiciones iniciales y finales. (Xavier Ferré Grau)

1.6. Javacard.

Es una tecnología que permite ejecutar aplicaciones Java en tarjetas inteligentes y similares dispositivos empotrados[2]. Esta tecnología ofrece la posibilidad y capacidad al usuario de implementar soluciones aplicadas en distintas esferas de la sociedad y la economía, con elevados niveles de seguridad, ejemplos claros los podemos encontrar en las aplicaciones que permiten la firma digital, la autenticación de usuarios, tarjetas de monedero electrónico y además en las tarjetas del módulo de identificación del suscriptor (SIM por sus siglas en inglés) utilizadas en la telefonía celular. Una tarjeta inteligente posee una estructura lógica además de su estructura física. El microprocesador es el encargado de realizar todas las operaciones, mientras el sistema operativo se encarga de traducir las acciones de las capas superiores en instrucciones que el microprocesador pueda entender. La máquina virtual de javacard permite la ejecución de aplicaciones javacard dentro de la tarjeta, mientras que las interfaces de programación de aplicación (APIs por sus siglas en inglés) de javacard poseen funcionalidades utilizadas por los applets. (Kenly Rodríguez Ruiz, 2012)

1.7. Applet javacard.

Los applets javacard son las aplicaciones que corren dentro de una tarjeta javacard. Dichas aplicaciones interactúan en todo momento con el JCRE (Javacard Runtime Environment) utilizando los servicios que este brinda, e implementan la interfaz definida en la clase abstracta javacard.framework.Applet.

La clase javacard.framework.Applet define cuatro métodos públicos que son utilizados por el JCRE para hacer funcionar las aplicaciones; el método install (), cuya implementación habitual es llamar al constructor de la clase (generalmente este constructor es privado), crear todos los objetos necesarios para ejecutar el applet y por último registrar el applet con el método registrer (), el método select() el cual es solicitado por el JCRE como resultado de la recepción a un select() APDU[3] (Application Protocol Data Unit), dicho APDU contiene el identificador de aplicación, el método process (APDU) dentro del cual el applet identifica el comando asociado al APDU y los parámetros, si los hay, y los procesa de acuerdo al protocolo que se haya definido para la interacción entre el applet y la aplicación terminal y el método deselect () para avisar al applet que se encuentra en ese momento seleccionado que va a dejar de estarlo. La tecnología javacard tiene como principal objetivo llevar los beneficios del desarrollo del software orientado a objetos al mundo de las tarjetas inteligentes. (Kenly Rodríguez Ruiz, 2012)

1.8. Máquina de estados.

Una máquina de estados es una estructura de programa que sirve para determinar el comportamiento de un objeto en base al estado en el que se encuentre. Para cada estado se tendrá un comportamiento.

[2] Un sistema embebido o empotrado es un sistema de computación diseñado para realizar una o algunas pocas funciones dedicadas frecuentemente en un sistema de computación en tiempo real.

3 Un APDU es la unidad de comunicación entre un lector de tarjetas inteligentes y una tarjeta inteligente.

Una máquina de estados se denomina máquina de estados finitos si el conjunto de estados de la máquina es finito, este es el único tipo de máquinas de estados que se puede modelar en una computadora en la actualidad; debido a esto se suelen utilizar los términos máquina de estados y máquina de estados finitos de forma intercambiable.

1.9. Proceso actual y tendencias.

1.9.1. Herramientas CASE (Computer Aided Software Engineering).

Existen varios métodos de desarrollo de software que impulsan la generación de código. Para tal fin se utilizan las herramientas CASE.

Las herramientas CASE son aplicaciones informáticas destinadas a aumentar la productividad en el desarrollo de software reduciendo el costo de las mismas en términos de tiempo y de dinero. Estas herramientas pueden ayudar en todos los aspectos del ciclo de vida de desarrollo del software.

1.9.2. Estado Actual.

En las últimas décadas se ha trabajado en el área de desarrollo de sistemas para encontrar técnicas que permitan incrementar la productividad y el control de calidad en cualquier proceso de elaboración de software.

La tecnología CASE reemplaza al papel y al lápiz por el ordenador para automatizar el proceso de desarrollo de software, contribuyendo así a elevar la productividad y la calidad en el desarrollo de los sistemas de información incrementando la productividad en el proceso de desarrollo del mismo.

1.10. Generación de código basado en modelos de objetos.

Esta generación está asociada con al menos un método de análisis y diseño, soportados por herramientas comerciales. La entrada básica para la obtención de código es el modelo de estructura de los objetos. Las herramientas ofrecen mecanismos para mantener los modelos y el código sincronizados. Los modelos han sido utilizados principalmente en desarrollo de sistemas de tiempo real y sistemas empotrados.

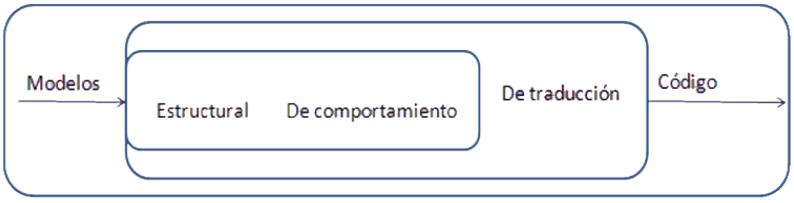

Figura 1. Tipos de generación basada en modelos.

En [Especificación de interfaz de usuario: De los requisitos a la generación automática] Moreno y Molina; se destacan tres enfoques para la generación de código:

- ✓ **El Enfoque Estructural:** Genera plantillas de código desde el modelo de estructura de objetos. Apropiado para metodologías en las cuales los modelos son elaborados mediante transición gradual entre análisis, diseño y código. Algunos productos son: Rational Rose, System Architect. Se ofrecen mecanismos para integrar código escrito manualmente junto con la estructura generada. Algunas herramientas protegen el código escrito manualmente para evitar reescribirlo en sucesivas generaciones. Se suele ofrecer mecanismos de ingeniería inversa para establecer modelos (la parte estructural). La generación de código estructural es incompleta pero ahorra esfuerzo de codificación manual y proporciona un marco de trabajo inicial consistente con los modelos. Las plantillas de traducción aportan un modesto grado de reutilización. La mayoría de las aplicaciones son adecuadas para esta aproximación.
- ✓ **Enfoque de Comportamiento:** Genera código completo usando máquinas de transición de estados y especificación de acciones. El beneficio adicional obtenido es la posibilidad de animar y validar el comportamiento del sistema (a partir de los modelos) pero antes de generar el código. La programación se reduce pero algunos aspectos deben incorporarse manualmente. Los traductores ofrecen poco control sobre la generación de código.
- ✓ **Traducción:** Usa un modelo de arquitectura independiente de la aplicación para dar un control total sobre la traducción a código. Se desarrollan modelos de arquitectura: un conjunto de patrones de código llamados “archetypes” que establecen reglas de traducción. El modelo de arquitectura es independiente del modelo de la aplicación. Esto favorece la reutilización de ambos modelos. La construcción de un modelo de arquitectura es en sí un proyecto. Pueden ofrecerse librerías y mecanismos de composición y especialización de arquitecturas. Suelen proveerse arquitecturas genéricas que pueden modificarse. Al igual que en el enfoque de comportamiento, la generación es sólo en un sentido.

1.11. Requisitos para la generación de código.

Los modelos deben ser apropiados para representar el problema, con constructores de modelado necesarios para lograr la generación de código basada en modelos, debe generar código de calidad y debe utilizarse una metodología y herramientas para la aplicación efectiva de la generación de código basada en modelos.

1.12. Herramientas generadoras de código más utilizadas en la actualidad.

1.12.1. Generador de código Altova UModel 2013.

El generador de código Altova UModel 2013 genera código basado en los estándares Java 1.4, Java 5.0 y Java 6.0. Es compatible con entornos de desarrollo como Eclipse, Borland JBuilder y otros. Los perfiles

para C# son 1.2, C# 2.0, C# 3.0 y C# 4.0. Genera código para Microsoft Visual C# .NET, Borland C#Builder y otros.

UModel 2013 es compatible con tipos genéricos de Java, como plantillas UML, y ofrece funciones de finalización automática y color de sintaxis para plantillas y enlaces de plantilla. Presenta un entorno de trabajo agradable, sencillo a la vista y muy intuitivo a la hora de realizar los diferentes modelos. En este entorno se puede trabajar de dos formas de manera simultánea, la vista de árbol del proyecto o gráficamente en el modelo. Es una herramienta rápida y eficaz. El único inconveniente de UModel es que en su versión actual, no permite definir asociaciones entre estereotipos en un perfil.

1.12.2. Generador de código CodeSmith Studio.

CodeSmith Studio es un generador de código basado en una plantilla que permite generar códigos para cualquier lenguaje de texto. Esta herramienta le permite condicionalmente agregar o quitar el código del resultado, para un objeto, como el objeto TableSchema (incluido en SchemaExplorer) que proporciona acceso sobre una tabla de base de datos. Se puede utilizar los lenguajes C#, VB.NET o JScript.NET en los modelos.

1.12.3. Generador de código Visual Paradigm.

Visual Paradigm for UML (VP-UML) permite generar diagramas de varios tipos, incluyendo UML, modelado de procesos de negocios, mapas mentales y Object-Relational Mapping (ORM) etc. Los diagramas ORM permiten generar el código y la estructura de la base de datos (en Java, PHP5 C#, C++). Permite seleccionar código y convertirlo a un diagrama de clases (UML) y viceversa a partir de Java, .dll, .exe, Xml, C++, Corba IDL, PHP5, Hibernate, JDBC, Ada 9x. Esta herramienta soporta aplicaciones Web, varios idiomas, es fácil de instalar y actualizar, tiene compatibilidad entre ediciones. Tiene como ventajas la generación de documentación en formatos HTML y PDF, brinda la posibilidad de intercambio de información con aplicaciones como Rational Rose, permite la generación de código e ingeniería inversa, además de la generación de documentación. Como desventaja las imágenes y reportes generados, no son de muy buena calidad. Entre sus limitaciones está que el código generado hace uso de librerías propias, suprimiendo al desarrollador la posibilidad de escoger que librería usar para acceder a los datos. El código generado no es de fácil comprensión para los desarrolladores. Es una herramienta propietaria que tiene un costo elevado. (Visual Paradigm)

1.12.4. Generador de código Rational Rose.

Rational Rose es una herramienta visual para el análisis y diseño de aplicaciones, utiliza UML como lenguaje de modelado y abarca todas las etapas del desarrollo de software. Esta herramienta permite generar código a partir de modelos Ada, ANSI C++, C++, CORBA, Java/J2EE, Visual C++ y Visual Basic.

También ofrece un lenguaje de modelado común que agiliza la creación del software. Como todos los demás productos Rational Rose, proporciona un lenguaje común de modelado para el equipo y facilita la creación de software de calidad más rápidamente. (IBM Corporation,)

1.12.5. Generador de código Visual Studio Ultimate.

En Visual Studio Ultimate se puede generar código de los diagramas de clases UML mediante el comando de Generar código. De forma predeterminada, el comando genera un código C# para cada UML que se seleccione. El comando generar código es especialmente adecuado para generar código a partir de la selección de elementos del usuario y para generar un archivo para cada clase UML u otro elemento.

1.12.6. Generador de código Magic Draw.

Magic Draw es una herramienta de modelado con completas características UML, es una herramienta que se mantiene actualizada. Es desarrollada por No Magic, Incorporation e implementada totalmente en Java. Diseñada para analistas del negocio, analistas de software, programadores, ingenieros de software, y escritores de la documentación. Esta herramienta de desarrollo dinámico es versátil y facilita el análisis y diseño de los sistemas y de las bases de datos orientados objetos. También incorpora soporte de generación de código e ingeniería inversa para lenguajes como: Java, C++, C#.

1.12.7. Generador de código Enterprise Architect.

Enterprise Architect (EA) Professional es una herramienta CASE de Sparx Systems. Soporta ocho diagramas estándares del UML: diagrama de casos de uso, de clases, de secuencia, de colaboración, de actividad, de estados, de componentes, de despliegue y varios perfiles del UML. Si fuera necesario, el diagrama de objetos se puede crear usando los diagramas de colaboración.

Enterprise Architect tiene un mecanismo de perfil UML genérico para cargar y trabajar con diferentes perfiles UML. En Enterprise Architect, estos perfiles se especifican en archivos XML con un formato específico.

Permite ingeniería de código (directa e inversa) para C++, Visual Basic 6, Java, C#, Delphi y Bases de datos.

1.13. Herramientas generadoras de bases de datos más utilizadas en la actualidad.

1.13.1. Generador de código CASE Studio.

CASE Studio es una herramienta de modelado para bases de datos, incluye facilidades para la creación de diagramas de relación, modelado de datos y gestión de estructuras. Tiene soporte para trabajar con una amplia variedad de formatos de base de datos (Oracle, SQL, MySQL, PostgreSQL, Access) y permite además, aplicar procesos de ingeniería inversa, usar plantillas de diseño personalizables y crear detallados

informes en HTML y RTF.

1.13.2. Generador de código ERStudio.

Es una herramienta de diseño de base de datos. Brinda productividad en diseño, generación, y mantenimiento de aplicaciones. Permite visualizar la estructura, los elementos importantes, y optimizar el diseño de la base de datos.

Genera automáticamente las tablas y líneas de procedimientos guardados y disparadores para los principales tipos de base de datos.

1.14. Generadores de código realizados en la Universidad de las Ciencias Informáticas.

En la Universidad de las Ciencias Informáticas se han desarrollado varias herramientas para generar código como: Generador de código para aplicaciones en php aplicado al ERPFAR, herramienta de Generación de Código Mediante Sistema Experto, JASCOG 1.0: herramienta para la generación de código JavaScript para la librería ExtJS, Desarrollo de la versión 2.0 de la herramienta Doctrine Generator para la generación de ficheros de mapeo basado en Doctrine empleando la tecnología PHP, herramienta para la generación de código de aplicaciones WEB, herramienta generadora de ficheros de mapeo, para la persistencia de objetos en esquemas relacionales basada en Doctrine, herramienta generadora de ficheros de mapeo para la persistencia de esquemas de objetos relacionales con los frameworks NHibernate, Hibernate y Propel, entre otras.

1.15. Valoración de las herramientas existentes o aportes de la investigación.

Luego de un estudio de diversas herramientas de generación de código y análisis de las características de las mismas se concluye que ninguna de ellas genera código para el desarrollo de applets javacard; característica fundamental en la herramienta que se requiere, por lo que la investigación realizada permitirá darle solución al problema planteado con la implementación de una herramienta capaz de facilitar el trabajo en el desarrollo de aplicaciones javacard, mediante la construcción de diagramas de estados que luego se exportan en código y así generar un arquetipo javacard, minimizando los errores de programación y el tiempo de desarrollo.

1.16. Herramientas, lenguajes, metodología y tecnologías a utilizar.

1.16.1. Herramientas de modelado a utilizar.

Con el estudio de homólogos antes realizado se escoge para el modelado la herramienta Altova Umodel, fundamentalmente porque permite la generación de código Java. Incluye ingeniería inversa con capacidad para leer código fuente en Java y generar modelos UML claros y precisos para abarcar rápidamente su

arquitectura de software. Es compatible con varios entornos de desarrollo, es intuitiva y fácil de usar, de interfaz sencilla y agradable.

1.16.2. Metodologías de desarrollo.

En el desarrollo de software es importante la selección de la metodología adecuada para el éxito del producto. Para ello se destacan dos enfoques principales: las metodologías tradicionales y metodologías ágiles.

