INSTITUTO POLITÉCNICO NACIONAL

UNIDAD PROFESIONAL INTERDISCIPLINARIA DE INGENIERÍA
Y CIENCIAS SOCIALES Y ADMINISTRATIVAS

SECCIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

MODELO DE SIMULACIÓN VIAL

BASADO EN AGENTES DE SOFTWARE

TESIS

QUE PARA OBTENER EL GRADO DE

MAESTRO EN CIENCIAS EN INFORMÁTICA

PRESENTA:

LUIS ANDRÉS SOTO OSORIO

DIRECTOR:

M. en C. EDUARDO RENÉ RODRÍGUEZ ÁVILA

México, D.F. 2013

SIP-14

INSTITUTO POLITÉCNICO NACIONAL
SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

ACTA DE REVISIÓN DE TESIS

En la Ciudad de México, D.F siendo las 18:00 horas del día 19 del mes de NOVIEMBRE del 2013 se reunieron los miembros de la Comisión Revisora de Tesis, designada por el Colegio de Profesores de Estudios de Posgrado e Investigación de UPIICSA para examinar la tesis titulada:

"MODELO DE SIMULACIÓN VIAL BASADO EN AGENTES DE SOFTWARE"

Presentada por el alumno:

SOTO	OSORIO	LUIS ANDRÉS
Apellido paterno	Apellido materno	Nombre(s)

Con registro: B 0 9 1 9 7 2

aspirante de:

MAESTRÍA EN CIENCIAS EN INFORMÁTICA

Después de intercambiar opiniones, los miembros de la Comisión manifestaron ***APROBAR LA DEFENSA DE LA TESIS***, en virtud de que satisface los requisitos señalados por las disposiciones reglamentarias vigentes.

LA COMISIÓN REVISORA

Director de tesis

M. EN C. EDUARDO RENÉ RODRIGUEZ ÁVILA

DR. MAURICIO JORGE PROCEL MORENO

M. EN C. ABEL BUENO MEZA

M. EN C. GUILLERMO PÉREZ VÁZQUEZ

DR. FERNANDO VÁZQUEZ TORRES

EL PRESIDENTE DEL COLEGIO DE PROFESORES

M. EN C. GUSTAVO MAZCORRO TÉLLEZ

S. E. P.
I. P. N.
U. P. I. I. C. S. A
SECCIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

CARTA DE CESIÓN DE DERECHOS

En la ciudad de México, el día 20 de noviembre de 2013, el que suscribe **Luis Andrés Soto Osorio**, alumno del Programa de Maestría en Ciencias en Informática con número de registro B091972, adscrito a la Unidad Profesional Interdisciplinaria de Ingeniería y Ciencias Sociales y Administrativas, manifiesta que es autor intelectual del presente trabajo de Tesis bajo la dirección del **M. en C. Eduardo René Rodríguez Ávila** y cede los derechos del trabajo intitulado **MODELO DE SIMULACIÓN VIAL BASADO EN AGENTES DE SOFTWARE** al Instituto Politécnico Nacional para su difusión con fines académicos y de investigación.

Los usuarios de la información no podrán reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expreso del autor y/o director del mismo. Este puede ser obtenido escribiendo a la siguiente dirección **luis.soto.osorio@gmail.com**. Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.

Luis Andrés Soto Osorio

Agradecimientos

"Si he logrado ver más lejos, es porque he subido a hombros de gigantes"
Isaac Newton

A mi esposa Julieta por acompañarme y ser parte de este proceso, por tu paciencia y apoyo constantes y, principalmente, por darme la oportunidad de aprender a ver la vida desde una perspectiva diferente y por motivarme a ser una mejor persona día con día. Te amo.

A mi padre Gilberto, por el enorme sacrificio de dedicar una vida a educar a tres hijos sin nada más que una gran fuerza de voluntad, responsabilidad y amor. De no haber sido por ti esto jamás hubiera sido posible. Gran parte de este logro es también tuyo, papá.

A mis hermanos Alejandra y Mario, por ser testigos, cómplices y compañeros, desde mi infancia, de miles de experiencias que me hicieron crecer como persona y que ahora como adulto dibujan una sonrisa en mi rostro cada que las recuerdo.

A mis amigos y compañeros, por todas las vivencias, las memorias, las alegrías y las tristezas que hemos compartido y que nos faltan por vivir. Gracias por permitirme compartir parte del camino con todos ustedes.

A mi director de tesis, por su orientación, apoyo y guía sin las cuáles no hubiera logrado culminar este trabajo.

RESUMEN

Los agentes de *software*, así como los sistemas multi-agentes, representan una nueva manera de adentrarse en el análisis, diseño e implementación de sistemas complejos. El enfoque basado en agentes ofrece una potente colección de técnicas y herramientas con un gran potencial para mejorar de manera considerable la manera en que las personas conceptualizan e implementan muchos tipos de software que van desde sistemas relativamente pequeños, como los filtros personalizados de correo electrónico, hasta grandes y complejos sistemas de misión crítica como los utilizados para el control de tráfico aéreo, por citar sólo unos ejemplos.

Sin embargo, la conceptualización e implementación de agentes de *software* puede ser por sí difícil. Así, en este trabajo de tesis se aborda el desarrollo de una biblioteca de rutinas (que puede servir de *framework* de desarrollo) que no sólo muestra la forma de implementar agentes de software sino que además, y de manera particular, aborda el tema de simulación de tráfico vehicular bajo este concepto.

El resultado de este trabajo plasma el proceso de diseño y desarrollo de la biblioteca mencionada con el objetivo de simulación en mente. Dicho resultado nos permite observar, y nos permite concluir, que el uso de agentes de software es una opción viable y con la madurez suficiente como para ser considerada una alternativa real en implementación de sistemas de software complejos para la simulación de tráfico vehicular.

ABSTRACT

Software agents and multi-agent systems represent a new way to get into the analysis, design and implementation of complex software systems. The agent-based approach offers a powerful set of techniques and tools with great potential to significantly improve the way people conceptualize and implement many types of software ranging from relatively small systems such as personalized email filters, to large and complex mission-critical systems such as those for air traffic control, to name a few.

However, the conceptualization and implementation of software agents can be inherently difficult. Thus, this thesis addresses the development of a library of routines (which can serve as a development framework) that not only shows how to implement software agents but also, and particularly, addresses simulation vehicular traffic under this concept.

This work describes the design and development process of this software library with the simulation objective in mind. This result allows us to see and conclude that the use of software agents is viable and mature enough to be considered as a real alternative for the implementation of complex software applications for the domain of traffic and transportation systems.

ÍNDICE GENERAL

RELACIÓN DE ILUSTRACIONES

RELACIÓN DE TABLAS

INTRODUCCIÓN

El campo de los agentes de *software*, es relativamente joven dentro del campo de las tecnologías de la información y las comunicaciones (TIC). Representa una fuente de generación de soluciones para un cúmulo de situaciones presentes en diversas disciplinas que pudieran parecer tan dispares como la educación, la salud, el comercio o los procesos industriales, y que se encuentran estrechamente relacionadas por problemáticas de la misma naturaleza: la búsqueda, recuperación, clasificación y organización de la información a lo largo de todo el proceso de negocio.

De manera cada vez más frecuente, se hace patente la necesidad de contar con aplicaciones flexibles, capaces de anticiparse y adaptarse a los requisitos cambiantes de los usuarios informáticos. Los agentes de *software* representan una solución para esta problemática, al definirse como entidades autónomas de software cuya característica principal es la habilidad para interactuar con su entorno.

Este trabajo de tesis se encuentra estructurado en 4 capítulos, a saber:

- El capítulo 1 realiza una revisión de la situación actual de los agentes de *software*: definiciones básicas, características, tipos y retos para su implementación en los sistemas actuales. También hace un breve recorrido sobre la plataforma de desarrollo de Microsoft conocida como Microsoft .Net.

- El capítulo 2 es una exploración más detallada de la temática de este trabajo de tesis. A lo largo de esta sección se describen las características deseables que deben ser consideradas para la creación de una biblioteca de clases (API). Adicionalmente, se detallan algunos de los puntos más importantes para la elaboración de simulaciones de tráfico y simulación basada en agentes.

- El capítulo 3. presenta de manera general la documentación del proceso de análisis, diseño y desarrollo de la biblioteca de clases para la simulación de tráfico. La metodología de desarrollo seleccionada para este proyecto es la de *Extreme Programming* por las bondades que ofrece en cuanto a simpleza y flexibilidad y a su naturaleza de desarrollo iterativo e incremental.

- El capítulo 4 nos muestra un caso de estudio donde se aplica la API elaborada y se realiza una simulación de tráfico basada en agentes. Con este proceso de implementación se da cumplimiento al objetivo del proyecto, a la vez que abre la puerta para futuras investigaciones que permitan evaluar el uso continuado de esta API, así como extender sus características.

DESCRIPCIÓN DEL PROBLEMA

Durante los últimos años, hemos sido testigos de un cambio progresivo en el mundo de la informática. Factores tales como el descenso en los costos del *hardware*, el mejoramiento en la durabilidad y fiabilidad de las redes, así como los avances en miniaturización y duración de la batería han permitido otorgar un mayor poder de cómputo a los dispositivos inalámbricos móviles.

Como consecuencia, hemos entrado de pleno en un proceso de "descentralización" de los servicios de cómputo y aplicaciones, obteniéndose así dos entornos con características muy peculiares:

- El primero, el mundo convencional de bases de datos y servidores conectados de manera estática que hemos utilizado durante mucho tiempo como plataforma para la creación y consumo de servicios de *software*.
- El segundo, un mundo distribuido y sumamente cambiante con clientes ligeros conectándose y desconectándose constantemente a distintas redes y consumiendo distintos servicios a través de internet.

Los consumidores de servicios de *software* de ambos entornos incluirán no sólo a usuarios humanos sino también a sistemas de *software* que fungirán como proveedores y consumidores de servicios. Esta misma dualidad de ambientes requerirá la utilización de "entes" que, en aras de establecer una conexión entre ambos entornos, sean capaces de:

- Representar por igual a consumidores y proveedores de servicios sin importar si estos son humanos o basados en *software* (por ejemplo, los motores de búsqueda y de metadatos).
- Establecer una capa de abstracción que permita consumir servicios de sistemas con código antiguo
- Encapsular protocolos de coordinación y compartición de información entre otros entes

Dentro de este contexto, los **agentes de software**, que de forma breve definiremos como programas de computadora que pueden actuar en representación de un usuario humano, una organización o un sistema (en *hardware* o *software*), representan una alternativa de solución para lograr conectar ambos entornos de forma eficiente.

Como ocurre con cualquier otra tecnología, existen beneficios inherentes tales como mayor productividad, confiabilidad, facilidad de uso y tolerancia a fallas, una reducción en los costos y en el nivel de complejidad del desarrollo, así como un incremento en la flexibilidad, por mencionar algunos. Sin embargo, existen dos razones principales que resaltan la importancia de la tecnología de agentes de *software* y que se mencionan a continuación.

Manejo de emergencias

Bajo situaciones de tensión extremas, las personas, organizaciones, o incluso los sistemas de cómputo pueden presentar inconsistencias o fallas en sus tareas habituales, lo que puede desencadenar en problemas mayores si no existe alguna intervención correctiva. En el caso de las personas, situaciones tales como enfermedad, fallecimientos de familiares o problemas personales lo pueden distraer de las tareas comunes cuyos resultados o productos, en caso de ser utilizados a su vez como insumos para tareas más críticas, pueden desencadenar en problemas de mayor índole. Normalmente, este tipo de tareas se pueden llevar a cabo sin necesidad aparente de un equipo sofisticado, por lo que bien podrían implementarse utilizando tecnología de agentes. Por tanto, no es una cuestión sobre si los agentes se requieren en todo momento, sino que pueden existir momentos críticos en los que resulte demasiado complicado establecer una acción correctiva específica.

Sistemas distribuidos, sistemas adaptativos complejos, cómputo distribuido

Los agentes de software presentan un paradigma mucho más apropiado para explotar el paralelismo y la dinámica existentes en las redes informáticas a gran escala. Las operaciones asíncronas y autónomas son características naturales de los agentes de *software* independientes,

por lo que se convierten en una opción ideal para explotar el potencial de la computación distribuida

En las últimas décadas se han presentado un número significativo de avances, tanto en el diseño como en la implementación, de agentes autónomos individuales, así como en la forma en la que interactúan entre ellos. Esta evolución ha sido de forma tal, que a pesar de ser un campo relativamente nuevo dentro de la Informática, ha encontrado un nicho dentro de los productos comerciales y enfocados a solucionar problemáticas de *software* presentes en las empresas actualmente.

Sin embargo, uno de estos conflictos que ocurren de forma más frecuente es que, a pesar del desarrollo de estándares y de mecanismos que permitan uniformizar los procesos de creación y comunicación de y entre agentes, a menudo el desarrollo de este tipo de aplicaciones es llevado a cabo a través de implementaciones realizadas *ex profeso*, sin seguir ninguna metodología, estándar o herramientas y que tienen un dominio de aplicación que dificulta el establecimiento de servicios de información compartidos para el intercambio de datos. Esto constituye un obstáculo importante que limita el uso extendido de los sistemas multi-agentes dentro de las organizaciones.

Actualmente existen en el mercado herramientas que buscan enfrentar dicha problemática. El ejemplo más representativo de lo anterior es la plataforma JADE, desarrollada por TILAB y cuyo propósito es proporcionar un entorno para el desarrollo de aplicaciones multi-agentes distribuidas basadas en una arquitectura de comunicación *peer to pee*r (entre pares). Esta plataforma, desarrollada en Java, es, al mismo tiempo, una biblioteca de clases para la creación de agentes y un contenedor que hace las veces de entorno para la comunicación y monitoreo de los mismos.

JUSTIFICACIÓN

Los agentes de *software* son, probablemente, una de las áreas de más rápido crecimiento en el campo de las tecnologías de la información (TI). Nwana Hyacinth [NW1] asegura que este crecimiento se debe a la gran demanda de herramientas necesarias para administrar el crecimiento explosivo de la información que vivimos actualmente y que seguiremos experimentando en adelante. Bajo este escenario, los agentes de *software* juegan un rol preponderante en la gestión, manipulación y cotejo de información proveniente de varias fuentes distribuidas.

Por otro lado, L. Leal [LM2] sugiere que el auge que vive actualmente la teoría de agentes se debe principalmente a dos factores:

- La popularidad de las redes de cómputo que han permitido pasar de programas que viven en una sola computadora a sistemas que corren en ambientes distribuidos, paralelos, abiertos y dinámicos
- El crecimiento en el nivel de complejidad de las tareas que pueden ser automatizadas y delegadas sin requerir de supervisión humana.

La consultora Gartner [FE1], por su parte, incluyó a los agentes inteligentes dentro de las tecnologías emergentes más mencionadas en el año 1995, tal como se puede apreciar en la figura siguiente:

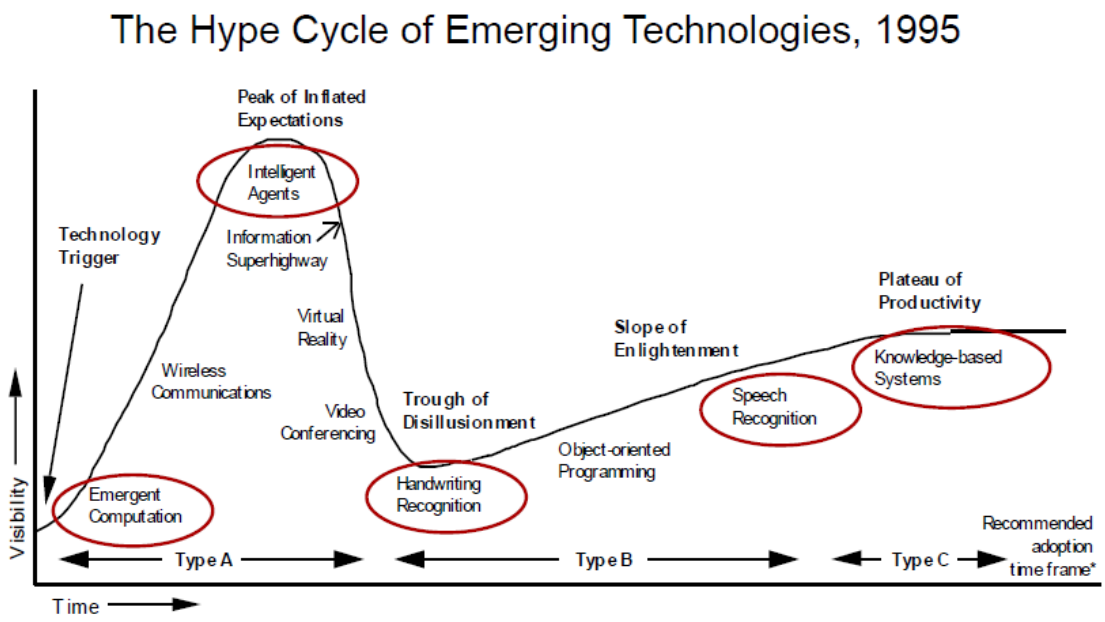


Figura 1.Ciclo de tecnologías emergentes más populares. Extraído de [FE1]

Actualmente, los agentes de software encuentran cabida en una gran diversidad de escenarios que van desde el manejo de información personal, comercio electrónico, diseño de interfaces, juegos de computadora, y el control de procesos comerciales e industriales.