Las tradicionales ofrecen cierta resistencia a cambios que sean necesarios durante el ciclo de vida del proyecto. El proceso de desarrollo de software con el uso de estas metodologías es muy controlado, el volumen de documentación es elevado, plantillas, normas, etcétera. Se realiza un contrato prefijado y el cliente puede interactuar con el equipo de desarrollo del proyecto. Poseen una rigurosa definición de roles, actividades, artefactos, herramientas y notaciones para el modelado. Un ejemplo de esta metodología es Rational Unified Process (RUP) el cual es un proceso formal, cuyo objetivo fundamental es la producción de software con alto nivel de calidad. Satisfacer los requisitos de los usuarios finales es su premisa, cuyos requisitos deben cumplirse sin violar el cronograma ni el presupuesto. Esta metodología es un proceso iterativo e incremental, centrado en la arquitectura y guiado por casos de uso, utiliza UML como lenguaje de notación. Incluye artefactos y roles. (Ivar Jacobson, 2000)

Las metodologías ágiles permiten realizar cambios durante el ciclo de vida del proyecto haciendo el proceso menos controlado. En caso de existir un contrato, es bastante flexible. El cliente es parte del equipo de desarrollo del proyecto. Se hace menos énfasis en la arquitectura del software y existen pocos roles y artefactos. Ejemplo de esta metodología son Scrum y eXtreme Programming (XP).

Scrum es indicado para proyectos en entornos complejos, donde los requisitos sean cambiantes o poco definidos y la innovación, la competitividad, la flexibilidad y la productividad son fundamentales. El software se desarrolla mediante iteraciones conocidas como sprint, el resultado de estas iteraciones se le muestra al cliente. (Kenly Rodríguez Ruiz, 2012)

XP es la más sobresaliente dentro de las metodologías ágiles. Está diseñada para que el cliente reciba el software que necesita en el tiempo que lo necesita. Su principal objetivo es satisfacer las necesidades de los usuarios y es por ello que se considera un éxito. El trabajo es desarrollado en equipo, preocupándose en todo momento del aprendizaje de los desarrolladores y estableciendo un buen clima de trabajo. XP le confiere las facultades a los desarrolladores para que puedan responder con confianza a las necesidades cambiantes de los clientes. Es la metodología más apropiada para entornos volátiles. Este tipo de programación es la adecuada para los proyectos con requisitos imprecisos, muy cambiantes y con un riesgo técnico excesivo. (Joskowicz, 2008)

1.16.3. Fundamentación de la selección de XP como metodología a utilizar.

Se selecciona esta metodología debido a la necesidad de realizar el software en un corto período de tiempo, no es necesario el uso profundo de documentación durante todo el ciclo de vida del proyecto y además el software debe poseer una gran capacidad de respuesta a los cambios.

1.16.4. UML (Unified Modeling Language).

Es un lenguaje de modelado visual usado para visualizar, especificar, construir y documentar el sistema de software. Se utiliza en varias etapas del ciclo de vida de la realización del software. Permite ser utilizado en una herramienta generadora de código y simplifica el desarrollo de software en sus inicios. (Ivar Jacobson, 2000)

1.16.5. Lenguajes de programación orientada a objetos.

1.16.5.1. Características principales de Java.

El lenguaje multiplataforma por excelencia en la actualidad es Java. Es muy potente para crear aplicaciones de escritorio, ligero, multiplataforma, es código libre, además de ser uno de los mejores lenguajes de programación y que va en progreso. Es un lenguaje orientado a objetos, es decir un POO, es un lenguaje gratuito y solo se necesita un editor de programación también gratuito para poder trabajar. (1Am)

1.16.5.2. Características principales de C#.

C# elimina muchos elementos añadidos por otros lenguajes y que facilitan su uso y compresión, como por ejemplo operadores diferentes al operador de acceso a métodos. Incluye restricciones para garantizar su seguridad, no permitiendo el uso de punteros. Sin embargo, y a diferencia de Java, existen modificadores para saltarse esta restricción, pudiendo manipular objetos a través de punteros. Esta característica puede resultar de utilidad en situaciones en las que se necesite gran velocidad de procesamiento. Para facilitar la migración de programadores, C# no sólo mantiene una sintaxis muy similar a C, C++ o Java, tiene la posibilidad de acceder a código nativo escrito como funciones sueltas no orientadas a objetos tales como dll. (1Am)

1.16.6. Lenguaje de programación orientado a objetos a utilizar.

Se utiliza el lenguaje Java por ser un lenguaje simple y robusto, es altamente fiable, es indiferente a la arquitectura, compatible con sistemas operativos como Windows 95, Unix a Windows NT y Mac, versátil, portable, especifica tamaños básicos, es un lenguaje dinámico y permite la creación de applets.

1.16.7. Entornos de desarrollo.

Entorno de desarrollo integrado (IDE: acrónimo que significa Integrated Development Environment), es un programa informático compuesto por un conjunto de herramientas de programación, en su ambiente de trabajo pueden utilizarse uno o varios lenguajes, por lo que son empaquetados en una única aplicación. Los IDE pueden ser aplicaciones por sí solas o pueden ser parte de aplicaciones existentes. (Universidad de la Republica de Uruguay, 2009)

Existen varias alternativas para ser utilizadas como IDE durante el desarrollo de proyectos basados en el lenguaje Java, pero el número se reduce si se busca software gratuito y preferentemente de código abierto:

- Eclipse es desarrollado por la Fundación Eclipse, una organización independiente que fomenta una comunidad de código abierto y un conjunto de productos complementarios, capacidades y servicios. Es un entorno de desarrollo integrado de código abierto y multiplataforma. Mayoritariamente se utiliza para desarrollar lo que se conoce como "Aplicaciones de Cliente Enriquecido", opuesto a las aplicaciones "Cliente-liviano" basadas en navegadores, es una plataforma de programación, desarrollo y compilación de elementos variados como sitios web, programas en C++ o aplicaciones Java. Además contiene una atractiva interfaz que lo hace fácil y agradable de usar. (Eclipse)
- NetBeans es un proyecto de código abierto fundado por Sun Microsystems especialmente diseñado para el desarrollo de aplicaciones en Java, pero acepta otros lenguajes de programación. Consta de una gran base de usuarios y una comunidad en constante crecimiento, lo que le ha permitido, al igual que muchos otros sistemas libres, el progreso paulatino de sus prestaciones y la eliminación de Bugs que pudiesen existir. NetBeans permite el desarrollo de aplicaciones javacard. Posee las siguientes ventajas: permite que las aplicaciones se desarrollen a partir de un conjunto de módulos o componentes de software. Un módulo contiene clases de Java escritas para interactuar con las Apis de NetBeans y un archivo especial que lo identifica como módulo. NetBeans IDE es fácil de instalar y de uso instantáneo y se ejecuta en varias plataformas incluyendo Windows XP, Linux, Mac OS y Solaris. Además del soporte completo para todas las plataformas. (SUN)

1.16.8. Entorno de desarrollo a utilizar.

Se escoge la herramienta NetBeans por ser gratuito, de código abierto, además permite trabajar con un gran número de librerías como FSM y Visual Library, las cuales son necesarias en el desarrollo de la aplicación. Tiene la posibilidad de utilizar otras versiones de compiladores y depuradores. Además se tiene un conocimiento previo de su uso.

1.16.9. Librerías a utilizar.

1.16.9.1. FSM.

Fue desarrollada por la compañía Continuent como parte del Proyecto Tungsten Replicator. Esta librería modela y describe el comportamiento de programas que reciben información de fuentes externas como un set de entradas, estados, transiciones y acciones. Fue diseñada para dar una simplicidad máxima y al mismo tiempo proveer todas las características para un proceso más rápido en memoria. Permite la definición de eventos, estados, transiciones y acciones, el procesamiento sincronizado seguro de transiciones, los protocolos de control de errores incluyendo reducción de precios de transiciones.

Posee clases como:

- **State:** Clase para crear una instancia física de un estado. Establece los estados y subestados.
- **Transition:** Define una transición entre dos condiciones con un cerrojo acompañante que determina cuando la transición puede ser aplicada.
- **TransitionStateMap:** Establece para cada estado la transición que tiene.
- **FiniteStateException:** Denota algo único en su género generado por un error en autómata finito yendo en procesión, como una transición faltante.
- **StateMachine:** Modela una máquina de estado finita. Está compuestas por:
  - State: El set de estados legales del sistema.
  - Transition: Los cambios legales entre las condiciones.
  - Events: Un set de mensajes que pueden inducir qué condición se altera.
  - Guards: Un set de condiciones que determinan las condiciones bajo las cuales un acontecimiento causa una transición particular para ser tomado.

1.16.9.2. Visual Library 2.0.

Provee visualización multiuso. Facilita construir y modificar un árbol de elementos visuales que son llamados widgets. La raíz del árbol es representada por una clase Scene que sujeta todos los datos visuales de la escena. Posee clases como:

- **Widget**: elemento visual primitivo el cual contiene información acerca de su posición, límite, posición /límite preferido, límite preferido tamaños mínimos o máximos, trazado, borde, primer plano, antecedentes, carácter de imprenta, cursor, herramientas, contexto asequible.
- **LabelWidget:** Representa un texto de la línea.

- **ConnectionWidget:** Representa una conexión /camino entre el punto fuente y el destino. La fuente y el destino están definidos por Anchor. El camino está definido por puntos de control que están resueltos por assignedRouter.
- **WidgetAction:** Son elementos visuales que están en la escena y se dibujan a sí mismos. Cada cual realiza una acción en específico.
- **LayerWidget:** Es usado para acomodar elementos en una capa sobre otros que podrían servir para diversas optimizaciones.

1.17. Valoración de idoneidad a partir de los estudios realizados.

Dado que no existe una herramienta generadora de código javacard, la propuesta de desarrollo de dicha herramienta da solución a este problema permitiendo el mantenimiento y calidad de las aplicaciones que se desarrollen, además de reducir el tiempo y costo del desarrollo, mejorar la planificación del proyecto, la generación de código y la portabilidad. Además del modelado visual de los diferentes estados por lo que transita una aplicación javacard permitiendo que el applet sea más robusto.

1.18. Conclusiones.

En este capítulo se realizó un estudio de las principales herramientas generadoras de código lo cual permitió obtener las características, ventajas y desventajas y tomar lo positivo para la herramienta que se desea implementar.

Debido a las nuevas necesidades surgidas se llegó a la conclusión de:

- Se trabajará con el lenguaje de programación Java debido a la versatilidad, eficacia, portabilidad de plataformas y seguridad de esta tecnología la cual se convierte en la ideal. Este lenguaje ha sido probado, ajustado, ampliado por toda una comunidad. Permite escribir aplicaciones potentes y eficaces, el uso del lenguaje es gratuito y con librerías de código abierto y se puede contar con un IDE gratuito para utilizar durante el desarrollo.
- Se plantea el diseño e implementación de una herramienta, la cual abarque las principales ventajas de las existentes y cubra la mayor parte de las necesidades de los desarrolladores de software de aplicaciones javacard, teniendo como punto positivo la agilización del desarrollo de aplicaciones.
- Luego de haber realizado un estudio detallado sobre algunas metodologías de desarrollo y teniendo en cuenta las características del proyecto y el tamaño del equipo de desarrollo se decide el uso de la metodología de desarrollo ágil XP (eXtreme Programming) debido a que permite la retroalimentación concreta y frecuente del equipo de desarrollo, el cliente y los usuarios finales, utiliza como una de sus prácticas fundamentales la programación en pareja lo que facilita que el código sea revisado

constantemente evitando una mayor cantidad de errores y dando como resultado un código eficaz y de alta calidad.

- Para el modelado de las clases se seleccionó la herramienta Altova Umodel debido a sus diversas funcionalidades principalmente la de generar código Java, además de ser una herramienta gratuita y ser la usada en el departamento de Tarjetas Inteligentes.
- El tipo de generación de código a utilizar es el templating para facilitar la modificación del arquetipo generado por el programador.

CAPÍTULO 2. Características del sistema.

2.1. Introducción.

En el presente capítulo se hace un análisis del funcionamiento del negocio, haciendo una caracterización del mismo, así como su representación gráfica. Se plantean los conceptos relacionados con la generación de código y se enumera los requisitos funcionales y no funcionales que debe tener la aplicación; dando así un acercamiento al sistema.

2.2. Objeto de automatización.

Como se presenta desde el diseño teórico de la investigación, para el desarrollo de aplicaciones javacard con más eficiencia es necesaria la generación de código al inicio del ciclo de desarrollo. El presente proyecto se propone lograr resolver esta problemática mediante la generación de código que resulte de un modelo de entrada UML que contenga la información primaria para que la herramienta logre su propósito; obtener un código lo más cercano al que hubiera hecho un programador.

2.3. Descripción general de la propuesta de sistema.

2.3.1. Funcionamiento del generador.

El sistema se encargará de la generación de un arquetipo de un applet javacard mediante diagramas de estado. Funcionará de manera que le permita al programador generar el código del applet que desea desarrollar.

Para la realización de la aplicación se utiliza la librería FSM, la cual facilita la realización de un diagrama de estados, así mismo la librería Visual Library facilita la modelación de este diagrama de estados de forma visual, además permite transformar un documento XML en un diagrama de estados y viceversa. De esta forma a partir de un diagrama de estados o un documento XML se obtiene la información para generar un arquetipo de un applet javacard. El generador también permite obtener un diagrama de estados mediante la información contenida en un applet javacard y luego generar el código correspondiente al mismo.

Las figuras 2 y 3 muestran el funcionamiento del generador de código en los procesos de generación de código e ingeniería inversa respectivamente.

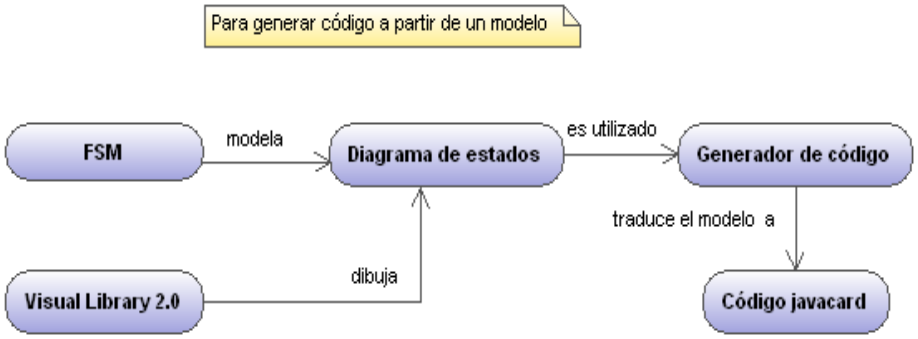

Figura 2. Funcionamiento del generador de código para ingeniería directa.

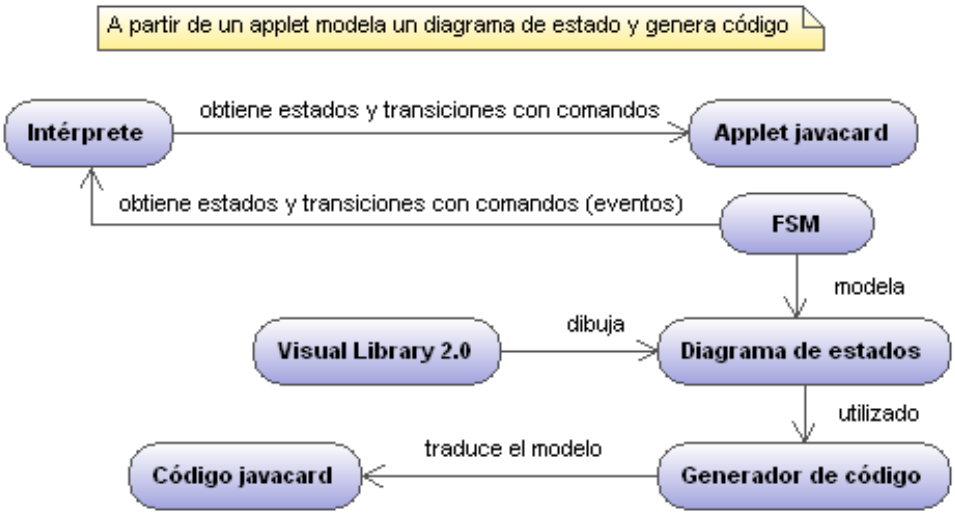

Figura 3. Funcionamiento del generador de código para ingeniería inversa.

2.3.2. XML de un diagrama de estados.

Para representar los componentes de un diagrama de estados se definen documentos XML que contienen sus propiedades y que luego serán utilizadas para representar el diagrama de forma visual.