El campo de los agentes de *software* es el resultado de la fusión de conceptos originados en diversas áreas de estudio, tales como la inteligencia artificial, el cómputo distribuido, la ingeniería de *software* o la orientación a objetos, sólo por mencionar algunas, y se erigen como respuesta al creciente nivel de complejidad existente actualmente en las tareas de recuperación, recopilación y organización de información para la resolución de problemas.

De manera particular, la simulación basada en agentes ha cobrado especial importancia como un nuevo método para la investigación social [GV1]. Uno de los grandes atractivos de la simulación basada en agentes es que permite explicar cómo emergen estructuras sociales a partir de las acciones individuales, y a su vez cómo las creencias, deseos y oportunidades de los individuos son afectadas por dichas estructuras, integrando de este modo los niveles macro y micro de la realidad social. En este sentido, la simulación de tráfico vehicular se ve altamente beneficiada por este nuevo enfoque basado en agentes.

Al respecto, los doctores Bo y Harry Chen [CH2] coinciden al afirmar que el paradigma de los agentes, al ser una tecnología de gran utilidad para la implementación de sistemas distribuidos a gran escala, se constituye como una excelente herramienta para el dominio de sistemas de transporte y tráfico, ya que éstos en esencia se encuentran inmersos en ambientes distribuidos geográficamente con características cambiantes.

El paradigma basado en agentes propone un mayor nivel de abstracción en el diseño de programas al dotar a las entidades de *software* con atributos característicos de las personas, es decir, les otorga tres características importantes:

- **Autonomía**. Los agentes tienen un cierto grado de control sobre sus propias acciones. Son capaces de elegir un curso de acción dependiendo de circunstancias específicas.
- **Inteligencia**. Los agentes no sólo responden a eventos externos sino que cuentan con un comportamiento intencionado a cumplir con un objetivo y, cuando es apropiado, pueden tomar la iniciativa.
- **Interacción social**. Los agentes son capaces y, de hecho, necesitan interactuar con otros agentes para cumplir con sus objetivos. Con ello se permite que el sistema a su vez cumpla sus objetivos también.

Estos discriminantes permiten centrarse en la solución del problema de forma más natural y más coherente con los procesos ya existentes en la organización, relegando a un segundo término los detalles técnicos de la implementación.

ALCANCES Y LIMITACIONES

El campo de estudio de los agentes de software es bastante extenso y cada día se enriquece con las nuevas aportaciones de vanguardia, en lo que se conoce como el **estado del arte**. Al ser éste campo un punto común para diversas áreas de estudio, los avances presentes en cada una de ellas repercuten casi de inmediato en la forma en que se conciben y operan los agentes de *software*.

Tópicos tales como protocolos de comunicación en red, lenguajes de comunicación entre agentes, servicios *web*, tecnologías de desarrollo de *software* y estándares para el intercambio de información, por mencionar algunos, encuentran cabida en este tema y amplían los horizontes y capacidades del mismo. Incluso los avances en áreas que no guardan relación directa, como el desarrollo de cómputo en la nube, los servicios *web* o el desarrollo de nuevas tendencias en el campo de la ingeniería de *software* tales como la programación orientada a aspectos, repercuten en la forma en que se conciben y trabajan los agentes de *software*.

Un estudio detallado de dichas tecnologías existentes se encuentra fuera de los alcances de este proyecto, por lo que la propuesta de solución se limitará a la construcción de una plataforma funcional enfocada a la simulación de tráfico vehicular con condiciones controladas

Para los fines de este trabajo, se considerarán las siguientes limitaciones

- No se realizará una revisión exhaustiva de toda la tipología de agentes existente en la literatura.
- La biblioteca de clases generada no estará planeada para representar a todos los tipos de agentes existentes en la literatura relacionada al tema.
- La biblioteca de agentes se enfocará única y exclusivamente a la simulación de tráfico vehicular.
- La biblioteca de agentes no estará optimizada para su utilización en aplicaciones de misión crítica con alto nivel de rendimiento

RESULTADOS ESPERADOS

Al concluir este trabajo de tesis, se espera obtener los siguientes resultados:

- Una biblioteca de clases (API) que sirva de soporte al desarrollo de aplicaciones multi-agentes enfocadas a la simulación de tráfico vehicular
- Una aplicación que permita observar una implementación de un escenario de simulación construido utilizando la biblioteca de clases
- Casos de estudio con escenarios de simulación construidos utilizando la API
- Un documento técnico descriptivo del diseño de la biblioteca de clases especificando de manera general los métodos más significativos contenidos en las clases de dicha biblioteca

Capítulo 1

Panorama general de los agentes de software

1. Introducción al capítulo

Pocos acontecimientos han marcado de manera tan notable el progreso de la civilización como lo fue el desarrollo de las computadoras. Su relativamente rápida transición desde los enormes y sofisticados laboratorios de universidades y empresas hacia los hogares de las personas comunes, aunado al acelerado ritmo de evolución que nos ha dado acceso a equipos con mayores capacidades a un costo muchísimo menor al de sus antecesores, han sido factores clave que han propagado su uso en prácticamente todas las actividades humanas.

La misma necesidad, inherente en el ser humano, de establecer nexos de comunicación y colaboración entre pares, motivó la creación de redes de computadoras que, en última instancia, permitieron desarrollar lo que hoy conocemos como Internet. Éste representó un salto cuántico que terminó por situarnos en lo que el sociólogo Manuel Castells [MC 2] denominó, en sus escritos durante la década de los 90, como la **Era de la Información**.

La **Era de la Información** se caracteriza por la habilidad que tienen los individuos de transferir información de manera libre, así como de obtener acceso casi instantáneo a conocimientos que hubieran resultado de difícil o imposible acceso en épocas anteriores. Esta particularidad sirve como base para el concepto de revolución digital, utilizada para representar el cambio de paradigma de la economía basada en la industria tradicional, a la economía basada en la manipulación de información.

La era de la información presupone la creación de un nuevo escenario que otorga un nivel de influencia al individuo, al crear organizaciones que crecen de forma horizontal (redes) en lugar de la tradicional forma vertical (jerarquías). En este nuevo esquema la materia prima son los datos, la información, y el conocimiento que se forma con ellos.

Existen algunas implicaciones que es necesario considerar. La primera de ellas, es que este nuevo cambio de paradigma representa un aumento exponencial en las actividades relacionadas con la demanda y el suministro de información. Los avances tecnológicos actuales permiten obtener acceso a una mayor cantidad de canales de suministro de información en comparación con épocas anteriores. La información sobre un tema o sobre un evento en específico, que hasta hace algunas décadas nos llegaba casi de forma impresa, hoy se nos puede presentar, además de impresa, a través de boletines electrónicos, audio, video, mensajes de texto o mensajes de redes sociales, por mencionar algunas. El aumento en el número de canales conlleva un segundo cambio: la frecuencia de actualización. En este momento, es posible, desde cualquier parte del mundo, acceder a la información más reciente sobre cualquier campo del conocimiento humano, y obtener información sobre los descubrimientos y eventos más importantes instantáneamente o con pocos minutos de retraso.

Un tercer cambio importante es el origen de la información. En el pasado, el manejo de la información era controlado por un relativamente pequeño grupo de proveedores de información fácilmente identificable (editoriales, periódicos, empresas de medios de comunicación); actualmente, cualquier persona con acceso a Internet es capaz de publicar mensajes y generar conocimiento accesible a millones de personas, lo que ha provocado que dicho grupo de proveedores de información haya crecido exponencialmente.

Este panorama representa un nuevo y cada vez más grande nivel de complejidad para el manejo de información, y nos lleva a definir nuevos retos para poder utilizar todo el caudal de conocimiento disponible en la resolución de problemas para nuestro beneficio.