El archivo XML de un diagrama de estados guardado por la herramienta generadora de código es de la forma siguiente:

```
- <Scene edgeIDcounter="4" nodeIDcounter="5" version="1">
    <Node id="VolatileState2" image="volatile" x="404" y="434"/>
    <Node id="VolatileState1" image="volatile" x="625" y="314"/>
    <Node id="PersistentState1" image="persistent" x="376" y="193"/>
    <Node id="." image="start" x="203" y="333"/>
    <Node id="," image="end" x="439" y="603"/>
    <Edge id="edge0" label="transition1" source="." target="PersistentState1"/>
    <Edge id="transition44" label="transition4" source="VolatileState2" target=","/>
    <Edge id="transition33" label="transition3" source="PersistentState1" target="VolatileState2"/>
    <Edge id="transition32" label="transition3" source="VolatileState1" target="VolatileState2"/>
    <Edge id="transition21" label="transition2" source="PersistentState1" target="VolatileState1"/>
  </Scene>
```

Figura 4. Estructura de una archivo XML de un diagrama de estados.

A continuación se describe cada una de las etiquetas.

- <Scene> escena

  Representa el inicio y fin del XML, es una escena que representa un único diagrama de estados que es guardado, posee la cantidad de estados (nodeIDcounter) y transiciones (edgeIDcounter) de dicho diagrama.

- <Node id="VolatileState2" image="volatile" x="404" y="434"/> nodo

  Representa un estado y la información correspondiente al mismo. El id es un nombre único del estado. La imagen (image) es el tipo de estado y por tanto la imagen que lo representa. Además tiene una posición en la escena que es el punto (x, y) los cuales son los valores de x, y respectivamente.

- <Edge id="edge0" label="transition1" source="." target="PersistentState1"/> arista

  Representa a una transición y la instrucción que dispara la misma. El id es un identificador único para cada transición, el label es el nombre de la instrucción que se corresponde con la transición, la fuente (source) es el nombre del estado de salida de la transición y el destino (target) es el nombre del estado de entrada de la transición.

2.4. Fases de XP (eXtreme Programming) desarrolladas.

- Exploración: En esta fase, los clientes plantearon a grandes rasgos las historias de usuario que son de interés para la primera entrega del producto. Al mismo tiempo el equipo de desarrollo se familiarizó con las herramientas, tecnologías y prácticas que se utilizarán en el proyecto. Se prueba la tecnología y se exploran las posibilidades de la arquitectura del sistema construyendo un prototipo.

- Planificación de la Entrega (*Release*): En esta fase el cliente establece la prioridad de cada historia de usuario, y correspondientemente, se hace una estimación del esfuerzo necesario de cada una de ellas.

Se toman acuerdos sobre el contenido de la primera entrega y se determina un cronograma en conjunto con el cliente.

- Iteraciones: En esta fase se hacen varias iteraciones sobre el sistema antes de ser entregado. Los elementos que se toman en cuenta durante la elaboración del plan de la iteración son: historias de usuario no abordadas, velocidad del proyecto, pruebas de aceptación no superadas en la iteración anterior y tareas no terminadas en la iteración anterior. Todo el trabajo de la iteración es expresado en tareas de programación, cada una de ellas es asignada a un programador como responsable, pero llevadas a cabo por parejas de programadores.
- Producción: En la fase de producción fueron necesarias pruebas adicionales y revisiones de rendimiento antes de que el sistema fuera trasladado al entorno del cliente. Al mismo tiempo, se tomaron decisiones sobre la inclusión de nuevas características a la versión actual, debido a cambios durante esta fase. (Penadés)

2.5. Modelo de dominio.

Se identificaron un conjunto de conceptos que definen el proceso de generación de código, específicamente la generación de applets javacard mediante diagramas de estado, evidenciando el entorno en el cual está enmarcado el problema de la investigación, por lo que se determina la creación del modelo de dominio para un mejor entendimiento de la solución.

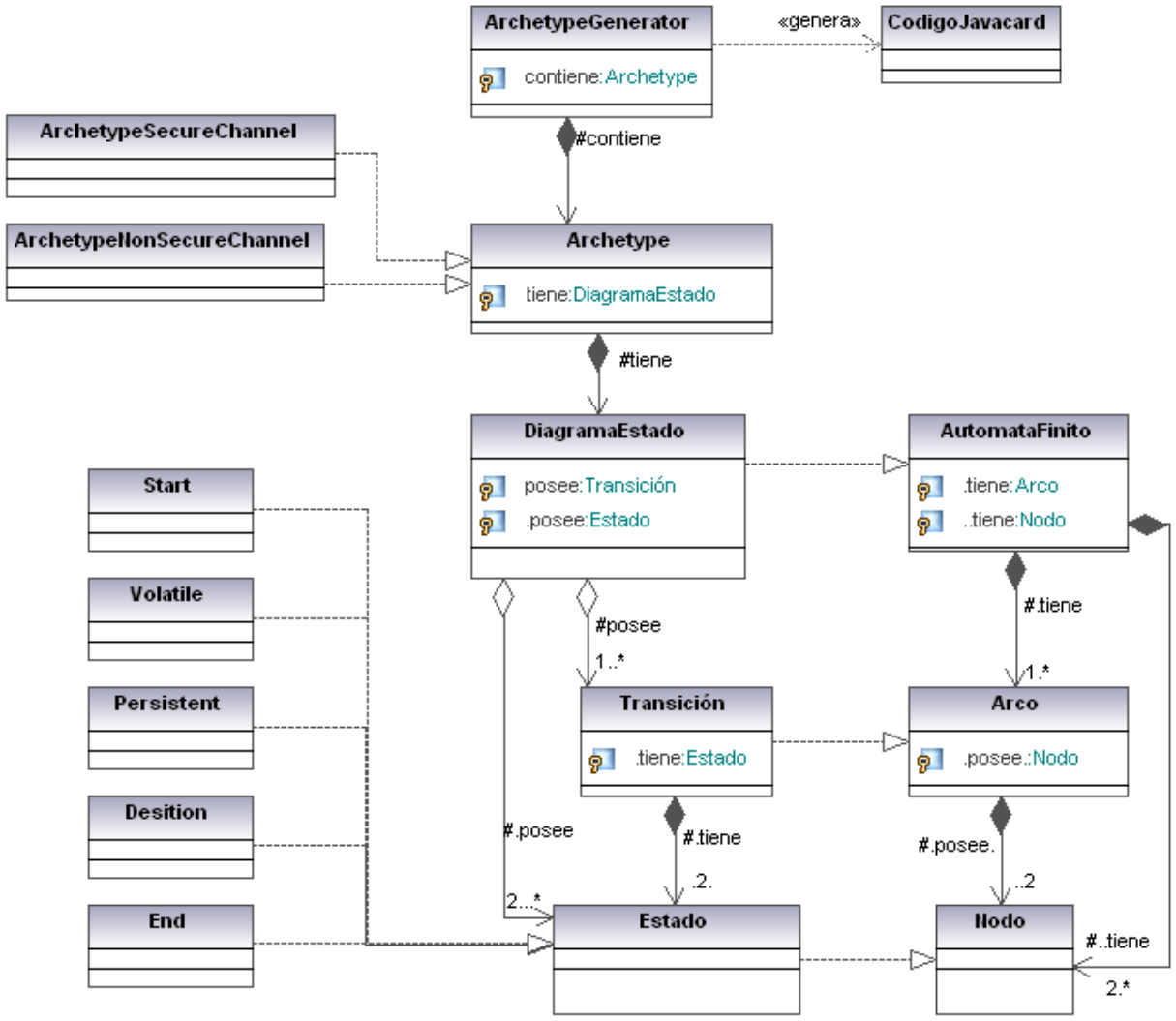

Figura 5. Modelo de dominio.

2.5.1. Glosario de términos del modelo de dominio.

- **Javacard**: permite que las tarjetas inteligentes y otros dispositivos con memoria muy limitada ejecuten pequeñas aplicaciones, llamadas applet, que utilizan la tecnología Java.
- **Estado persistente:** son los estados a los cuales regresa el applet en caso de que la tarjeta sea retirada del lector o el applet deje de estar seleccionado, regresando siempre al último estado persistente alcanzado.
- **Estado volátil:** es un estado donde la tarjeta se encuentra al recibir un comando y en caso de que la tarjeta sea retirada del lector o del applet, no permanecerá en este estado sino que regresará a un estado persistente.

- **Estado inicial:** primer estado; tiene un evento que indica el estado donde comienza el proceso de modelado.
- **Estado final:** último estado; indicado por un evento el cual da cierre al proceso de modelado.
- **Estado de decisión:** indica si se transita o no a un estado activo en caso de recibir una instrucción.
- **Transición:** acción de poder ir de un estado a otro mediante determinado comando recibido por la tarjeta inteligente.
- **Arco:** representación con una flecha de una transición.
- **Autómata finito:** es un diagrama de estados pero que con un comando y un estado origen solo permite una única transición a un estado destino.
- **Diagrama de estados:** diagrama mediante el cual se representa la vida de la tarjeta o del applet, a través de estados y transiciones.
- **Nodo:** representación gráfica de un estado; con una caja redondeada.
- **Generador de arquetipos:** clase que encargada de generar plantillas de applets javacard.
- **Arquetipo:** plantilla de un applet javacard.
- **Arquetipo de applet para canal seguro:** plantilla de un applet javacard con canal seguro.
- **Arquetipo de applet sin canal seguro:** plantilla de un applet javacard sin canal seguro.

2.6. Historias de usuario.

Las historias de usuario son utilizadas con el objetivo de especificar los requisitos del software. Son escritas por los clientes, atendiendo a las necesidades del sistema identificados por ellos mismos. Son descripciones breves y en lenguaje simple. A continuación se presentan algunas de ellas:

Historia de Usuario	
Número: HU_1	**Nombre de Historia de Usuario**: Crear nuevo diagrama de estados
Modificación de Historia de Usuario Número: Ninguna	
Usuario: Desarrollador	**Iteración asignada**: 1
Prioridad en negocio: Alta	**Puntos estimados:** 3
Riesgo en desarrollo: Medio	**Puntos reales:** 2
Descripción: Permite crear un diagrama de estados que modela el comportamiento de un applet javacard.	

Observaciones: La herramienta responderá si desea guardar un proyecto en caso de que: Se esté realizando algún proyecto o se haya cargado un diagrama antes realizado por la herramienta.

Tabla 1. Historia de usuario crear nuevo diagrama de estados.

Historia de Usuario	
Número: HU_2	**Nombre de Historia de Usuario**: Abrir diagrama de estados
Modificación de Historia de Usuario Número: Ninguna	
Usuario: Desarrollador	**Iteración asignada**: 1
Prioridad en negocio: Alta	**Puntos estimados:** 1
Riesgo en desarrollo: Medio	**Puntos reales:** 1
Descripción: Permite cargar de un archivo XML con la información de un diagrama de estados antes realizado en la herramienta.	
Observaciones: La herramienta permitirá seleccionar un proyecto realizado con anterioridad en la herramienta y mostrará el diagrama de estados antes guardado.	

Tabla 2. Historia de usuario abrir diagrama de estados.

El resto de las historias de usuario se encuentran desde el anexo 2 al anexo 10.

2.7. Requerimientos no funcionales.

No	Requerimientos no funcionales.
1.	**Portabilidad.** El uso del sistema será multiplataforma.
2.	**Apariencia o Interfaz interna.** La interfaz interna estará determinada por los desarrolladores, construyendo así una vista escalable de las clases o agrupaciones de clases que permitirán un mejor encapsulamiento de las funcionalidades y una mayor abstracción modular del sistema.

3.	**Software** Lenguaje de programación: Java. IDE: NetBeans 7.1 Para el Modelado de UML se utilizará: Altova Umodel 2010 Enterprise Edition. El sistema necesitará para funcionar la Máquina Virtual de Java 1.6 (JDK 1.6) para versiones de windows de 32 bits y 1.7 (JDK 1.7) para todas las versiones de Linux. El sistema podrá ser usado en todas las versiones de Windows a partir de la XP y en la mayoría de las versiones de Linux Ubuntu.
4.	**Requisitos de Hardware** Requerimientos mínimos de hardware. PC Intel Pentium 4 o superior. 512 MB de Memoria RAM o superior.
5.	**Rendimiento** La aplicación está concebida para ser utilizada por los desarrolladores para facilitar su trabajo, por lo que los tiempos de respuestas deben ser 2 a 5 segundos. La aplicación requiere de un buen rendimiento en máquinas de pocos recursos de hardware.

Tabla 3. Requisitos no funcionales.

2.8. Metáfora.

El término metáfora hace referencia a una historia compartida que describe cómo debería funcionar el sistema. El diseño con metáforas es el diseño de la solución más simple que pueda funcionar y ser implementado en un momento dado del proyecto. (Freire)

La metáfora definida para la solución es la siguiente:

Teniendo en cuenta que un applet se comporta como una máquina de estados finita, se puede representar su comportamiento con un diagrama de estados. Luego con los datos de los estados por los cuales transita la tarjeta inteligente y los comandos que disparan una transición de un estado a otro, se puede construir una plantilla de un applet donde luego se le insertará código de forma manual. De forma general la aplicación para generar código javacard puede ser utilizada por cualquier cliente que desee desarrollar una aplicación javacard.

2.9. Arquitectura en capas.

En [GS94] Garlan y Shaw definen el estilo en capas como una organización jerárquica tal que cada capa proporciona servicios a la capa inmediatamente superior y se sirve de las prestaciones que le brinda la inmediatamente inferior. En la práctica, las capas suelen ser entidades complejas, compuestas de varios paquetes o subsistemas. El uso de arquitecturas en capas, explícitas o implícitas, es frecuente.

Se definen en la parte de la presentación, las vistas con las que contará la aplicación y las clases que son parte de los elementos visuales de un diagrama de estados, dentro de la controladora estará una clase que manejará todo el flujo de la información necesaria para la generación de código y en persistencia de datos una clase que es la encargada de guardar los datos que el usuario introduzca y las entidades que serán las clases persistentes que simula el problema real para la generación y sincronización del código. La estructura que propone este patrón para la lógica del negocio del sistema se ve evidenciada en el diagrama que representa la estructura física de las clases, anteriormente descrita.

Figura 6. Arquitectura del sistema.

2.10. Plan de entregas.

Para elaborar el plan estimado de entrega se utilizaron las historias de usuario. Con cada historia de usuario previamente evaluada en tiempo de desarrollo ideal, el cliente las agrupó en orden de importancia. De esta forma se pudo trazar el plan de entregas en función de estos dos parámetros: tiempo de desarrollo ideal y grado de importancia para el cliente. Este artefacto fue elaborado con el objetivo de fijar que período de tiempo puede tardar la implementación de cada una de las historias de usuario, definiéndose las fechas en que serán liberadas las versiones funcionales del producto. En el anexo 11 se encuentra la tabla del plan de entregas.

2.11. Patrones.

Christopher Alexander [4] expresa que un patrón describe un problema que ocurre una y otra vez en nuestro entorno, para describir después el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser usada más de un millón de veces sin hacerlo ni siquiera dos veces de la misma forma.

Los patrones surgen a partir de la experiencia de seres humanos al tratar de lograr ciertos objetivos, capturan la experiencia existente y probada para promover buenas prácticas. Estos ayudan a los diseñadores a reutilizar con éxito los diseños con el objetivo de guardar la experiencia en diseños de programas orientados a objetos y hacer más fácil la reutilización de los diseños y arquitecturas, hacen a un sistema reutilizable y evitan alternativas que comprometen la reutilización.

Estos son una disciplina en la ingeniería del software que ha emergido pero que pueden ser aplicados en cualquier ámbito de la informática y las ciencias en general.

El objetivo de los patrones es crear un lenguaje común a una comunidad de desarrolladores para comunicar experiencia sobre los problemas y sus soluciones. Pueden referirse a distintos niveles de abstracción, desde un proceso de desarrollo hasta la utilización eficiente de un lenguaje de programación.

Según James Coplien[5], un buen patrón debería:

- ✓ Solucionar un problema.
- ✓ Ser un concepto probado.
- ✓ Describir participantes y relaciones entre ellos.
- ✓ Tener un componente humano alto: estética y utilidad.

2.11.1. Patrones de diseño.

Un patrón de diseño es una descripción de clases y objetos comunicándose entre sí, adaptada para resolver un problema de diseño general en un contexto particular. Identifican clases, instancias, roles, colaboraciones y la distribución de responsabilidades.

Los patrones del diseño se pueden clasificar según su propósito en:

- ✓ **Patrones de creación:** Los patrones de creación abstraen la forma en la que se crean los objetos, permitiendo a las clases crear de forma genérica dejando para más tarde la decisión de qué clases crear o cómo crearlas. Según donde se tome dicha decisión se puede clasificar a los patrones de creación en patrones de creación de clase (la decisión se toma en los constructores de las clases y usan la herencia para determinar la creación de las instancias) y patrones de creación de objeto (se modifica la clase desde el objeto). (Maribel Silva Muñoz, Diseño e Implementación de un sistema

[4] Christopher Alexander: Arquitecto, reconocido por sus diseños destacados de edificios en California, Japón y México.
[5] Escritor destacado, profesor e investigador en el campo de la Ciencia de la Computación.

informático integrado para la Gestión de Compras de Bienes y Contratación de Servicios en los Registros y las Notarías de la República Bolivariana de Venezuela.)

- ✓ **Patrones estructurales:** Tratan de conseguir qué cambios en los requisitos de la aplicación no ocasionan cambios en las relaciones entre los objetos. Lo fundamental son las relaciones de uso entre los objetos y estas están determinadas por las interfaces que soportan los objetos. Estudian cómo se relacionan los objetos en tiempo de ejecución. Sirven para diseñar las interconexiones entre los objetos. (Maribel Silva Muñoz, Diseño e Implementación de un sistema informático integrado para la Gestión de Compras de Bienes y Contratación de Servicios en los Registros y las Notarías de la República Bolivariana de Venezuela.)
- ✓ **Patrones de comportamiento:** Los patrones de comportamiento estudian las relaciones entre llamadas de los diferentes objetos, normalmente ligados con la dimensión temporal. (Maribel Silva Muñoz, Diseño e Implementación de un sistema informático integrado para la Gestión de Compras de Bienes y Contratación de Servicios en los Registros y las Notarías de la República Bolivariana de Venezuela.)
- ✓ **Patrones GRASP (General Responsibility Assignment Software Patterns):** Asignar correctamente las responsabilidades es muy importante en el diseño orientado a objetos. Los patrones GRASP describen los principios fundamentales de la asignación de responsabilidades a objetos, expresados en forma de patrones. Dentro de este grupo de patrones podemos encontrar los siguientes: experto, creador, bajo acoplamiento, alta cohesión, controlador, fabricación pura, variaciones protegidas y polimorfismo. (Maribel Silva Muñoz, Diseño e Implementación de un sistema informático integrado para la Gestión de Compras de Bienes y Contratación de Servicios en los Registros y las Notarías de la República Bolivariana de Venezuela.)

2.11.2. Patrones utilizados en la aplicación.

Los patrones del diseño utilizados en la aplicación son los patrones GRASP (General Responsibility Assignment Software Patterns):

- ✓ **Controlador:** Es el intermediario entre la interfaz y el algoritmo que la implementa. Se utiliza al crearse una clase controladora, la cual es la encargada del flujo de información de los datos entrados por el usuario como los estados y transiciones y la generación de código obtenido de dichos datos. Ejemplo:
  - En la solución propuesta se necesitaba gestionar el flujo de información correspondiente a un diagrama de estados diseñado en la interfaz por el desarrollador y así poder generar el código javacard correspondiente a dicho diagrama, para ello se creó una clase controladora “ArchetypeGenerator” la cual es la encargada de manejar el flujo de información

correspondiente al diagrama de estados diseñado y generar el código correspondiente al mismo.

En la siguiente figura se muestra el ejemplo anteriormente descrito.

```
public class ArquetypeGenerator {


    private ListaSE<StateJC> stateList;
    private ListaSE<TransitionJC> transitionlist;

    public ArquetypeGenerator() {

        transitionlist=new ListaSE<TransitionJC>();
        stateList=new ListaSE<StateJC>();
    }


    //generate an arquetype using a secure channel for the applet javacard.
    public String GenerateSecureChannel(ArchetypeAppletSecuredChannel arquetype){
         return  arquetype.GenerateSecureChannel();
    }
     //generate an arquetype using a non secure channel for the applet javacard.
    public String GenerateNoSecureChannel(ArchetypeAppletNonSecuredChannel arquetype){
        return arquetype.GenerateNoSecureChannelCode();
    }
}
```

Figura 7. Ejemplo de código de uso del patrón controlador.

- ✓ **Alta cohesión:** Cada una de las clases contiene cierta información la cual se relaciona con la misma.

  Ejemplo:

  - En la solución cada una de las clases que modelan el problema real; los componentes del diagrama de estados (estados, transiciones y eventos correspondientes), solo contienen la información que le corresponde a los mismos; los atributos y las acciones que la entidad realiza.

En la siguiente figura se muestra la clase StateJC la cual representa un estado de un diagrama de estados.

```
public class StateJC extends State{
    private  boolean create;
    private String newName;
    private int value;

    public int getValue() {...}

    public String getNewName() {...}

    public void setNewName(String newName) {...}

    public void setValue(int value) {...}

    public boolean isCreate() {...}

    public void setCreate(boolean create) {...}

    public StateTypeJC getTypeJc() {...}

    public void setTypeJc(StateTypeJC typeJc) {...}
    StateTypeJC typeJc;
    public StateJC( StateTypeJC typeJc, String name, StateType type, State parent) {
        super(name, type, parent);
        this.create = false;
        this.typeJc = typeJc;
        value=-1;
        newName="";
    }
}
```

Figura 8. Ejemplo de código de uso del patrón alta cohesión.

- ✓ **Creador:** Este patrón asigna a la clase B la responsabilidad de crear una instancia de clase A en uno de los siguientes casos:
  - B agrega los objetos A.
  - B contiene los objetos A.
  - B registra las instancias de los objetos A.
  - B utiliza específicamente los objetos A.
  - B tiene los datos de inicialización que serán transmitidos a la clase A cuando este objeto sea creado (así que B es un experto respecto a la creación de A).
  - B es un creador de los objetos A.

    Ejemplo:

    La clase VisualSceneSerializer es la encargada de crear y agregar los estados y transiciones y los utiliza para mostrarlos de forma visual al usuario.

En la siguiente figura se muestra como la clase VisualSceneSerializer es la encargada de crear los estados y transiciones del diagrama de estados.

```
public class VisualSceneSerializer extends GraphScene.StringGraph{

 private ListaSE<TransitionJC> transitions;
 private  ListaSE<StateJC> states;
 private NodeMenu nodeMenu;
 private HashMap<String, Anchor> anchors;
 private  EdgeMenu edgeMenu;
 private TreeJC tree;
 private TreeJC tree2;
 private int nodeValue;

public VisualSceneSerializer () {

 transitions=new ListaSE<TransitionJC>();
 states=new ListaSE<StateJC>();
 anchors = new HashMap<String, Anchor> ();
 nodeMenu=new NodeMenu(this);
 edgeMenu=new EdgeMenu(this);
 tree=new TreeJC();
 tree2=new TreeJC();
 nodeValue=2;

 }
```

Figura 9. Ejemplo de código de uso del patrón creador.

2.12. Descripción de las clases.

Clase ArchetypeGenerator.

Nombre	**ArchetypeGenerator**
Responsabilidades	Controla el flujo de información entre la parte lógica y la parte visual de la aplicación. Crea elementos del diagrama de estados.
Clases de las que depende	ListaSE State Transition State JC Transition JC

	StateMachine

Tabla 4. Clase ArchetypeGenerator.

Clase SceneSerializer.

Nombre	**SceneSerializer**
Responsabilidades	Crea la escena donde se visualizarán los elementos del visual.
Clases de las que depende	ListaSE Widget Layer Widget Label Widget Connection Widget File Image My Hover Provider

Tabla 5. Clase SceneSerializer.

Clase TransitionLoaded.

Nombre	TransitionLoaded
Responsabilidades	Permite obtener los datos de una transición en un applet javacard.
Clases de las que depende	StateLoaded InstructionLoaded

Tabla 6.Clase TransitionLoaded.

El resto de las descripciones de las clases se encuentran desde el anexo 13 al anexo 36.

2.13. Conclusiones.

En este capítulo se comenzó a desarrollar la propuesta de solución, obteniéndose a partir de las historias de usuario un listado con las funciones que debe tener el sistema. También se identificaron cada una de las clases que modelan la solución, se realizó un modelo de dominio para un mejor entendimiento de dicha solución, además de identificarse bajo qué condiciones la aplicación funcionará.

Gracias a ello se da inicio al desarrollo de la aplicación, tratando de que se cumplan todos los requerimientos y las funciones que han sido consideradas necesarias en el capítulo.

CAPITULO 3. Implementación y prueba del sistema.

3.1. Introducción.

En el presente capítulo se valida y verifica el cumplimiento de los requerimientos estipulados, mediante la aplicación de métodos de pruebas que garanticen la calidad del sistema. Los métodos a utilizar son las pruebas de caja blanca y las pruebas de caja negra. También se establecen los estándares de codificación a tener presentes en la implementación del sistema y se establece su distribución física.

Para verificar que la solución da respuesta al problema planteado al inicio de la investigación, se hace un muestreo con varios desarrolladores y se mide el tiempo para la generación de código utilizando la herramienta creada y la codificación de forma manual.

3.2. Diagrama de despliegue.

El sistema para su funcionamiento, solo necesita una PC cliente, debido a que los datos guardados se archivan en un documento XML, el cual será utilizado por el mismo sistema para generar las plantillas de código javacard.

3.3. Estándares de codificación a tener en cuenta en el código.

Un estándar de codificación son reglas que se siguen para la escritura del código fuente. De tal manera que otros programadores se les facilite entender el código (como identificar las variables, las funcionalidades o métodos, etc.). Para la aplicación se tuvo en cuenta el estándar camel para los nombres, también se tuvo en cuenta los estándares definidos para el lenguaje Java. Dichos estándares son:

- Son evitados los ficheros de más de 2000 líneas.
- Cada fichero fuente contiene una única clase o interface pública. Cuando algunas clases o interfaces privadas están asociadas a una clase pública se ponen en el mismo fichero que la clase pública.
- Los ficheros fuentes tienen la siguiente ordenación:
    1. Sentencias package e import.
    2. Declaraciones.
    3. Métodos.
- Se evitan las líneas de más de 80 caracteres.
- Cuando una expresión no entra en una línea, se rompe de acuerdo con los siguientes principios:
    - Después de una coma.
    - Antes de un operador.

  - Preferir roturas más a la derecha que el padre.
- Alinear la nueva línea con el comienzo de la expresión al mismo nivel de la línea anterior.
- En caso de que se aglutine el código el margen derecho, indentar justo 8 espacios en su lugar.
- Saltar de líneas por sentencias if sigue generalmente la regla de los 8 espacios.
- Se utiliza una declaración por línea.
- Se intenta inicializar las variables locales donde se declaran. La única razón para no inicializarla donde se declara es si el valor inicial depende de algunos cálculos que ocurren.
- Se ponen las declaraciones solo al principio de los bloques (un bloque es cualquier código encerrado por llaves "{" y "}").
- Cada línea debe contener como máximo una sentencia.
- La llave de apertura se pone al final de la línea que comienza la sentencia compuesta; la llave de cierre debe empezar una nueva línea y ser indentada al mismo nivel que el principio de la sentencia compuesta.
- Se usa dos líneas en blanco en las siguientes circunstancias:
  - ✓ Entre las secciones de un fichero fuente.
  - ✓ Entre las definiciones de clases e interfaces.
- Se usa una línea en blanco en las siguientes circunstancias:
  - ✓ Entre métodos.
  - ✓ Entre las variables locales de un método y su primera sentencia.
  - ✓ Entre las distintas secciones lógicas de un método para facilitar la lectura.
- Los nombres de las clases son sustantivos y cuando son compuestos la primera letra de cada palabra que lo forma es en mayúscula. Los nombres de las clases son simples y descriptivos. Se usan palabras completas evitando acrónimos y abreviaturas.
- Los métodos son verbos, cuando son compuestos la primera letra de todas las palabras que lo forma es en mayúscula.
- Excepto las constantes, todas las instancias y variables de clase o método empiezan con minúscula. Las palabras internas que lo forman si son compuestas empiezan con su primera letra en mayúsculas. Los nombres de variables no empiezan con los caracteres subguión "_" o signo del dólar "$", aunque ambos estén permitidos por el lenguaje.
- Los nombres de variables de un solo carácter se evitan.

3.4. Pruebas realizadas a la solución.

“El desarrollo de sistemas de software implica una serie de actividades de producción en las que las posibilidades de que aparezca el fallo humano son enormes. Los errores pueden empezar a darse desde el

primer momento del proceso, en el que los objetivos pueden estar especificados de forma errónea o imperfecta, así como en posteriores pasos de diseño y desarrollo. Debido a la imposibilidad humana de trabajar y comunicarse de forma perfecta, el desarrollo de software ha de ir acompañado de una actividad que garantice la calidad". (Pressman, 1997).

Para la evaluación de la calidad de la herramienta se realizan las pruebas de caja blanca a las funciones internas de un módulo, así como las pruebas de caja negra las cuales ejercitan los requisitos funcionales desde el exterior del módulo.

3.4.1. Prueba de Caja Blanca o Estructurales.

Se realizan pruebas de caja blanca para lograr la disminución en un porciento del número de errores existentes en el sistema y una mayor calidad y confiabilidad.

Mediante los métodos de prueba de caja blanca, se puede obtener casos de prueba que:

- Garanticen que se ejercita por lo menos una vez todos los caminos independientes de cada módulo.
- Ejerciten todas las decisiones lógicas en sus vertientes verdadera y falsa.
- Ejecuten todos los bucles en sus límites y con sus límites operacionales.
- Ejerciten las estructuras internas de datos para asegurar su validez." (Pressman, 1997)

3.4.1.1. Camino básico.

Permite obtener una medida de la complejidad lógica de un diseño procedimental y usar esa medida como guía para la definición de un conjunto básico de caminos de ejecución.

Toda sentencia de código tiene una representación en forma de nodo o conjunto de ellos, dentro del grafo de flujo que describe un algoritmo, y los caminos que se pueden definir a través de las aristas que unen los nodos en este grafo representan los posibles caminos básicos de un algoritmo.

Un grafo de flujo está formado por 4 componentes fundamentales. Estos son:

Nodo (N): son los círculos representados en el grafo de flujo, el cual representa una o más secuencias del procedimiento, donde un nodo corresponde a una secuencia de procesos o a una sentencia de decisión. Los nodos que no están asociados se utilizan al inicio y final del grafo.

Aristas (A): son constituidas por las flechas del grafo, son iguales a las representadas en un diagrama de flujo y constituyen el flujo de control del procedimiento. Las aristas terminan en un nodo, aun cuando el nodo no representa la sentencia de un procedimiento.

Regiones (R): áreas delimitadas por las aristas y nodos donde se incluye el área exterior del grafo, como una región más. Se enumeran siendo la cantidad de regiones equivalente a la cantidad de caminos independientes del conjunto básico de un procedimiento.

Nodos predicados (P): cuando en una condición aparecen uno o más operadores lógicos se crea un nodo distinto por cada una de las condiciones simples. Cada nodo generado de esta forma se denomina nodo predicado.

```
public String StateDeclarations(){
    String declarations=""; (1)
    ListaSE<String> declarationsStates= DeclarationsStateName(); (1)
      int cont=1; (1)
      for (int i = 0; i < declarationsStates.size(); i++) {(2)
        declarations+=declarationsStates.get(i)+"="+cont+";\n"; (3)
        cont++;(3)
      }(4)
   return declarations; (5)
} (6)
```

Figura 10. Ejemplo de prueba de caja blanca.

A partir de las sentencias de código enumeradas se crea el grafo de flujo asociado a estas.