Desde la perspectiva del desarrollo de software, para afrontar dichos retos, hemos sido testigos de una serie de cambios de paradigma que buscan elevar el nivel de abstracción en el desarrollo de sistemas de tal manera que sea necesario centrarse cada vez más en el problema en sí y menos en

la tecnología necesaria para ello. García Alonso [GA1] sugiere que esta evolución de manera similar a la que puede apreciarse en la figura siguiente:

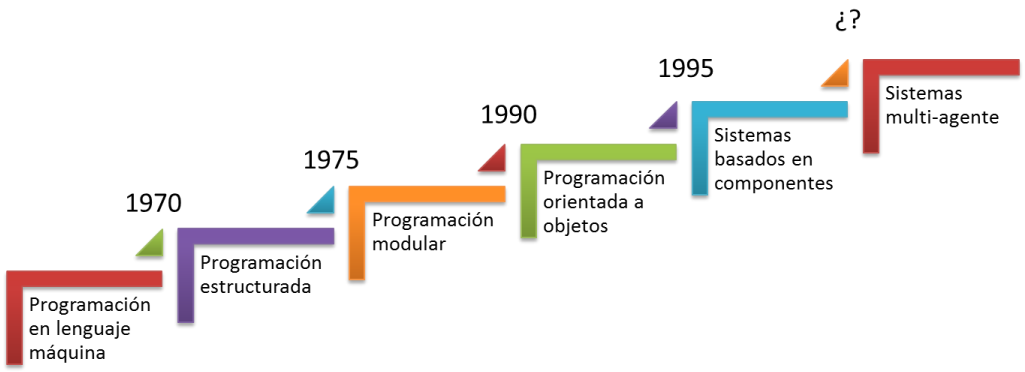


Figura 2. Cambios en el paradigma del desarrollo de software
(Elaboración propia adaptada de [GA1])

En este sentido, el paradigma del desarrollo de aplicaciones basadas en agentes pretende modelar entidades de software otorgándoles características similares a las de una "persona", de tal manera que la resolución de un determinado problema tenga un alto nivel de similitud con su solución en el mundo real.

La intención de este trabajo es continuar con los esfuerzos dirigidos al cambio de este nuevo paradigma de diseño basado en agentes a través de la generación de una herramienta que facilite la construcción de aplicaciones multi-agente.

1.2. Agentes de software

Los agentes de software, así como los sistemas multi-agentes, constituyen una nueva manera de analizar, diseñar e implementar complejas aplicaciones informáticas y desarrollos de sistemas de información. El enfoque basado en agentes ofrece un amplio repertorio de recursos y técnicas cuyo potencial puede incrementar, de forma considerable, la forma en la que las personas conciben e implementan distintos tipos de *software*.

Los agentes son utilizados en una amplia variedad de escenarios, que pueden ir desde tareas relativamente pequeñas como la personalización de filtros de correo electrónico personal, hasta el desarrollo de aplicaciones de misión crítica complejos como lo es el control de tráfico aéreo. Aunque en primera instancia pudiera parecer que aplicaciones como las mencionadas tienen pocos aspectos en común, en realidad ambas hacen uso del concepto de agente.

El término agente de software surge a partir de los Sistemas Multi-Agente (Multi Agent Systems, MAS), que es un concepto utilizado para hacer referencia a todos aquellos tipos de sistemas compuestos por múltiples componentes semi-autónomos.

De forma tradicional, el estudio de los sistemas multi-agente forma parte del campo de la Inteligencia Artificial Distribuida, que puede dividirse en dos áreas principales: la resolución de problemas de forma distribuida (distributed problem solving, DSP) y la inteligencia artificial distribuida.

La resolución de problemas de forma distribuida se refiere al análisis de la manera en que un problema en particular puede resolverse utilizando para ello a un conjunto de módulos (nodos) que cooperan entre sí compartiendo información e intercambiando conocimiento sobre dicho problema y la evolución de sus posibles soluciones.

En un sistema DSP, todas las estrategias de información son incorporadas como una parte integral del sistema. En contraste, la investigación en sistemas multi-agente se centra en el comportamiento de una colección de agentes autónomos ya existentes dedicados a solucionar un problema determinado.

Un sistema multi-agente puede definirse, entonces, como una red de solucionadores de problemas débilmente acoplada que trabajan de manera conjunta para resolver problemas que se encuentran más allá de sus capacidades individuales o el conocimiento de cada solucionador por separada. Dichos solucionadores de problemas (los agentes) son autónomos y pueden ser heterogéneos en su naturaleza.

1.2.1. ¿Qué es un agente?

Si bien los agentes de *software* tienen aplicación en una amplia variedad de disciplinas, una de las consecuencias resultantes es la falta de una definición "generalmente" aceptada sobre lo que es un agente. Esto puede deberse, principalmente, a que la misma palabra "agente" es un término general utilizado en un conjunto de campos de investigación heterogéneo y extenso. A continuación se presentan algunas de las definiciones más conocidas dentro del campo de los agentes de *software*.

- Hyacinth S. Nwana (1996) [NW1] define a un agente como "un componente de *software* y/o *hardware* que tiene la capacidad de actuar en nombre del usuario con el fin de realizar una tarea".

- Wooldridge y Jennings (1996) [WJ1] lo definen como un "programa auto-contenido capaz de controlar su proceso de toma de decisiones y de actuar, basado en su percepción del ambiente, en búsqueda de uno o más objetivos".

- Stan Franklin (1996) [FG1] define a un agente autónomo como "un sistema situado dentro y como parte de un entorno, capaz de percibir y actuar sobre él durante el tiempo, en búsqueda de sus propios objetivos definidos para llevar a cabo lo que percibe en el futuro".

- Russell y Norvig, en su libro "*Artificial Intelligence: A Modern Approach*" (2010) [RN1], mencionan que "un agente puede ser visto como algo que percibe su ambiente a través de sensores y actúa en este a través de efectores". Agregan, además, que "la noción de un agente está destinada a ser una herramienta para el análisis de sistemas, no una caracterización absoluta que divide el mundo en los agentes y no agentes".

1.2.2. Características

Para poder comprender lo que es un agente a partir de las definiciones listadas anteriormente, es necesario destacar sus elementos comunes. Si las revisamos con atención notaremos que todas ellas hacen referencia a un número de características que, en su conjunto, pueden proporcionar

una idea más clara de lo que es un agente. En la medida en que se agreguen o detallen dichas características en referencia a una entidad determinada, se obtendrá una noción más clara del tipo de agente o la subclase de agente en cuestión.

Un elemento en común en todas las definiciones anteriores, es la existencia del concepto de entorno. Todos los agentes autónomos se encuentran situados en un medio ambiente. Si se modifican las características de dicho entorno, es posible que el agente pierda su razón de ser. Por ejemplo, un agente dedicado al control de tráfico aéreo carecerá de utilidad si se le traslada a un entorno distinto, como podría ser un control de estacionamientos.

De acuerdo con los requerimientos de un problema en particular, un agente podría poseer, en mayor o menor grado, atributos como los siguientes:

- **Reactividad**: un agente percibe el entorno y responde en un tiempo razonable a los cambios en él.

- Un agente debe tener la **capacidad de percibir el estado del entorno** dentro del cual se encuentra inmerso, y a partir de la información obtenida, responder de manera adecuada a los cambios que se presenten en el medio ambiente. Las respuestas a dicho entorno pueden modificar el estado del mismo.

- **Autonomía**: un agente es capaz de actuar sin la intervención directa de una entidad externa, pudiendo ser ésta una persona u otro agente. Además, debe poder controlar sus propias acciones y estado interno.

- **Orientación a objetivos o proactividad**: un agente no sólo debe actuar en respuesta a los cambios detectados en el entorno sino que además debe actuar en función de los objetivos para los cual fue diseñado y las tareas que le fueron delegadas en cada momento. Un agente busca de forma permanente cumplir las metas para las cuales fue creado. En un modelo basado en agentes, el proceso es auto-motivado, es decir, el agente busca satisfacer cierto estado interno con mínima intervención humana.

- **Continuidad temporal**: un agente es un proceso que se ejecuta de manera continua, que cuenta con persistencia de su identidad y estado a través del tiempo. Un agente, a diferencia de un programa convencional, que tiene un inicio y un fin definidos, debe

ejecutarse hasta que haya cumplido con el conjunto de objetivos solicitados, o bien, mientras su ciclo de vida continúe y el usuario no desee detenerlo. Esta característica es la que le da "vida" al agente y que posibilita que se mantenga alerta a una solicitud o a algún cambio en el medio. El ciclo de vida de un agente depende de sus características, de las tareas que realice y de los deseos de su usuario en cuanto al tiempo durante el cual el agente debe ejecutarse.