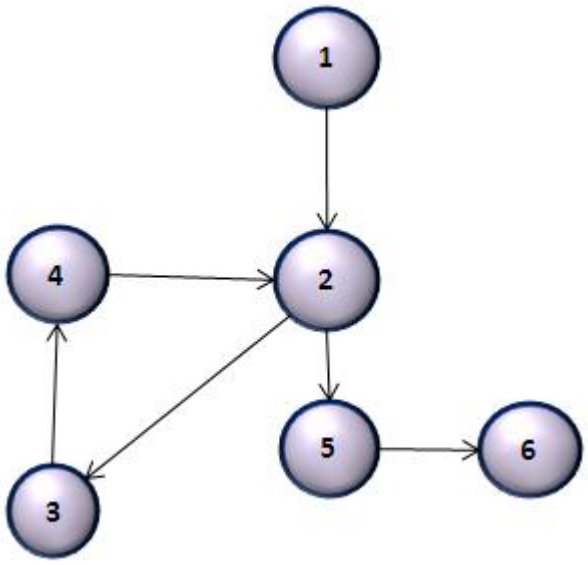

Figura 11. Grafo de flujo asociado a la función StateDeclarations ().

La complejidad ciclomática de este grafo es calculada aplicando las 3 fórmulas existentes para el análisis de la complejidad de algoritmos, garantizando que los resultados obtenidos en cada una de ellas sean los mismos para que el cálculo de la complejidad sea correcto. Estas fórmulas son:

1. V (G) = (A (Aristas) – N (Nodos)) + 2

2. V (G) = P (Nodos Predicados) + 1

3. V (G) = R

Una vez aplicadas estas fórmulas al grafo de flujo de la figura anterior se obtienen los siguientes resultados:

Aplicando la fórmula 1:

V (G) = (6 - 6) + 2

V (G) =2

Aplicando la fórmula 2:

V (G) = 1+ 1

V (G) = 2

Aplicando la fórmula 3:

V (G) =2

Los resultados obtenidos demuestran que la complejidad ciclomática del código es de 3, lo que significa que existen 3 posibles caminos por donde el flujo puede circular, este valor representa el límite mínimo del número total de casos de pruebas para el procedimiento seleccionado.

Camino Básico #1: 1, 2, 5, 6

Camino Básico #2: 1, 2, 3, 4, 5, 6

Casos de prueba para cada camino**.**

Camino: 1, 2, 5, 6

Entrada: lista de nombres de estados adicionados con longitud 0.

Salida: declaración de estados vacía; no se declaran estados.

Precondiciones: Tiene que haberse creado un nuevo proyecto y la lista de nombres de estados adicionados vacía.

Evaluación de los resultados obtenidos: Los resultados de la realización del caso de prueba de caja blanca para el caso de declarar los estados donde fueran nulos dichos estados fueron satisfactorios ya que

al generar el código de un proyecto donde no se hayan adicionado estados la plantilla se generó sin la información correspondiente a dichos estados.

Camino: 1, 2, 3, 4, 5, 6

Entrada: lista de nombres de estados con longitud mayor que 0; se introducen 1 o más estados.

Salida: declaración de todos los estados ingresados de la forma “private byte NombreDeEstado”.

Precondiciones: lista de nombres de estados adicionados no vacía.

Evaluación de los resultados obtenidos: Los resultados de la realización del caso de prueba de caja blanca para el caso de declarar los estados donde se adicionaron estados fueron satisfactorios ya que al generar el código de un proyecto la plantilla se generó con la información correspondiente a los estados.

3.4.2. Prueba de Caja Negra o Funcionales.

Este tipo de pruebas se centran en los requisitos funcionales del software y permiten condiciones de entrada las que ejercitan completamente todos los requisitos funcionales de un programa. Estas se llevan a cabo sobre la interfaz del sistema con el objetivo de reducir el número de casos de prueba mediante la elección de entradas y salidas válidas y no válidas que ejercitan toda la funcionalidad del sistema.

La prueba de caja negra intenta encontrar funciones incorrectas o ausentes, errores de interfaz, en estructuras de datos, errores de rendimiento, errores de inicialización y de terminación. (Pressman, 2002)

3.4.2.1. Diseño de Casos de Prueba de Caja Negra

3.4.2.1.1. Caso de prueba crear nuevo diagrama de estados.

Escenario	Descripción	Existe proyecto abierto	Respuesta del sistema	Flujo
SC1	Crear un diagrama de estados que modele el comportamiento de un applet javacard.	Falso	El sistema muestra un área de trabajo donde se pueden crear los componentes de un diagrama de estado.	Se da click en el botón New Project y se muestra un área de trabajo donde se pueden adicionar los componentes de un diagrama de estados, además de la opción de configurar esta área de trabajo.

Tabla 7. Caso de prueba crear nuevo diagrama de estados.

3.4.2.1.2. Caso de prueba abrir diagrama de estados.

Escenario	Descripción	Existe proyecto	Respuesta del sistema	Flujo
SC1	Cargar correctamente de un archivo origen un diagrama de estados antes realizado.	Verdadero	El sistema muestra el fichero cargado en el área de trabajo de la aplicación.	Se da click en el botón Open Project, se muestra una ventana donde se busca el archivo, se da click en Open y se muestra en el área de trabajo el diagrama correspondiente al archivo cargado.
SC2	Cargar incorrectamente de un archivo origen un diagrama de estados antes realizado.	Falso	El sistema no muestra el fichero que se intenta cargar	Se da click en el botón Open Project, se muestra una ventana donde se busca el archivo, se da click en Open y el sistema no lo abre.

Tabla 8. Caso de prueba abrir diagrama de estados.

3.4.2.1.3. Caso de prueba guardar diagrama de estados.

Escenario	Descripción	Nombre de archivo	Respuesta del sistema	Flujo
SC1	Salva un archivo con la información de un diagrama de estados antes realizado.	Nombre del archivo con el que se guarda.	El sistema guarda el archivo con el nombre señalado en el directorio seleccionado.	Se da click en el botón Save Project, se muestra una ventana donde se busca el directorio donde será guardado, se añade el nombre deseado al archivo y se da click en Save.

Tabla 9. Caso de prueba guardar diagrama de estados.

3.4.2.1.4. Caso de prueba sincronizar código.

Escenario	Descripción	Nombre de archivo	Respuesta del sistema	Flujo

SC1	Se carga un archivo con la información de un applet javacard antes generado	Nombre del archivo guardado.	El sistema muestra un diagrama de estados con los datos de los estados, transiciones y comandos existentes en el applet.	Se da click en el botón Open Project, se muestra una ventana donde se busca el archivo, se da click en Open y se muestra en el área de trabajo el diagrama del archivo cargado.

Tabla 10. Caso de prueba sincronizar código.

3.4.2.1.5. Caso de prueba exportar código javacard.

Escenario	Descripción	Nombre de archivo	Respuesta del sistema	Flujo
SC1	Convierte un diagrama de estados en un applet de javacard. Debe hacerse en extensiones que puedan ser cargadas en otros IDE de desarrollo para el lenguaje javacard.	Nombre del archivo guardado.	El sistema convierte el diagrama de estados creado con anterioridad en código javacard y lo guarda en una dirección url señalada por el desarrollador.	Se da click en el botón Export code, se muestra una ventana donde se indica la dirección donde se desea guardar. Se le indica un nombre al archivo y se da click en Guardar.

Tabla 11. Caso de prueba exportar código javacard.

3.4.2.1.6. Caso de prueba crear nuevo estado.

Escenario	Descripción	Variable Nombre	Respuesta del sistema	Flujo
SC1	Crea un estado inicial.	Es asignado por el sistema.	El sistema crea el estado si no existe y lo muestra en el área de trabajo en forma de círculo relleno de color negro en la posición señalada por el desarrollador.	Se da click en el botón Start state y se da click en la posición de área de trabajo en que desea añadirlo. Luego aparece el estado en forma de círculo relleno.

SC2	Crea un estado final.	Es asignado por el sistema.	El sistema crea el estado si no existe y lo muestra en el área de trabajo en forma de 2 círculos concéntricos siendo el del interior de color negro de en la posición señalada por el desarrollador.	Se da click en el botón End state y se da click en la posición de área de trabajo en que desea añadirlo. Luego aparece el estado en forma de círculo relleno redondeado por otro círculo.
SC3	Crea un estado persistente.	Se introduce un nombre en la ventana que se muestra.	El sistema crea el estado si no existe y lo muestra en el área de trabajo en forma de caja redondeada en la posición señalada y con el nombre introducido por el desarrollador. En caso de no indicar un nombre el sistema le añadirá uno.	Se da click en el botón Persistent state, se introduce un nombre, se da click en OK y se da click en la posición de área de trabajo en que desea añadirlo.
SC4	Crea un estado volátil.	Se introduce un nombre en la ventana que se muestra.	El sistema crea el estado si no existe y lo muestra en el área de trabajo en forma de caja redondeada en la posición señalada y con el nombre introducido por el desarrollador. En caso de no indicar un nombre el sistema le añadirá uno.	Se da click en el botón Volatile state, se introduce un nombre se da click en OK y se da click en la posición de área de trabajo en que desea añadirlo.
SC5	Crea un estado de decisión	Es asignado por el sistema	El sistema crea el estado y lo muestra en el área de trabajo en forma de rombo en la posición señalada por el desarrollador.	Se da click en el botón Desition y se da click en la posición de área de trabajo en que desea añadirlo.

Tabla 12. Caso de prueba crear nuevo estado.

3.4.2.1.7. Caso de prueba eliminar estado.

Escenario	Descripción	Respuesta del sistema	Flujo
SC1	Elimina un estado seleccionado del diagrama y las transiciones asociadas al estado.	El sistema elimina el estado señalado del diagrama de estados.	Se da click derecho sobre el estado que se desee eliminar, se da click en la opción Delete state del menú. El estado es eliminado del área de trabajo.

Tabla 13. Caso de prueba eliminar estado.

3.4.2.1.8. Caso de prueba eliminar transición.

Escenario	Descripción	Respuesta del sistema	Flujo
SC1	Elimina una transición seleccionada del diagrama.	El sistema elimina la transición señalada del diagrama de estados.	Se da click derecho sobre la transición que se desee eliminar, se da click en la opción Delete transition del menú. La transición y el comando asociado son eliminados del área de trabajo.

Tabla 14. Caso de prueba eliminar transición.

3.4.2.1.9. Caso de prueba crear nueva transición.

Escenario	Descripción	Variable Nombre Instrucción	Variable Valor del comando	Respuesta del sistema	Flujo
SC1	Crea una transición con un comando nuevo entre dos estados	Texto deseado por el desarrollador para nombrar el comando.	Valor hexadecimal deseado de dos caracteres.	El sistema muestra una ventana para introducir un comando javacard y el valor del mismo y muestra una flecha desde un estado origen señalado al estado destino indicado con el	Se da click en el botón New Transition, se introduce los datos del comando, se acepta y luego con la combinación Ctrl + Click sobre un estado origen y se arrastra el click a un estado destino, se suelta el click y se muestra la transición con el comando introducido.

				nombre de la instrucción en una caja de texto.	
SC2	Crea una transición con un comando nuevo entre dos estados	Texto deseado por el desarrollador para nombrar el comando.	Valor incorrecto.	El sistema muestra un mensaje indicando que el valor es incorrecto.	Se da click en el botón New Transition, se introduce los datos del comando, se acepta muestra un mensaje indicando que los datos no son correctos.
SC3	Crea una transición con un comando existente en la Api de Plataforma Global 2.1.1 entre dos estados	Texto seleccionado en un combo Box.	El sistema le asigna el valor correspondiente a dicho comando.	El sistema muestra una ventana donde permite seleccionar un comando el cual ya tiene un valor predeterminado.	Se da click en el botón Api Transition, se selecciona un comando, se acepta y luego con la combinación Ctrl + Click sobre un estado origen y se arrastra el click a un estado destino, se suelta el click y se muestra la transición con el comando seleccionado.

Tabla 15. Caso de prueba crear nueva transición.

3.4.2.1.10. Caso de prueba mostrar estructura del espacio de trabajo.

Escenario	Descripción	Respuesta del sistema	Flujo
SC1	El sistema mostrará en forma de árbol la estructura del espacio de trabajo.	El sistema muestra una ventana con los componentes del diagrama en forma de árbol.	De forma automática se muestra en la parte superior izquierda una ventana con los componentes del sistema según sean agregados.

Tabla 16. Caso de prueba mostrar estructura del espacio de trabajo.

3.4.2.1.11. Caso de prueba renombrar componente de un diagrama de estados.

Escenario	Descripción	Variable Nuevo Nombre	Respuesta del sistema	Flujo
SC1	El sistema permite renombrar un estado seleccionado por el usuario dando click derecho sobre del mismo.	Nombre del estado no existente en el diagrama	El sistema muestra una ventana donde se introduce el nuevo nombre, luego se actualiza en el diagrama y en la estructura en forma de árbol del espacio de trabajo.	Se da click derecho sobre el estado, se selecciona renombrar el estado, se introduce el nuevo nombre, se acepta y se actualiza en el diagrama y en la estructura en forma de árbol.
SC2	El sistema no permite renombrar un estado seleccionado por el usuario.	Nombre del estado existente en el diagrama	El sistema muestra una ventana donde se introduce el nuevo nombre y no se actualiza en el diagrama y en la estructura en forma de árbol del espacio de trabajo.	Se da click derecho sobre el estado, se selecciona renombrar el estado, se introduce el nuevo nombre, se acepta y se muestra un mensaje señalando que ya existe.
SC3	El sistema permite renombrar un comando seleccionando una transición dando click derecho sobre la misma.	Nombre del comando deseado por el usuario.	El sistema muestra una ventana donde se introduce el nuevo nombre, luego se actualiza en el diagrama y en la estructura en forma de árbol del espacio de trabajo.	Se da click derecho sobre la transición, se selecciona renombrar el comando, se introduce el nuevo nombre, se acepta y se actualiza en el diagrama y en la estructura en forma de árbol.
SC4	El sistema no permite renombrar un comando seleccionado por el usuario.	Nombre del comando existente en el diagrama	El sistema muestra una ventana donde se introduce el nuevo nombre y no se actualiza en el diagrama y en la estructura en forma de árbol del espacio de trabajo.	Se da click derecho sobre la transición, se selecciona renombrar el comando, se introduce el nuevo nombre, se acepta y se muestra un mensaje señalando que ya existe.

3.5. Tipos de no conformidades.

Una no conformidad es una desviación en los resultados esperados sobre la solución informática. Pueden ser clasificadas en:

- Bloqueantes: Es el tipo de no conformidad que detiene la operación de un programa / componente o hace que este arroje resultados que impiden la continuidad de la operación del cliente. (Oficina de Informática y Telecomunicaciones, 2009)
- Funcionales: Es el tipo de no conformidad que se presenta cuando al ejecutar una opción particular del sistema creada para dar solución a un requerimiento funcional, no se evidencia que el requerimiento se solucione. (Oficina de Informática y Telecomunicaciones, 2009)
- Presentación: Son no conformidades relacionadas con la presentación de los resultados en la ejecución de una opción del sistema; estos deben ajustarse a los estándares definidos y con las reglas gramaticales y ortográficas del idioma en que es presentada la solución. (Oficina de Informática y Telecomunicaciones, 2009)

3.6. Resultados de las pruebas.

En las pruebas de caja negras realizadas se identificaron 11 casos de pruebas con un número de 3 iteraciones con cantidad de escenarios variables para cada uno de ellos. En la primera iteración se realizaron 5 casos de pruebas detectándose una no conformidad bloqueante y una no conformidad en la presentación de los resultados a las cuales se les dio solución. En la segunda iteración se realizaron 3 casos de pruebas donde se detectaron 2 no conformidades funcionales y una en la presentación de los resultados las cuales fueron resueltas. En la tercera iteración se realizaron 3 casos de pruebas detectándose una no conformidad bloqueante la cual fue resuelta.

El total de no conformidades en las 3 iteraciones fue de 6 y todas tuvieron solución al cabo de 5 días. Luego se hicieron 2 iteraciones adicionales para revisar que no existían no conformidades, en la primera de dichas iteraciones se detectó una no conformidad funcional la cual fue resuelta y en la última iteración no se obtuvieron no conformidades. A continuación se representa gráficamente los resultados obtenidos.

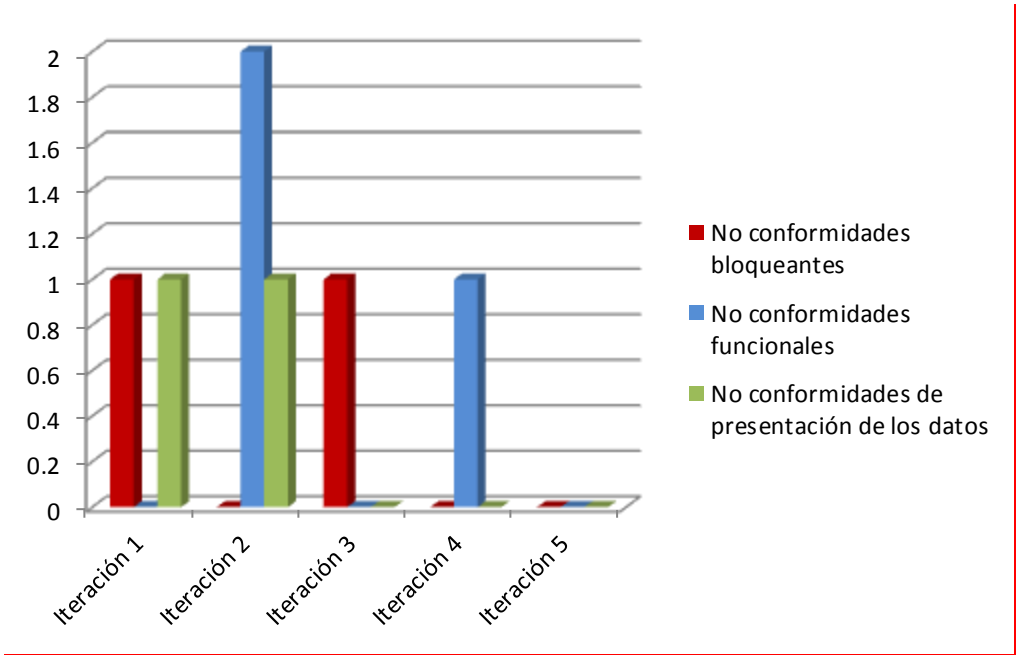

Figura 12. Resultados de prueba de caja negra.

Las no conformidades encontradas son las siguientes:

No conformidades	Iteración	Tipo de no conformidad
Al eliminar un estado si el mismo había sido renombrado no se eliminaba correctamente y no se actualizaba la vista en forma de árbol.	Iteración 1	Bloqueante
Al eliminar una transición si la misma había sido renombrada no se eliminaba correctamente y no se actualizaba la vista en forma de árbol.	Iteración 3	Bloqueante
Cuando se renombraba un estado a la hora de generar el código no se actualizaba el nombre en el código.	Iteración 2	Funcional
Cuando se renombraba una transición no se actualizaba la vista en forma de árbol.	Iteración 1	Presentación
Cuando se cargaba un archivo guardado no se actualizaban los estados y transiciones en la vista en forma de árbol.	Iteración 2	Presentación
Cuando se cargaban los datos de un applets no reconocía las bifurcaciones.	Iteración 2	Funcional
Cuando se cargaban los datos de un applets no reconocía los estados persistentes.	Iteración 4	Funcional

Tabla 17. No conformidades encontradas en la aplicación.

3.7. Prueba para validación de la herramienta generadora de código javacard.

Con el objetivo de comprobar que la herramienta da solución al problema investigativo de reducir tiempo en la implementación de applets javacard al inicio del ciclo de desarrollo, se tomó una muestra de 5 desarrolladores, cada uno de ellos modeló un diagrama de estados en la herramienta creada y generó el código correspondiente al mismo, además utilizando la herramienta de desarrollo para aplicaciones javacard Gemalto Developer Suite cada desarrollador implementó el código correspondiente al mismo diagrama de estados, tomándose cada uno de los tiempos aproximados que demoró cada desarrollador en la implementación de la plantilla del applet y en la generación del código en la herramienta generadora de código. El diagrama de estados utilizado y modelado en la herramienta generadora de código es el siguiente:

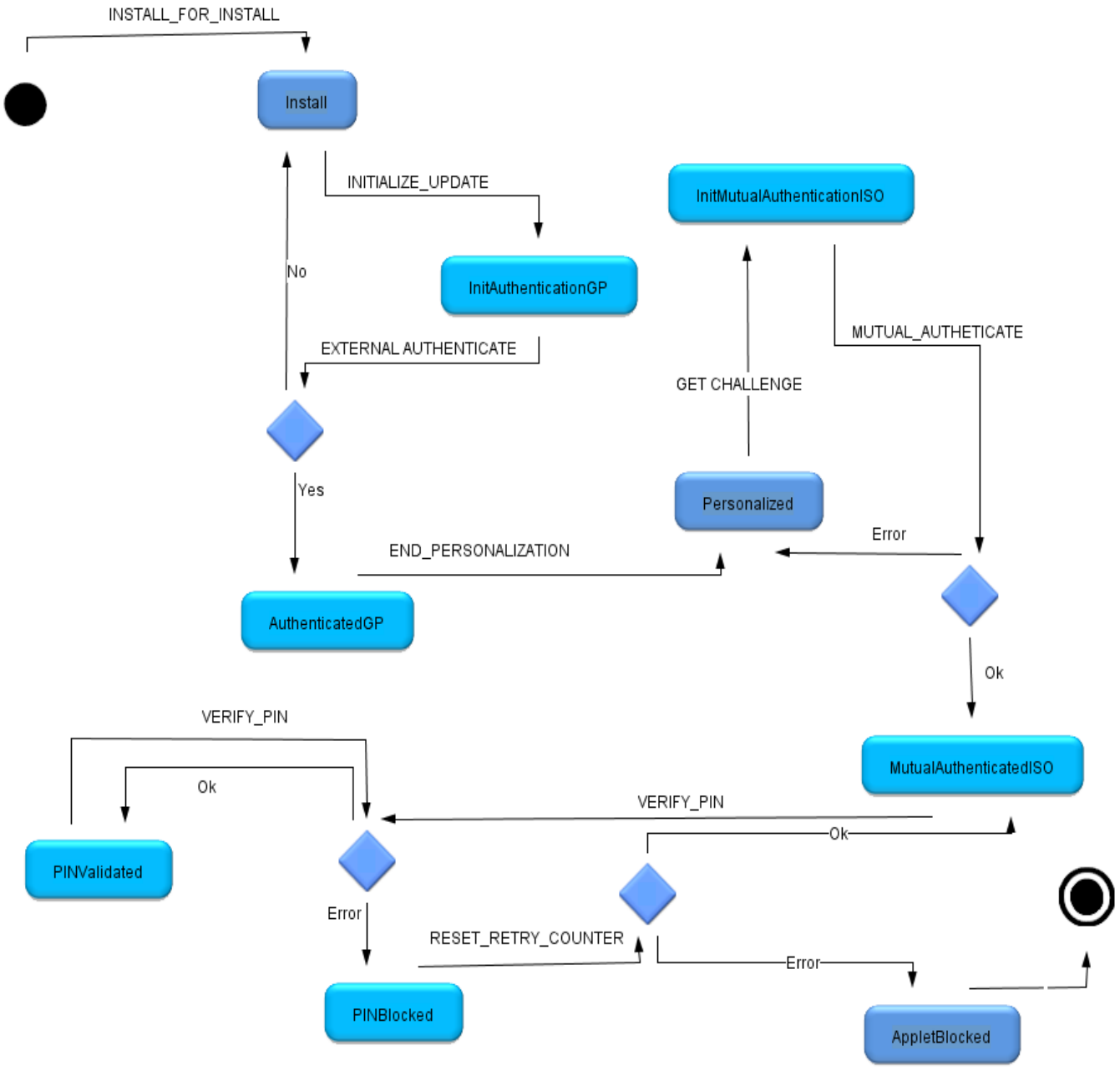

Figura 13. Diagrama de estados modelado en el generador de código; "autenticación de un usuario".

Como se muestra en la tabla, a continuación, los tiempos de desarrollo de forma manual son mayores que los tiempos de la generación de código en la herramienta generadora de código. En el desarrollo de forma manual se cometieron errores de sintaxis del lenguaje javacard, lo que demoró aún más la implementación, por la corrección de dichos errores.

Desarrollador	Duración de la implementación del applet mediante la generación de código en	Duración de la implementación del applet de forma manual en minutos

	minutos	
Desarrollador 1	15	30
Desarrollador 2	12	35
Desarrollador 3	10	38
Desarrollador 4	13	40
Desarrollador 5	15	35
Total	65	178

Tabla 18. Comparación de tiempos de implementación de un applet.

Teniendo en cuenta los resultados arrojados, se concluye que la herramienta reduce significativamente la duración del trabajo de los desarrolladores.

A continuación se muestra un ejemplo de código generado y código de forma manual del applet correspondiente al diagrama de estados de la figura 11.

```
//Tables elements
private static final byte[] Instructions= new byte[]{(byte)0x50,(byte)0x82,(byte)0x0F,(byte)0xB4,(byte)0x61,(byte)0x20,(byte)0x00};

private static final byte[] _Install_Transitions= new byte[]{2,0,0,0,0,0,0};
private static final byte[] _InitAuthenticationGP_Transitions= new byte[]{0,3,0,0,0,0,0};
private static final byte[] _AuthenticatedGP_Transitions= new byte[]{0,0,4,0,0,0,0};
private static final byte[] _Personalized_Transitions= new byte[]{0,0,0,5,0,0,0};
private static final byte[] _InitMutualAuthenticationISO_Transitions= new byte[]{0,0,0,0,6,0,0};
private static final byte[] _MutualAuthenticatedISO_Transitions= new byte[]{0,0,0,0,0,7,0};
private static final byte[] _PINValidated_Transitions= new byte[]{0,0,0,0,0,7,0};
private static final byte[] _PINBlocked_Transitions= new byte[]{0,0,0,0,0,0,6};
private static final byte[] _AppletBlocked_Transitions= new byte[]{0,0,0,0,0,0,0};
```

Figura 14. Código manualmente escrito.

```
private static final byte[] Instructions = new byte[]{(byte)0x50,(byte)0xB2,(byte)0x0F,(byte)0xB4,(byte)0x61,(byte)0x20,(byte)0x00};

private static final byte[] _Install_Transition=new byte[]{3,0,0,0,0,0,0};
private static final byte[] _InitAuthenticationGP_Transition=new byte[]{0,4,0,0,0,0,0};
private static final byte[] _AuthenticatedGP_Transition=new byte[]{0,0,5,0,0,0,0};
private static final byte[] _Personalized_Transition=new byte[]{0,0,0,6,0,0,0};
private static final byte[] _InitMutualAuthenticationISO_Transition=new byte[]{0,0,0,0,7,0,0};
private static final byte[] _MutualAuthenticatedISO_Transition=new byte[]{0,0,0,0,0,8,0};
private static final byte[] _PINValidated_Transition=new byte[]{0,0,0,0,0,8,0};
private static final byte[] _PINBlocked_Transition=new byte[]{0,0,0,0,0,0,7};
private static final byte[] _AppletBlocked_Transition=new byte[]{0,0,0,0,0,0,0};
```

Figura 15. Código generado.

3.8. Conclusiones.

Para garantizar el buen funcionamiento del código implementado, así como su eficiencia y solidez se realizaron las pruebas del camino básico como parte de las pruebas de caja blanca, y para demostrar el cumplimiento de los requisitos definidos, se aplicaron pruebas de caja negra. Todo esto permitió verificar que las funcionalidades implementadas responden a las necesidades y propósitos del desarrollador.

Con el objetivo de validar la aplicación se hizo un estudio en función del tiempo con una cantidad determinada de desarrolladores y así poder observar si ocurre una disminución de esta variable o no, y por lo tanto comprobar que la herramienta da solución al problema planteado al inicio de la investigación.

4. Conclusiones generales.

Con la realización y terminación de este trabajo se llega a las siguientes conclusiones:

- La aplicación de los métodos teóricos y análisis de una amplia bibliografía permitieron concretar el marco teórico en correspondencia con el uso de los generadores de código en el desarrollo de aplicaciones javacard.
- Se desarrolló una aplicación que puede ser utilizada para la generación de código de applets javacard al inicio del ciclo de desarrollo de aplicaciones javacard.
- La aplicación desarrollada podrá ser comercializada en un futuro como uno de los productos del Centro de Identificación y Seguridad Digital (CISED) de la Universidad de las Ciencias Informáticas (UCI).
- Con el uso de tecnologías Java se logró una aplicación multiplataforma lo cual permite ser utilizada en la mayoría de los sistemas operativos existentes en el mercado.

5. Recomendaciones.

Se sugiere para futuras iteraciones y versiones del sistema propuesto:

- Integrar la aplicación como un plugin a la herramienta de desarrollo de aplicaciones javacard Gemalto Developer Suite el cual permita modelar el comportamiento de un applet javacard mediante diagrama de estados.
- Implementar nuevas funcionalidades para dar más comodidades al desarrollador.
- Incluir un sistema de bases de datos donde se puedan guardar los nuevos comandos que se agreguen con cada diagrama que se haga.

6. Bibliografía referenciada.

(s.f.). Obtenido de http://www.Ventajas y Desventajas: Comparación de los Lenguajes C, C++ y JavaV1.1 – AmericaTI.com

(s.f.).

Cepeda., L. A. (2007). *GENERADOR DE CÓDIGO PARA APLICACIONES EN PHP APLICADO AL ERPFAR.* Habana.

Cepeda.v, L. A. (2007). *GENERADOR DE CÓDIGO PARA APLICACIONES EN PHP APLICADO AL ERPFAR.* La habana.

Eclipse. (s.f.). *Plataforma Eclipse: Comunidad en español de Eclipse IDE, el entorno de desarrollo multiplataforma más completo del mundo*. Obtenido de plataformaeclipse: http://plataformaeclipse.com/

Freire, M. R. (s.f.). *API para la gestión de tarjetas inteligentes.* Artemisa.

HERRINGTON, J. (2006). *Code Generation In Action.* Manning.

IBM Corporation,. (s.f.). *IBM Rational solutions for software and systems delivery*. Recuperado el 03 de 2013, de IBM Rational solutions for software and systems delivery: http://www-306.ibm.com/software/rational/

Ivar Jacobson, J. R. (2000). *El lenguaje unificado de modelado. Manual de referencia. .*

Joskowicz, I. J. (2008). *Reglas y Prácticas en eXtreme Programming.*

Kenly Rodríguez Ruiz, Y. C. (2012). *Aplicación para tarjetas inteligentes en una Infraestructura de Clave Pública.* Habana.

Maribel Silva Muñoz, S. R. (s.f.). *Diseño e Implementación de un sistema informático integrado para la Gestión de Compras de Bienes y Contratación de Servicios en los Registros y las Notarías de la República Bolivariana de Venezuela.*

Maribel Silva Muñoz, S. R. (s.f.). *Diseño e Implementación de un sistema informático integrado para la Gestión de Compras de Bienes y Contratación de Servicios en los Registros y las Notarías de la República Bolivariana de Venezuela.*

Oficina de Informática y Telecomunicaciones. (2009). *MP_ASEGURAMIENTO DE CALIDAD DEL SOFTWARE.* Universidad del Valle.

Pressman, R. S. (2002). *Ingeniería del Software. Un enfoque práctico.*

SUN. (s.f.). *Portal del IDE Java de Código Abierto..* Obtenido de http://netbeans.org/index_es.html.

Universidad de la Republica de Uruguay. (2009). Obtenido de http://www.iie.fing.edu.uy/ense/asign/tap/obrar09/...comp/.../VOCABULARIO.doc

Visual Paradigm. (s.f.). *10 Reasons to Choose Visual Paradigm.* Obtenido de 10 Reasons to Choose Visual Paradigm: http://www.visual-paradigm.com/aboutus/10reasons.jsp

Xavier Ferré Grau, M. I. (s.f.). *fermat.usach.c.* Recuperado el 01 de 02 de 2013, de fermat.usach.c: http://http://fermat.usach.cl/~msanchez/comprimido/OBJETOS.pdf

7. Bibliografía consultada.

(n.d.). Retrieved from http://www.Ventajas y Desventajas: Comparación de los Lenguajes C, C++ y JavaV1.1 – AmericaTI.com

(n.d.).