- **Habilidad social**: un agente debe tener habilidad para interactuar con otros agentes o incluso con alguna persona, con el fin de solicitar información o bien para presentar los resultados de la ejecución de las tareas encomendadas. El mecanismo a través del cual se establecerá la comunicación dependerá del tipo de entidad con la que se comunique (humanos u otros agentes). En cualquier caso, es necesario establecer un protocolo de intercambio de información entre ambas partes. Un agente debe contar con una interfaz que le permita comunicarse con las demás entidades.

- **Adaptabilidad o aprendizaje**: Un agente es capaz de modificar su comportamiento basado en sus experiencias previas

- **Movilidad**: un agente es capaz de migrar de forma auto-dirigida desde una plataforma a otra, en busca de los recursos que le permitan cumplir con sus objetivos. Esto se refiere a que, en un determinado momento, el agente detiene su ejecución, almacena su estado interno y se dirige a otro sitio dentro de una red de computadoras para luego continuar con su ejecución en la nueva ubicación. La movilidad no es una propiedad indispensable para un agente sino que modifica la forma por la cual el agente cumple con sus objetivos, en este caso recurriendo a los recursos que puede ofrecer una red de computadoras. Aporta una nueva forma de computación distribuida

- **Personalidad**: Mientras que algunos investigadores en el campo de los agentes de software se muestran satisfechos con crear agentes que sean "útiles" únicamente para los propósitos requeridos, algunos otros están en la búsqueda de un objetivo más ambicioso: la creación de agentes con una interfaz de usuario bastante detallado y que sean capaces de proyectar la ilusión de ser conscientes, bien intencionados, y capaces de simular emociones y manejar interacciones sociales significativas. Un buen ejemplo de esta

característica es el proyecto Persona [BA1], que tenía como objetivo el crear prototipos de interfaces para asistentes por computadora que contaran con características antropomórficas como mantener un diálogo en una conversación y que sean capaces de interactuar con el usuario para aceptar asignaciones de tareas y reportar resultados. Dichos asistentes, para ser eficaces, deberían ser capaces de realizar actividades como la negociación, la realización de preguntas con el fin de clarificar objetivos, entender cómo y cuándo era apropiado interrumpir al usuario para reportar avances o solicitar información y reconocer los impactos sociales y emocionales de esta interacción.

Un agente generalmente cumple con las primeras cuatro características. En la medida en que se van añadiendo propiedades se obtienen clases más especializadas de agentes.

1.2.3. Tipología de agentes

Con el fin de proporcionar una manera más sencilla de caracterizar el espacio de tipos de agentes que resultaría si se intentara describir cada combinación de atributos posibles, se han propuesto varios esquemas de clasificación.

Stan Franklin (1996) [FG1] propone una clasificación basada en un modelo de taxonomía biológica en forma de árbol en donde las “criaturas vivientes” se encuentran en la raíz y las especies individuales se ubican en las hojas:

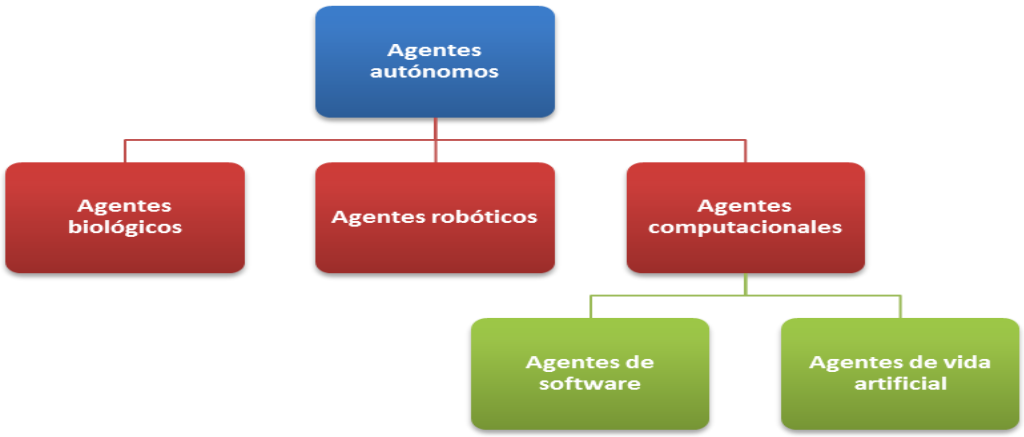


Figura 3. Taxonomía Natural de los agentes
(Elaboración propia, adaptado de [FG1])

Gilbert (1995) [GD1] en el documento denominado *IBM Intelligent Agent Strategy* describe a los agentes inteligentes en términos de un espacio definido por tres dimensiones: agencia, inteligencia y movilidad.

De acuerdo a Gilbert, la **agencia** es el nivel de autonomía y autoridad de un agente, y puede ser medido, al menos de forma cualitativa, por la naturaleza de la interacción entre el agente y otras entidades en el sistema. Como mínimo, un agente debe ejecutarse de forma asíncrona. El nivel de agencia se incrementa si el agente representa al usuario de alguna manera. La **inteligencia** es el nivel de razonamiento y de aprendizaje obtenido: es la habilidad del agente para aceptar las definiciones de los objetivos del usuario y llevar a cabo las tareas delegadas a él. La **movilidad** es el nivel en el que los agentes viajan, ya sea a través de una red o bien a través de los componentes dentro de un mismo computador.

Nwana (1996) [NW1] propone una clasificación de los agentes en diferentes clases, basado en cinco dimensiones de clasificación.

- En primer lugar, los agentes pueden ser clasificados por su movilidad, esto es, por su habilidad para moverse a través de alguna red. De acuerdo a este criterio, se clasifican en móviles y estáticos.

- En segundo lugar, pueden ser clasificados entre deliberativos y reactivos. Los agentes deliberativos derivan del paradigma del pensamiento deliberativo, es decir, que cuentan con un modelo de razonamiento simbólico interno. En contraste, los agentes reactivos no cuentan con este modelo de razonamiento, y en su lugar actúan usando un comportamiento del tipo estímulo/respuesta para interactuar con el estado actual del entorno en el que se encuentran.

- En tercer lugar, los agentes pueden ser clasificados de acuerdo a la exhibición de tres atributos primarios ideales: **autonomía**, **aprendizaje** y **cooperación**.

- En cuarto lugar, los agentes a veces pueden ser clasificados de acuerdo al rol que desempeñan; por ejemplo, los agentes de información, que se encargan de administrar el amplio caudal de información que se obtiene sobre un tópico en particular a través de los motores de búsqueda en Internet.

- Finalmente, se incluye una categoría de agentes híbridos que combinan dos o más de las dimensiones de clasificación en un mismo agente.

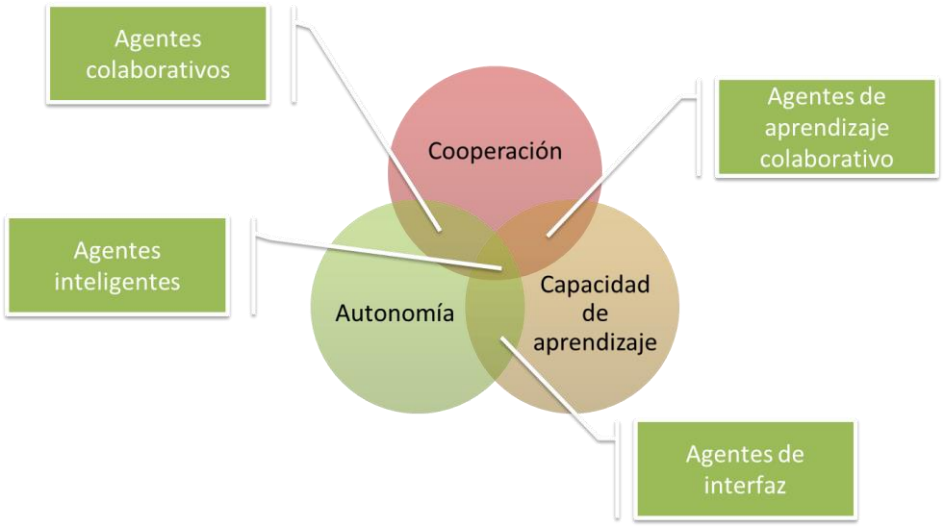


Figura 4. Vista parcial de una tipología de agentes [NW1]

A partir de estas dimensiones, Nwana propone una tipología de siete tipos de agentes:

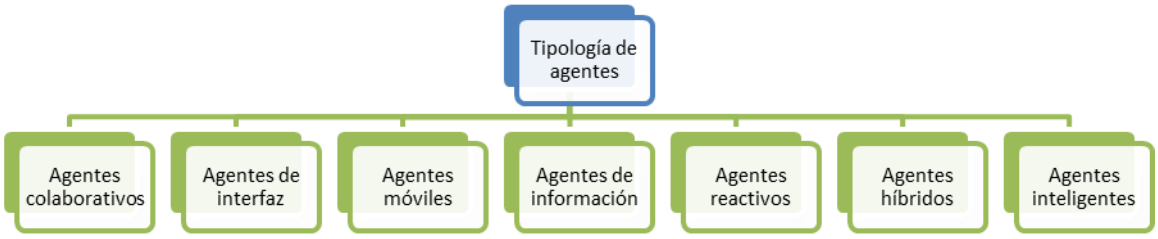


Figura 5. Clasificación de agentes de software [NW1]

Agentes colaborativos: este tipo de agentes hacen uso intensivo de la autonomía y la cooperación con el fin de realizar tareas para el usuario. Pueden aprender, aunque este no es un rasgo característico de su operación. Pueden negociar con otros agentes en busca de acuerdos con el fin de alcanzar sus objetivos y lograr un mejor nivel de coordinación.

De forma general, las características principales de este tipo de agentes son la autonomía, la habilidad social, la capacidad de respuesta y la proactividad. Por tanto, deben, o al menos debieran, ser capaces de actuar de forma racional y autónoma en ambientes multi-agente con restricciones de tiempo. Tienden a ser estáticos y de bajo nivel de granularidad. Típicamente, la mayoría de los agentes colaborativos implementados actualmente no llevan a cabo ningún proceso de aprendizaje complejo, aunque pueden realizar procesos de memorización o aprendizaje limitado.

Agentes de interfaz: Este tipo de agentes resaltan la autonomía y el aprendizaje con el fin de realizar tareas para el usuario. Un ejemplo de metáfora típica que se sugiere para ejemplificar este tipo de agentes, es el de un “asistente personal” que colabora con el usuario en el mismo entorno de trabajo. Es preciso destacar la sutil diferencia entre colaborar con el usuario y colaborar con

otros agentes, como es el caso de los agentes colaborativos. Colaborar con el usuario podría no requerir de un lenguaje de comunicación especial como el que se utiliza entre agentes.

Esencialmente, los agentes de interfaz proveen asistencia y soporte al usuario. Generalmente, a un usuario que se encuentra aprendiendo una aplicación en particular como una hoja de cálculo o un sistema operativo. En este escenario, el agente observa y monitorea las acciones realizadas por el usuario en la interfaz, aprende nuevos atajos y sugiere mejores maneras de completar una tarea.

Agentes móviles: son procesos de software capaces de trasladarse a través de una red de computadoras (como Internet) interactuando con agentes externos para recopilar información en nombre del usuario y, una vez completadas las tareas definidas por el usuario, regresar al entorno del cual salieron originalmente. Estas tareas pueden variar desde la reservación de vuelos hasta la administración de una red de telecomunicaciones. Sin embargo, la movilidad no es una característica necesaria ni suficiente para definir a un proceso como agente. Los agentes móviles lo son porque son autónomos y pueden colaborar entre sí, aunque de manera diferente a los agentes colaborativos. Por ejemplo, un agente podría comunicarse y cooperar con otros dando a conocer la ubicación de algunos de sus objetos y métodos internos a otros agentes. De esta forma, intercambia información con otros agentes sin dar a conocer la totalidad de su información.

Agentes de información/Internet: los agentes de información son la respuesta a la enorme demanda de herramientas para administrar el crecimiento explosivo de información que experimenta la sociedad actualmente. Los agentes de información son las entidades encargadas de administrar, manipular y seleccionar información proveniente de diversas fuentes distribuidas.

Agentes reactivos: representan una categoría especial de agentes que no poseen un modelo simbólico internos en sus entornos; en su lugar, contestan bajo un esquema estímulo/respuesta a los estados presentes del entorno en el que se encuentran inmersos. Uno de los puntos más importantes sobre los agentes reactivos es el hecho de que son relativamente simples y que

interactúan con otros agentes a través de procesos básicos; sin embargo, se pueden apreciar patrones complejos de comportamiento que emergen de las interacciones entre este tipo de agentes cuando se observan de forma global.

Un agente reactivo es visto comúnmente como una colección de módulos que operan de forma autónoma y que son responsables de tareas específicas (por ejemplo, sensores, motores, unidades de cálculo, etc.). La comunicación entre dichos módulos es mínima y a un nivel bastante primitivo. No existe un modelo global dentro de cada uno de dichos agentes, por lo que emerge un comportamiento a nivel global.

Los agentes reactivos tienden a operar en representaciones cercanas a datos del sensor en bruto, en contraste con las representaciones simbólicas de alto nivel que abundan en los tipos de agentes discutidos anteriormente.

Agentes híbridos: Cada uno de los tipos de agentes que se han discutido hasta el momento (colaborativos, de interfaz, móviles, de información y reactivos) cuentan con un conjunto de fortalezas y deficiencias que los hacen más o menos apropiados para cada situación en particular. Una manera de maximizar los beneficios de cada tipo de agente, es adoptar un enfoque híbrido. Bajo esta filosofía, un agente híbrido se refiere a aquel cuya constitución es una combinación de dos o más filosofías de agentes.

La hipótesis principal que sustenta el uso de los agentes híbridos es la creencia de que, para un tipo específico de aplicación, los beneficios obtenidos de contar con un conjunto de filosofías dentro de un mismo tipo de agente son mayores que aquellos obtenidos del mismo agente si éste está basado completamente en una única filosofía.

Agentes inteligentes: Aunque Nwana no describe este tipo de agentes por considerarlos más una aspiración de los investigadores en el campo de los agentes que una realidad, Gilbert (1995) considera a los agentes inteligentes como “una pieza de software que ejecuta una tarea dada utilizando información recolectada del ambiente, para actuar de manera apropiada hasta completar la tarea de manera exitosa. El software debe ser capaz de auto ajustarse basándose en

los cambios que ocurren en su ambiente de forma tal que un cambio en las circunstancias producirá un resultado esperado".

1.2.4. Aplicaciones

La tecnología basada en agentes rápidamente comenzó a salir de las universidades y los laboratorios de investigación, y encontró un nicho resolviendo problemas reales en una gran variedad de escenarios industriales y comerciales. Actualmente, las principales áreas en las que se han documentado aplicaciones basadas en agentes son la manufactura, el control de procesos, los sistemas de telecomunicaciones, el control de tráfico aéreo, la gestión de tráfico y de transportes, la recopilación y filtrado de información, el comercio electrónico, la gestión de procesos de negocio, el entretenimiento y la salud. A continuación se mencionan con mayor detalle algunas de las aplicaciones principales en dichas áreas.

Manufactura

H. Van Dyke Panurak (1987) [VP1] documenta el sistema YAMS (*Yet Another Manufacturing System*) que aplica el "Protocolo de Contrato de Red" para el control de manufactura. El objetivo de dicho sistema es administrar, de forma eficiente, la producción de una empresa de manufactura que opera bajo el modelo de células de trabajo especializadas distribuidas en diversas fábricas. El proceso se encuentra definidos por diversos parámetros modificables, como los productos que deben ser manufacturados, los recursos disponibles, las restricciones de tiempo, etc. Para cumplir con su objetivo, YAMS utiliza un enfoque basado en sistemas multi-agente, donde cada fábrica y cada componente de la fábrica es representado por un agente. Cada agente posee una colección de planes que representa sus capacidades. El protocolo de contrato de red permite que las tareas sean delegadas a una fábrica en específico, y de dicha fábrica a una célula en particular.

Control de proceso

El control de procesos es una aplicación natural de los agentes, dado que los controladores de procesos son en realidad sistemas reactivos autónomos. Una de las aplicaciones más conocidas es la plataforma de software ARCHON, que ha sido aplicada a diversos procesos de control, que van desde la gestión de transportes eléctricos (que opera actualmente en España) y el control de la aceleración de partículas.

Telecomunicaciones

Los sistemas de telecomunicaciones se componen de extensas redes distribuidas de componentes interconectados que necesitan ser monitoreados y administrados en tiempo real. Las compañías de telecomunicaciones buscan obtener ventajas competitivas a través del ofrecimiento de servicios más confiables y de mejor calidad, por lo que recurren a la implementación de diversas aplicaciones de software, incluidas aquellas basadas en agentes. En particular, existe información documentada sobre aplicaciones encargadas del control de la red, de la transmisión e intercambio de información en la red, y de la administración de servicios.