(UNAM), U. N. (2009 - 2013). *Universidad Nacional Autónoma de México*. Retrieved 03 21, 2013, from Universidad Nacional Autónoma de México: http://recursosweb.unam.mx/recursos-web/creacion-de-paginas-web/estandares-de-codificacion/

AmericaTI.com . (2006, 11 11). Retrieved 12 2012, from AmericaTI.com : http://www.AmericaTI.com

java. (2013, 1). Retrieved from java: http://www.java.com/es/about

altova. (n.d.). Retrieved 1 2013, from altova: http://altova.com/umodel

Camejo, R. R. (n.d.). *Desarrollo de una herramienta generadora de ficheros de mapeo, para la persistencia de objetos en esquemas relacionales basada en Doctrine.* La Habana.

Cepeda., L. A. (2007). *GENERADOR DE CÓDIGO PARA APLICACIONES EN PHP APLICADO AL ERPFAR.* Habana.

Cepeda.v, L. A. (2007). *GENERADOR DE CÓDIGO PARA APLICACIONES EN PHP APLICADO AL ERPFAR.* La habana.

Chalmers University of Technology, G. S. (n.d.). Rigorous development of JavaCard.

Cuesta, M. Y. (2010). *Desarrollo de la versión 2.0 de la herramienta Doctrine Generator para la generación de ficheros de mapeo basado en Doctrine empleando la tecnología PHP.* La Habana.

Eclipse. (n.d.). *Plataforma Eclipse: Comunidad en español de Eclipse IDE, el entorno de desarrollo multiplataforma más completo del mundo*. Retrieved from plataformaeclipse: http://plataformaeclipse.com/

Enrique, Y. H. (2010). *Desarrollo de una Herramienta generadora de.* La Habana.

Freire, M. R. (n.d.). *API para la gestión de tarjetas inteligentes.* Artemisa.

GlobalPlatform. (2003). *GlobalPlatform.*

Gómez, G. L. (n.d.). Actores y sus roles. *Actores y sus roles.* Medellín, Colombia : Escuela de Sistemas.

Gómez, R. P. (n.d.). *Herramientas Case.* Veracruz.

González, H. L. (2009). *Trabajo de Diploma para optar por el título de Ingeniero Informático.* La Habana.

HERRINGTON, J. (2006). *Code Generation In Action.* Manning.

IBM Corporation,. (n.d.). *IBM Rational solutions for software and systems delivery*. Retrieved 03 2013, from IBM Rational solutions for software and systems delivery: http://www-306.ibm.com/software/rational/

Informática, I. N. (1999). *Herramientas Case.* Talleres de la Oficina de Impresiones de la Oficina Técnica de Difusión.

Ivar Jacobson, J. R. (2000). *El lenguaje unificado de modelado. Manual de referencia. .*

Java Card Management (JCM). (2000). JAVA CARD Management Specification. In J. C. (JCM), *JAVA CARD Management Specification.*

javamexico. (n.d.). Retrieved 2012, from javamexico: http://www.javamexico.org/blogs/skuarch/compendio_de_apis_librerias_frameworks_herramientas_plug_ins_y_lenguajes

Joskowicz, I. J. (2008). *Reglas y Prácticas en eXtreme Programming.*

Kendall, K. &. (n.d.). *Analisis Y Diseño De Sistemas 3ª. Edición.*

Kenly Rodríguez Ruiz, Y. C. (2012). *Aplicación para tarjetas inteligentes en una Infraestructura de Clave Pública.* Habana.

Luque, D. R. (2007). *Herramienta para la generación de código.* La Habana.

Maribel Silva Muñoz, S. R. (n.d.). *Diseño e Implementación de un sistema informático integrado para la Gestión de Compras de Bienes y Contratación de Servicios en los Registros y las Notarías de la República Bolivariana de Venezuela.*

Maribel Silva Muñoz, S. R. (n.d.). *Diseño e Implementación de un sistema informático integrado para la Gestión de Compras de Bienes y Contratación de Servicios en los Registros y las Notarías de la República Bolivariana de Venezuela.*

Marín, R. S. (2009). *"JASCOG 1.0". Herramienta para la generación de código JavaScript para la librería ExtJS. .* La Habana.

Mario Rincón Nigro1, J. A. (2010). *Generación Automática de Código a Partir de Máquinas de Estado.* Mérida. Venezuela.

Moreno, P. J. (2003). *Especificación de interfaz de usuario: De los requisitos a la generación automática.* Universidad de Valencia.

Mostowski, W. (2006). Systematic Development of JAVA CARD Applets.

msdm. (n.d.). Retrieved 03 21, 2013, from http://msdn.microsoft.com/es-es/library/aa291593(v=vs.71).aspx

msdn. (n.d.). Retrieved 12 2012, from msdn: http://msdn.microsoft.com/es-es/library/ee329480.aspx

Oficina de Informática y Telecomunicaciones. (2009). *MP_ASEGURAMIENTO DE CALIDAD DEL SOFTWARE.* Universidad del Valle.

ongei. (n.d.). Retrieved 2012, from ongei: http://www.ongei.gob.pe/publica/metodologias/Lib5083/cap2016.HTM

Oracle. (n.d.). *Applet (Java 2 Platform SE 5.0).* Retrieved from http://docs.oracle.com/javase/1.5.0/docs/api/java/applet/Applet.html

Penadés, P. L. (n.d.). *Métodologías ágiles para el desarrollo de software: eXtreme Programming (XP) .* Valencia.

Pressman, R. S. (2002). *Ingeniería del Software. Un enfoque práctico.*

Reynoso, C. B. (2004). Introducción a la Arquitectura de Software. In C. B. Reynoso, *Introducción a la Arquitectura de Software.* Buenos Aires.

Ricardo Grau, C. C. (2004). *Metodologia de la investigación.*

Sistema Integrado de Información Universitaria. (2010). *Especificación de Requisitos.*

slideshare. (n.d.). Retrieved 2012, from slideshare: http://www.slideshare.net/aoteroe/libreras-de-java#btnNext APRENDER JAVA

Suárez, A. R. (2009). *Validación de herramientas.* La Habana.

SUN. (n.d.). *Portal del IDE Java de Código Abierto..* Retrieved from http://netbeans.org/index_es.html.

Universidad de la Republica de Uruguay. (2009). Retrieved from http://www.iie.fing.edu.uy/ense/asign/tap/obrar09/...comp/.../VOCABULARIO.doc

Vasilios Almaliotis, A. L. (n.d.). *dmst.* Retrieved 2013, from dmst: http://www.dmst.aueb.gr/dds/pubs/conf/2008-CARDIS-JCard/html/ALKLS08.htm

Vera, L. C. (2008). *Herramienta de Generación de Código Mediante Sistema Experto.* la Habana.

Visual Paradigm. (n.d.). *10 Reasons to Choose Visual Paradigm.* Retrieved from 10 Reasons to Choose Visual Paradigm: http://www.visual-paradigm.com/aboutus/10reasons.jsp

Vlášek, P. (2008). *Java Card Manager NetBeans plug-in user guide.*

Xavier Ferré Grau, M. I. (n.d.). *fermat.usach.c.* Retrieved 02 01, 2013, from fermat.usach.c: http://http://fermat.usach.cl/~msanchez/comprimido/OBJETOS.pdf

8. Glosario de términos.

API: Una Interfaz de Programación de Aplicaciones (del inglés Application Programming Interface (API)) es el conjunto de funciones y procedimientos (o métodos si se refiere a programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

Applet: Aplicación que se ejecutan dentro de las tarjetas inteligentes y gestiona la información almacenada en ellas.

Arquetipo: En el actual contexto se refiere a la estructura del código generado, una plantilla donde estará el código correspondiente al diagrama de estados y que luego el desarrollador modificará.

CASE: Ingeniería de Software Asistida por Computadoras (Computer Aided Software Engineering). Es el uso de asistencia de software para organizar y controlar el desarrollo de software.

Código: Cifras, clave. Conjunto de instrucciones en un lenguaje de programación.

Diagrama de estados: Muestran el conjunto de estados por los cuales pasa un objeto durante su vida en una aplicación en respuesta a eventos.

IDES: Ambiente integrado de desarrollo (Integrated Development Environment). Conjunto de software que permite el desarrollo de aplicaciones.

Ingeniería inversa: Obtención de información y características técnicas a partir de un producto ya creado.

Ingeniería directa: Es el proceso de producción del código de una aplicación a partir de sus especificaciones.

Java: Lenguaje de programación de alto nivel orientado a objetos y multiplataforma.

Javacard: Tecnología que permite ejecutar de forma segura pequeñas aplicaciones Java (applets) en tarjetas inteligentes y dispositivos similares.

Java Development Kit o (JDK): Es un software que provee herramientas de desarrollo para la creación de programas en Java.

Modelo: Representación abstracta de la realidad. Diagramas que representan la estructura de un sistema dado.

Multiplataforma: Es un término usado para referirse a los programas, sistemas operativos o lenguajes de programación que puedan funcionar en diversas plataformas como Windows o Linux.

Patrones: Normas de comportamiento, características que identifican una situación. Solución a un problema de diseño no trivial que es efectiva y reutilizable.

Software: Término genérico que designa al conjunto de programas que posibilitan realizar una tarea específica en un ordenador.

UML: Lenguaje Unificado de Modelado (Unified Modeling Language). Lenguaje gráfico para visualizar, especificar, construir y documentar un sistema de software.

W3C (World Wide WEB Consortium): Consorcio internacional que produce estándares para la WEB.

XML: Lenguaje de marcas extensible (eXtensible Markup Language).

9. Anexos.

Anexo1: Cronograma de trabajo.

No.	Acciones a realizar	Responsable	Fecha de entrega
1.	Análisis de las herramientas generadoras de códigos existentes.	Yoel Amed Rendón Zurbano Marisela Campos Donal	09/10/2012
2.	Análisis de la sintaxis del lenguaje javacard a tener en cuenta para la generación de código.	Yoel Amed Rendón Zurbano	30/12/2012
3.	Análisis del lenguaje y las herramientas de programación.	Yoel Amed Rendón Zurbano	16/10/2012
4.	Descripción de las tecnologías, metodologías y estándares más adecuados para el desarrollo de la solución.	Marisela Campos Donal	16/10/2012
5.	Definición y refinamiento de los	Yoel Amed Rendón Zurbano Marisela Campos Donal	12/12/2012
6.	Diseño del prototipo de interfaz de usuario.	Yoel Amed Rendón Zurbano	14/01/2013
7.	Definición de plantillas de diagramas de estado para los principales estándares de comunicación y gestión de información de tarjetas.	Marisela Campos Donal	20/01/2013
8.	Implementación de los requerimientos definidos para la solución haciendo uso de patrones de diseño.	Marisela Campos Donal Yoel Amed Rendón Zurbano	14/4/2012
9.	Validación del prototipo de interfaz diseñado.	Marisela Campos Donal Yoel Amed Rendón Zurbano	15/4/2012
10.	Realización de pruebas de unidad a la	Marisela Campos Donal	14/4/2012

	herramienta desarrollada.	Yoel Amed Rendón Zurbano	
11.	Realización de pruebas de aceptación a la herramienta.	Yoel Amed Rendón Zurbano Marisela Campos Donal	14/4/2012

Tabla 19. Cronograma de trabajo.

Anexo2: Historia de usuario exportar código javacard.

Historia de Usuario	
Número: HU_3	**Nombre de Historia de Usuario**: Exportar código javacard.
Modificación de Historia de Usuario Número: Ninguna	
Usuario: Desarrollador	**Iteración asignada**: 1
Prioridad en negocio: Alta	**Puntos estimados:** 2.5
Riesgo en desarrollo: Medio	**Puntos reales:** 3
Descripción: Acción de orden primario que debe realizar la conversión de un diagrama de estado a un applet a código javacard. Debe hacerse en extensiones que puedan ser cargadas en otros IDE de desarrollo para el lenguaje javacard.	
Observaciones: La herramienta verificará que exista un diagrama de estados creado, da la opción de seleccionar si es una plantilla con entorno seguro y luego exporta el mismo.	

Tabla 20. HU exportar código javacard.

Anexo3: Historia de usuario crear nuevo estado.

Historia de Usuario	
Número: HU_4	**Nombre de Historia de Usuario**: Crear nuevo estado.
Modificación de Historia de Usuario Número: Ninguna	
Usuario: Desarrollador	**Iteración asignada**: 1

Prioridad en negocio: Alta	**Puntos estimados:** 0.5
Riesgo en desarrollo: Medio	**Puntos reales:** 0.3
Descripción: Permite crear un estado del sistema que puede ser un estado inicial, persistente, volátil o final. Cada uno de los estados va a tener un nombre único y según las transiciones un listado de hijos los cuales van a ser los estados a los cuales se puede llegar directamente cuando suceda algún evento.	
Observaciones: El sistema permite poner un nombre al estado y la posición del área de trabajo donde lo dibujará.	

Tabla 21. Historia de usuario crear nuevo estado.

Anexo4: Historia de usuario eliminar estado.

Historia de Usuario	
Número: HU_5	**Nombre de Historia de Usuario**: Eliminar estado.
Modificación de Historia de Usuario Número: Ninguna	
Usuario: Desarrollador	**Iteración asignada**: 1
Prioridad en negocio: Alta	**Puntos estimados:** 0.5
Riesgo en desarrollo: Medio	**Puntos reales:** 0.4
Descripción: Permite eliminar un estado seleccionado del diagrama y las transiciones asociadas al estado.	
Observaciones: Se selecciona un estado y se presiona eliminar, luego el sistema lo elimina del visual.	

Tabla 22. Historia de usuario eliminar estado.

Anexo5: Historia de usuario crear nuevo evento.

Historia de Usuario	
Número: HU_6	**Nombre de Historia de Usuario**: Crear nuevo evento.
Modificación de Historia de Usuario Número: Ninguna	
Usuario: Desarrollador	**Iteración asignada**: 1

Prioridad en negocio: Alta	**Puntos estimados:** 1.5
Riesgo en desarrollo: Medio	**Puntos reales:** 1.6
Descripción: Es la acción que provoca exista una transición de un estado a otro, como condición deben existir inicialmente al menos un estado volátil y uno persistente, además de los estados inicial y final.	
Observaciones: Se selecciona un estado origen presionando Ctrl –Click y se arrastra a un estado destino soltando las teclas. El sistema da la opción de ponerle el nombre deseado, mostrándolo al igual que la transición representada con una flecha.	

Tabla 23. Historia de usuario crear nuevo evento.

Anexo6: Historia de usuario eliminar transición.

Historia de Usuario	
Número: HU_7	**Nombre de Historia de Usuario**: Eliminar transición.
Modificación de Historia de Usuario Número: Ninguna	
Usuario: Desarrollador	**Iteración asignada**: 1
Prioridad en negocio: Alta	**Puntos estimados**: 0.3
Riesgo en desarrollo: Medio	**Puntos reales:** 0.1
Descripción: Permite eliminar una transición seleccionada en el diagrama la instrucción a la transición.	
Observaciones: El sistema elimina una transición y la instrucción asociada señalada por el desarrollador.	

Tabla 24. Historia de usuario eliminar transición.

Anexo7: Historia de usuario mostrar estructura del espacio de trabajo.

Historia de Usuario	
Número: HU_8	**Nombre de Historia de Usuario**: Mostrar estructura del espacio de trabajo.

Modificación de Historia de Usuario Número: Ninguna	
Usuario: Desarrollador	**Iteración asignada**: 1
Prioridad en negocio: Alta	**Puntos estimados**: 0.1
Riesgo en desarrollo: Medio	**Puntos reales:** 0.1
Descripción: El sistema mostrará en forma de árbol la estructura del espacio de trabajo.	
Observaciones: El sistema muestra una estructura donde se adicionan cada uno de los componentes del diagrama de estados.	

Tabla 25. Historia de usuario mostrar estructura del espacio de trabajo.

Anexo8: Historia de usuario sincronizar código.

Historia de Usuario	
Número: HU_9	**Nombre de Historia de Usuario**: Sincronizar código.
Modificación de Historia de Usuario Número: Ninguna	
Usuario: Desarrollador	**Iteración asignada**: 1
Prioridad en negocio: Alta	**Puntos estimados:** 10
Riesgo en desarrollo: Medio	**Puntos reales:** 15
Descripción: Permite cargar un código de un applet javacard de un fichero y visualizarlo en forma de diagrama al que se le pueden hacer cambios.	
Observaciones: El sistema da la opción de seleccionar un fichero con el código javacard y luego muestra un diagrama de estados.	