Control de tráfico aéreo

Ljunberg y Lucas (1992) [LL1] presentaron informes de pruebas, en el aeropuerto de Sydney, Australia, de un sofisticado sistema para el control de tráfico aéreo denominado OASIS. En este sistema, los agentes son utilizados para representar aeronaves y diversos sistemas de control de tráfico en operación. En este escenario, la metáfora del agente ofrece una manera natural de modelar componentes autónomos en el mundo real. En cuanto una aeronave ingresa al espacio aéreo de Sydney, le es asignado un agente, que es *instanciado* con la información y objetivos de la aeronave real. Los agentes de control de tráfico aéreo son responsables de la administración del sistema.

Sistemas de transporte

La gestión de tráfico y transporte es un escenario idóneo para el enfoque basado en agentes dada su inherente distribución geográfica. Burmeister (1998) [BU1] documentó un sistema multi-agente diseñado para implementar una aplicación de distribución de transportes. En esta aplicación existen dos tipos de agentes: unos que representan a los clientes que requieren u ofrecen servicios de transporte, y otra que representa las estaciones donde los clientes se reúnen para utilizar los servicios de transporte. Los agentes "clientes" informan a las estaciones sobre sus necesidades (cuándo y hacia dónde desean ir) y los agentes "estaciones" determinan si sus peticiones pueden ser atendidas y, de ser así, qué transporte deben reservar.

Gestión de información

La enorme riqueza y variedad de información disponible vía Internet pone de manifiesto una necesidad de herramientas para la gestión de información que den solución al fenómeno conocido como sobrecarga de información. Existen dos variantes de este problema en particular

La filtración de información: cada día, nos encontramos expuestos a enormes cantidades de información a través de Internet, o del correo electrónico, de la cual tan sólo un pequeño porcentaje podría considerarse como relevante. Es necesario establecer un mecanismo que permita enfocarse en la información importante que realmente nos interesa.

Recopilación de información: el volumen de información disponible dificulta que encontremos los datos que responden las preguntas específicas que nos formulamos. Es necesario, entonces, definir mecanismos que nos permitan recolectar información que satisfaga nuestras necesidades, sin importar si es recopilada de diferentes sitios.

P. Maes (1994) [MA1] documenta información sobre Maxims, un agente de filtrado de correo electrónico que aprende a asignar prioridades, eliminar, reenviar, ordenar y archivar mensajes de correo en nombre del usuario. Trabaja monitoreando la actividad del usuario y tomando cada acción que ejecuta como una lección. Maxims realiza de manera frecuente predicciones internas sobre lo que el usuario hará con un mensaje.

Si estas predicciones se vuelven inexactas, entonces el agente las conserva para sí mismo. Sin embargo, cuando alcanza un nivel de precisión aceptable en sus predicciones, comienza a ofrecer sugerencias sobre lo que debería hacer el usuario.

Comercio electrónico

En la actualidad, el correo electrónico está basado, casi en su totalidad, por interacciones humanas. Son las personas las que deciden cuándo y qué comprar, y a qué precio; sin embargo, en principio, no existe una razón por la cual este tipo de operaciones no puedan ser automatizadas, es decir, que este tipo de decisiones sean trasladadas a agentes.

A. Chávez y P. Maes (1996) [MP1] documentan el caso de un "mercado electrónico" denominado Kasbah, en el cual se crean agentes "compradores" y "vendedores" para cada producto ofrecido.

Las transacciones comerciales son el resultado de las interacciones entre agentes.

Gestión de procesos de negocio

Un escenario común en las organizaciones es la toma de decisiones basada en juicios realizados a partir de la información generada por varios departamentos. De forma ideal, la información relevante que es necesaria para tomar dichas decisiones debe ser recopilada y analizada antes de que la decisión sea tomada; sin embargo, la obtención de información actualizada y consistente suele ser una tarea compleja y que consume una gran cantidad de tiempo. Por este motivo, las organizaciones han desarrollado una gran variedad de sistemas de TI para asistirlos en los procesos de negocio.

Jennings (1996) [JE1] presenta el proyecto ADEPT, que ataca este problema considerando los procesos de negocios como una comunidad de agentes proveedores de servicios con capacidad de negociación. Cada agente representa un área distinta en la organización capaz de proveer uno o más servicios. Los agentes que requieren el servicio de otros, inician negociaciones en términos de precio, tiempo y calidad en búsqueda de lograr acuerdos.

Entretenimiento

La industria del entretenimiento rara vez es considerada de forma seria por los investigadores en computación e informática. Son percibidas, de forma general, como una aplicación "periférica" a las aplicaciones "serias" en computación. A pesar de ello, en este campo, los agentes son utilizados de forma frecuente en la creación de juegos de computadoras y aplicaciones de realidad virtual, en las que a menudo es necesaria la implementación de entes animados semiautónomos que pueden interactuar en tiempo real.

Aplicaciones médicas

La informática médica es una de las áreas con mayor crecimiento dentro del campo de la informática. Dos de las aplicaciones más comunes son el monitoreo y el cuidado de pacientes.

El sistema GUARDIAN, presentado por B. Hayes-Roth (1991) [HA1] fue diseñado para colaborar en el manejo de pacientes en una unidad de cuidados intensivos. El sistema distribuye las funciones de monitoreo entre un número determinado de agentes de tres tipos diferentes: agentes de percepción/acción, responsables de la interfaz entre GUARDIAN y el mundo exterior, y que interpretan los datos para traducirlos a una forma simbólica. Agentes de razonamiento, responsables de la toma de decisiones, y agentes de control, encargados de la coordinación entre agentes.

1.3. Definiciones

Actualmente, se tienen diversas definiciones para la Interfaz de Programación de Aplicaciones (API), el fin de este trabajo es utilizar el término correcto y no dejarse llevar por una definición poco general.

Por ello, es necesario explicar el concepto a utilizar y en el cual nos basaremos. Para efectos de este texto, se utilizarán las siguientes definiciones:

1.3.1 Framework

En el ámbito del desarrollo de *software*, un ***framework*** (también conocido como infraestructura digital, aunque este término es poco utilizado) se refiere a una abstracción en la que un *software* proporciona funcionalidad genérica que puede ser complementada o sustituida por código escrito por el usuario con el fin de crear una aplicación de propósito específico. Un *framework* de *software* es una plataforma reutilizable que puede incluir programas de apoyo (utilerías), compiladores, bibliotecas de código, o una API para la creación de aplicaciones, productos y soluciones [WK3],

1.3.2 Interfaz de Programación de Aplicaciones (Application Programming Interface, API

El término "Interfaz de Programación de Aplicaciones" se refiere al conjunto de funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro *software* como una capa de abstracción. Una interfaz de programación representa la capacidad de comunicación entre componentes de *software* a través de un conjunto de llamadas a ciertas bibliotecas que ofrecen acceso a determinados servicios desde los procesos y representa un método para conseguir abstracción en la programación, generalmente (aunque no necesariamente) entre los niveles o capas inferiores y los superiores del *software [WK4])*.

Uno de los principales propósitos de una API consiste en proporcionar un conjunto de funciones de uso general, por ejemplo, para dibujar ventanas o iconos en la pantalla. De esta forma, los programadores se benefician de las ventajas de la API haciendo uso de su funcionalidad, evitándose el trabajo de programar todo desde el principio. Las APIs asimismo son abstractas: el software que proporciona una cierta API generalmente es llamado la implementación de esa API.

1.4. La Plataforma de Desarrollo .Net

Abordando de manera específica el campo de la programación (y especialmente la programación para Windows), el desarrollo de *software* se ha visto enriquecido por una gran variedad de tecnologías a lo largo del tiempo: API de WIN32, MFC de C++, archivos .DLL de Visual Basic 6.0, los objetos COM, entre otros. Cada una de estas propuestas cuenta con sus pros y sus contras, y en su momento ofrecían soluciones que facilitaban la vida de los desarrolladores de *software*. Sin embargo, un denominador común que permanece es la dificultad que supone combinar diversas tecnologías y arquitecturas, y los problemas asociados en el momento de implementar los sistemas en las máquinas clientes. Además, la aparición, desarrollo y expansión del uso de Internet significó la incorporación de un escenario con características propias y distintas a las conocidas hasta el momento [LS1].