Tabla 26. Historia de usuario sincronizar código.

Anexo9: Historia de usuario Renombrar

Historia de Usuario

Número: HU_10	**Nombre de Historia de Usuario**: Renombrar
Modificación de Historia de Usuario Número: Ninguna	
Usuario: Desarrollador	**Iteración asignada**: 1
Prioridad en negocio: Media	**Puntos estimados:** 0.5
Riesgo en desarrollo: Medio	**Puntos reales:** 2
Descripción: Permite renombrar los estados y las instrucciones.	
Observaciones: El sistema muestra una ventana donde se introduce el nuevo nombre deseado y luego se actualiza la caja de texto correspondiente al elemento del diagrama de estados.	

Tabla 27. Historia de usuario renombrar.

Anexo10: Historia de usuario Guardar Diagrama.

Historia de Usuario	
Número: HU_10	**Nombre de Historia de Usuario**: Guardar diagrama.
Modificación de Historia de Usuario Número: Ninguna	
Usuario: Desarrollador	**Iteración asignada**: 1
Prioridad en negocio: Media	**Puntos estimados:** 0.5
Riesgo en desarrollo: Medio	**Puntos reales:**0.5
Descripción: Permite guardar los estados y las instrucciones correspondiente a un diagrama de estados.	
Observaciones: El sistema muestra una ventana donde se introduce el nombre deseado del diagrama que se desea guardar, se selecciona la dirección donde será guardado en forma de XML.	

Tabla 28. Historia de usuario guardar diagrama de estados.

Anexo11: Plan de entrega.

Entregable	Fin Iteración 1	Fin Iteración 2	Fin Iteración 3	Fin Iteración 4	Fin Iteración 5
Aplicación la generación de código javacard a partir de diagramas de estado.	Enero 2012	Marzo 2012	Abril 2012	Mayo 2012	Junio 2012

Tabla 29. Plan de entrega.

Anexo12: Diagrama de clases.

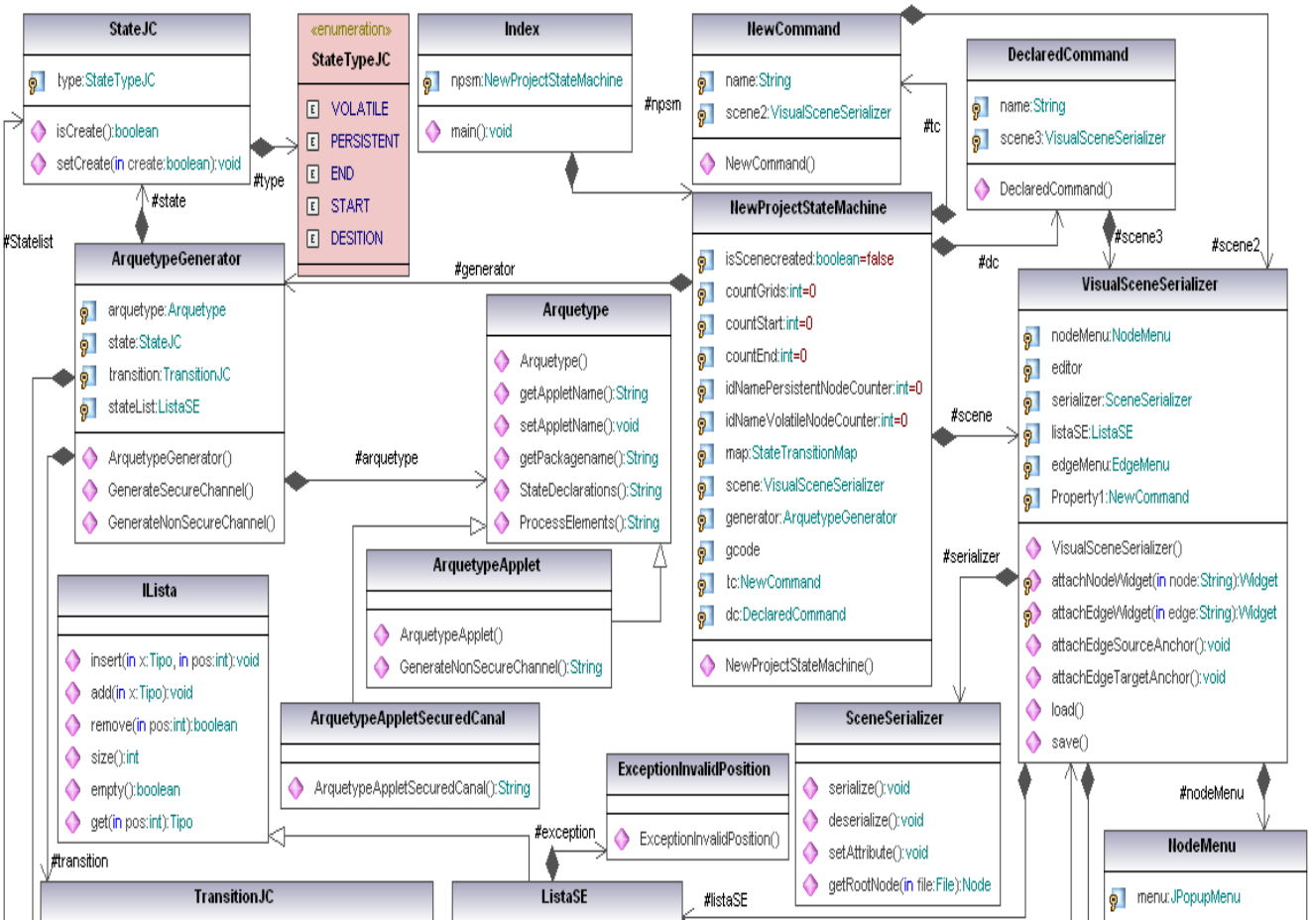

Figura 16. Fragmento 1 del diagrama de clases.

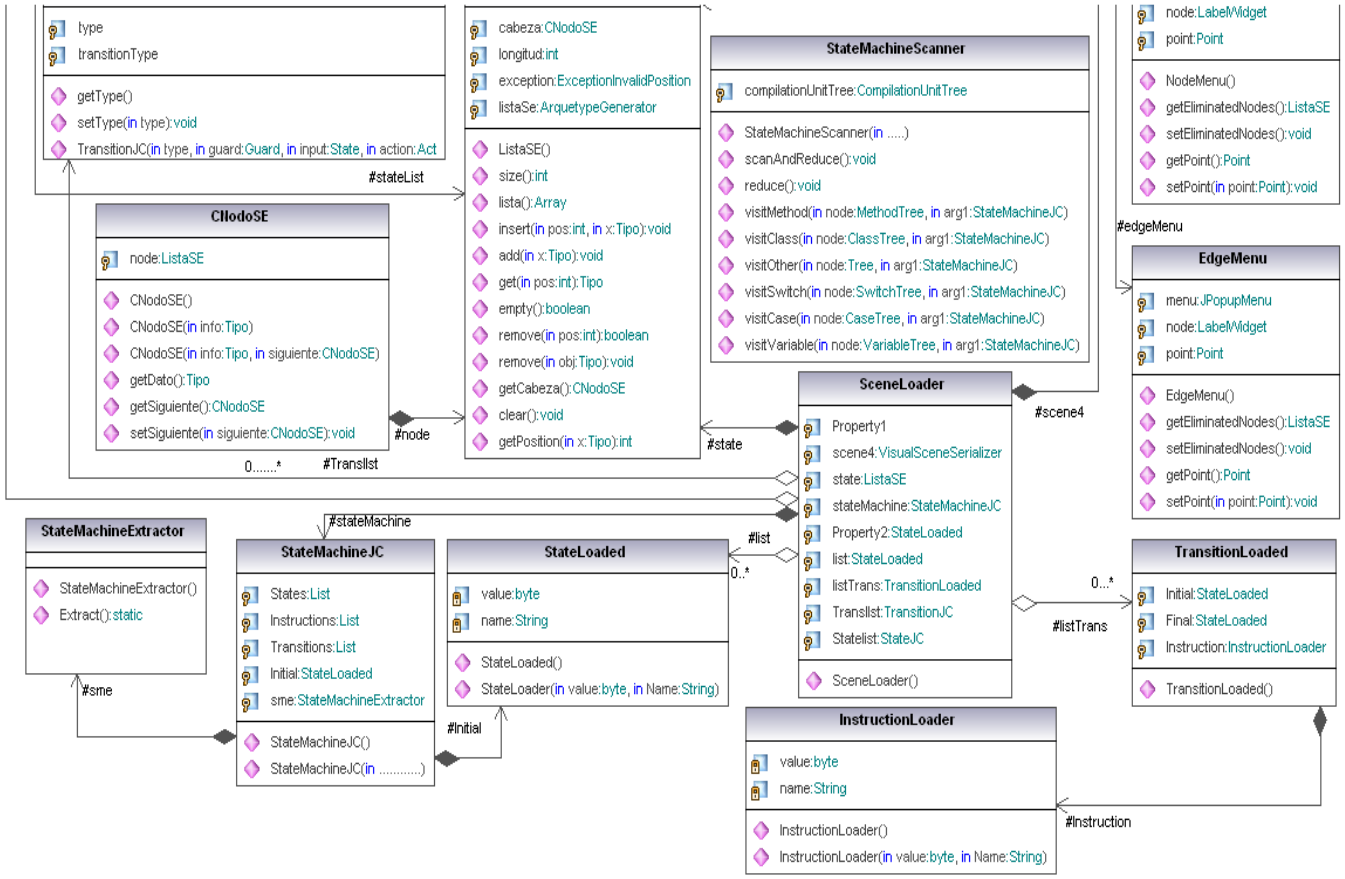

Figura 17. Fragmento 2 del diagrama de clases.

Anexo13: Clase Archetype.

Nombre	Archetype
Responsabilidades	Permite crear un arquetipo para la generación del código.
Clases de las que depende	

Tabla 30. Clase Archetype.

Anexo14: Clase ArchetypeNonSecuredChannel.

Nombre	ArchetypeNonSecuredChannel
Responsabilidades	Permite crear un arquetipo sin canal seguro en la generación del código.

Clases de las que depende	Archetype

Tabla 31. Clase ArchetypeNonSecuredChannel.

Anexo15: Clase ArchetypeAppletSecureChannel.

Nombre	ArchetypeAppletSecureChannel
Responsabilidades	Permite crear un arquetipo sin canal seguro en la generación del código.
Clases de las que depende	Archetype

Tabla 32. Clase ArchetypeAppletSecureChannel.

Anexo16: Clase StateJC.

Nombre	**StateJC**
Responsabilidades	Permite crear un estado javacard y guarda los datos.
Clases de las que depende	State (clase de la librería FSM)

Tabla 33. Clase StateJC.

Anexo17: Enumerativo StateTypeJC.

Nombre	StateTypeJC
Responsabilidades	Contiene las diferentes categorías de estados permisibles en un diagrama de estado.
Clases de las que depende	

Tabla 34. Enumerativo StateTypeJC.

Anexo18: Clase TransitionJC.

Nombre	**TransitionJC**

Responsabilidades	Clase que permite crear una transición y gestionar los datos de su creación.
Clases de las que depende	StateJC

Tabla 35. Clase TransitionJC.

Anexo19: Clase CNodoSE.

Nombre	**CNodoSE**
Responsabilidades	Crea la estructura primaria para trabajar con las listas simplemente enlazadas
Clases de las que depende	

Tabla 36. Clase CNodoSE.

Anexo20: Clase ExceptionInvalidPosition.

Nombre	ExceptionInvalidPosition
Responsabilidades	Excepción para el control de la inserción de elementos según la lista.
Clases de las que depende	

Tabla 37. Clase ExceptionInvalidPosition.

Anexo21: Interfaz ILista.

Nombre	ILista<Tipo>
Responsabilidades	Interfaz que declara los métodos que se deben declarar en las clases hijas.
Clases de las que depende	Serializable

Tabla 38. Interfaz ILista.

Anexo22: Clase ListaSE.

Nombre	ListaSE
Responsabilidades	Clase que permite crear y manejar una lista de tipo x donde x puede ser cualquier elemento.
Clases de las que depende	ILista Serializable Observable

Tabla 39. Clase ListaSE.

Anexo23: Clase ExportApplet.

Nombre	ExportApplet
Responsabilidades	Permite exportar una plantilla de un applet.
Clases de las que depende	ArchetypeGenerator

Tabla 40. Clase LabelTextFieldEditor.

Anexo24: Clase Index.

Nombre	Index
Responsabilidades	Clase que establece el vínculo a la aplicación.
Clases de las que depende	New Project State Machine.

Tabla 41. Clase Index.

Anexo25: Clase VisualSceneSerializer.

Nombre	VisualSceneSerializer
Responsabilidades	Crea un campo de texto editable y permite asignarlo a cualquier elemento visual.
Clases de las que depende	LabelTextFieldEditor

	LayerWidget WidgetAction Observable ListaSE LayerWidget ConnectionWidget WidgetAction Image MyHoverProvider

Tabla 42. Clase VisualSceneSerializer.

Anexo26: Clase NewProjectStateMachine.

Nombre	NewProjectStateMachine
Responsabilidades	Clase visual principal que permite visualizar todos los elementos de un diagrama de estado durante su creación.
Clases de las que depende	SceneSerializerTest ListaSE String

Tabla 43. Clase NewProjectStateMachine.

Anexo27: Clase NodeMenu.

Nombre	NodeMenu
Responsabilidades	Contiene los elementos principales para agregarle un menú a los nodos que se muestran en el sistema.
Clases de las que depende	PopupMenuProvider

	ActionListener

Tabla 44. Clase NodeMenu.

Anexo28: Clase SceneLoader.

Nombre	SceneLoader
Responsabilidades	Permite obtener los datos en el diagrama de estados cargado de un applet y guardar dichos datos para convertirlos en un diagrama de estados y luego generar código javacard.
Clases de las que depende	StateMachineJC StateLoaded TransitionLoaded StateJC TransitionJC InstructionLoaded

Tabla 45. Clase SceneLoader.

Anexo29: Clase TreeJC.

Nombre	**TreeJC**
Responsabilidades	Contiene los atributos principales para agregarle los elementos adicionados al diagrama de estados al árbol de componentes.
Clases de las que depende	DefaultTreeModel Jtree DafaultMutableTreeNode

Tabla 46. Clase EdgeMenu.

Anexo30: Clase NewCommand.

Nombre	**NewCommand**

Responsabilidades	Clase visual que permite agregarle nuevos comandos APDUs a los eventos de un diagrama de estado durante su creación.
Clases de las que depende	SceneSerializerTest

Tabla 47. Clase TransitionCommand.

Anexo31: Clase APICommand.

Nombre	**APICommand**
Responsabilidades	Clase visual que permite agregarle comandos APDUs existentes en la API de javacard a los eventos de un diagrama de estado durante su creación.
Clases de las que depende	SceneSerializerTest

Tabla 48. Clase APICommand.

Anexo32: Clase InstructionLoaded.

Nombre	InstructionLoaded
Responsabilidades	Permite crear una transición obtenida de un applet javacard.
Clases de las que depende	String

Tabla 49. Clase InstructionLoaded.

Anexo33: Clase StateLoaded.

Nombre	StateLoaded
Responsabilidades	Permite crear un estado obtenido de un applet javacard.
Clases de las que depende	String

Tabla 50. Clase StateLoaded.

Anexo34: Clase StateMachineExtractor.

Nombre	StateMachineExtractor
Responsabilidades	Permite obtener los datos de un applet javacard.
Clases de las que depende	StateMachineScanner

Tabla 51. Clase StateMachineExtractor.

Anexo35: Clase StateMachineJC.

Nombre	StateMachineJC
Responsabilidades	Permite crear un diagrama de estados con los datos correspondientes a un applet javacard.
Clases de las que depende	StateLoaded TransitionLoaded InstructionLoaded

Tabla 52. Clase StateMachineJC.

Anexo36: Clase StateMachineScanner.

Nombre	StateMachineScanner
Responsabilidades	Obtiene y reconoce la estructura de un applet javacard para obtener sus datos
Clases de las que depende	CompilationUnitTree SourcePositions LineMap StateMachineJC

Tabla 53. Clase StateMachineScanner.