1.4.1. El *Framework* .Net

Alrededor del año 2000, Microsoft ideó una propuesta que significó una evolución de las tecnologías para desarrollo de software que representaba la creación y utilización de una plataforma única independiente del hardware y con completa integración a redes, organizada y con una completa integración entre software antiguo y entre distintos lenguajes: el *framework* .Net

Entre sus características más sobresalientes podemos mencionar:

- Un diseño basado completamente en el paradigma de la Programación Orientada a Objetos: Toda la estructura del *framework* está organizada jerárquicamente a través de clases, y éstas por medio de agrupaciones lógicas conocidas como namespaces o espacios de nombres
- Un entorno de ejecución administrado que permite crear ambientes seguros: Esto limita los problemas con el manejo de memoria (aunque posibilita el manejo de ubicaciones en memoria a través de apuntadores si es necesario), y la posibilidad de definir niveles de acceso para el código

- Integración total entre lenguajes y entre plataformas: Con el *framework* .Net es posible crear una librería con un lenguaje A, llamarla desde otro lenguaje B y depurar ambos desde el mismo ambiente integrado de desarrollo. Además, el *framework* .Net no es privativo de la familia de Sistemas Operativos Windows. Puede ser utilizado en distribuciones de Linux, Mac OS X. e incluso sistemas Android a través de complementos adicionales como el proyecto Mono [MO1].
- Un modelo de implementación simplificado: A veces, instalar una aplicación es algo tan sencillo como únicamente copiar y pegar un archivo ejecutable en una carpeta para que empiece a funcionar.
- Diseño basado en estándares: Toda la comunicación está basada en estándares del sector para asegurar que el código de .NET *framework* se puede integrar con otros tipos de código.

.NET *framework* es un componente integral de los sistemas operativos de la familia Windows que admite la compilación y la ejecución de la siguiente generación de aplicaciones y servicios Web XML. El diseño de .NET *framework* está enfocado a cumplir los objetivos siguientes:

- Proporcionar un entorno coherente de programación orientada a objetos, en el que el código de los objetos se pueda almacenar y ejecutar de forma local, ejecutar de forma local pero distribuida en Internet o ejecutar de forma remota.
- Proporcionar un entorno de ejecución de código que reduzca lo máximo posible la implementación de *software* y los conflictos de versiones.
- Ofrecer un entorno de ejecución de código que promueva la ejecución segura del mismo, incluso del creado por terceras personas desconocidas o que no son de plena confianza.
- Proporcionar un entorno de ejecución de código que elimine los problemas de rendimiento de los entornos en los que se utilizan scripts o intérpretes de comandos.
- Ofrecer al programador una experiencia coherente entre tipos de aplicaciones muy diferentes, como las basadas en Windows o en la *web*.

El .Net *framework* es considerado como la propuesta de Microsoft en respuesta a otras plataformas para el desarrollo de aplicaciones como lo es Java y diversos marcos de desarrollo basados en PHP. La intención del .Net *framework* es ofrecer una manera rápida y económica, a la vez que segura y robusta, de desarrollar aplicaciones –o como la misma plataforma las denomina, *soluciones*– permitiendo una integración más rápida y ágil entre empresas y un acceso más simple y universal a todo tipo de información desde cualquier tipo de dispositivo.

1.4.2. Estructura del .Net *framework*

El *framework* de desarrollo .Net, visto de manera general, se encuentra estructurado de la siguiente manera:

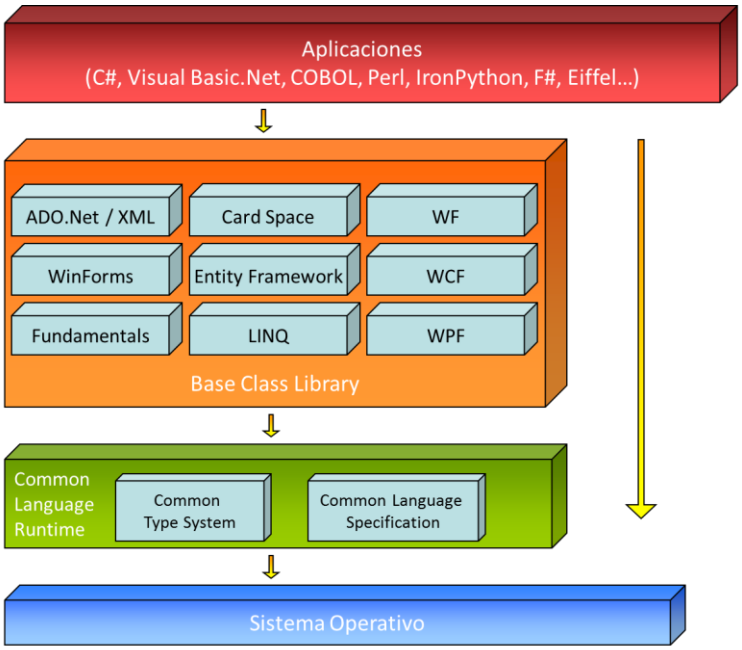


Figura 6. Estructura de .Net Framework (elaboración propia)

En el diagrama anterior podemos distinguir dos partes que resultan fundamentales, el *Common Language Runtime* (CLR) y la *Base Class Library* (BCL), también conocida como *Framework Class Library [LS1]*.

1.4.3. Common Language Runtime (CLR)

Dicho de manera sencilla, la CLR es una plataforma en tiempo de ejecución cuyo objetivo es servir de intermediaria entre el sistema operativo y nuestra aplicación. El CLR también es responsable de realizar por nosotros diversas operaciones de bajo nivel, tales como la administración de memoria, la creación de threads o hilos de ejecución, diversas verificaciones de seguridad, entre algunas otras. La estructura del CLR es similar a la de la siguiente figura:

Figura 7. Estructura interna del entorno de ejecución en lenguaje común [WK1]

El CLR realiza todas aquellas tareas de bajo nivel que resultan fundamentales para la operación normal. Comentaremos de manera rápida las más importantes:

- Cargador de clases (*Class Loader*): Es la sección del CLR responsable de ubicar en memoria todos los tipos del Framework (ya sean propios o creados por el desarrollador)
- Compilador MSIL a Código nativo: A este se le conoce también como JIT Compiler o *Jitter*, es el responsable de la traducción de código Intermedio (IL - *Intermediate Language*) a código nativo que se puede ejecutar en el equipo en el que estamos. Hablaremos de este aspecto un poco más adelante.
- Recolector de basura (*Garbage Collector*): Se encarga de retirar de memoria los objetos del *framework* que hayan cumplido con su ciclo de vida correspondiente o que hayan permanecido sin uso durante bastante tiempo
- Motor de seguridad: Es el responsable de controlar el acceso de cada pieza de código a los recursos del sistema
- Módulo de interacción con COM: Es la sección que permite trabajar con código antiguo (*legacy code*).
- Verificador de Tipos: De manera general, se encarga de asegurar la correspondencia de tipos de datos entre las aplicaciones que se ejecutan sobre el *framework*

1.4.4. Common Type System y Common Language Specification

Para hacer factible la utilización de múltiples lenguajes fácilmente integrables entre sí para la construcción de una aplicación, es necesario que exista un punto en común entre los tipos de datos utilizados por todos los lenguajes. Si todos estos lenguajes utilizan los mismos tipos de datos, entonces podrán muy fácilmente interactuar entre sí. Es aquí donde entra el Sistema Común de Tipos, o *Common Type System* (CTS), que es una especificación que define el conjunto de tipos dentro del Framework que todos los lenguajes diseñados para .Net deben soportar. Siempre que dos lenguajes cumplan con las especificaciones del CTS, podrán interactuar de manera natural y sin ningún conflicto.

Adicionalmente, existe otra especificación denominada *Common Language Specification* (CLS) que define los requisitos mínimos que un lenguaje diseñado para operar bajo el .Net *framework* debe cumplir.

1.4.5. Ensamblados

A los archivos binarios (es decir, archivos `*.dll` o `*.exe`) obtenidos a partir de la compilación de un programa escrito en un lenguaje .Net, se les denomina ensamblados (*assemblies*). Un ensamblado contiene información adicional referente a los permisos de dicha pieza de código, su versión, información sobre los formatos de presentación de acuerdo al país (*Culture Information*), la lista de los ensamblados que son requeridos para ejecutar dicho ensamblado, y una descripción de todos los tipos que contiene. A este conjunto de información se le denomina **manifiesto**.

Estos ensamblados, a diferencia de los programas diseñados con otros lenguajes, no están compilados en código máquina específico para la plataforma en que fueron desarrollados. En su lugar, contienen instrucciones en lenguaje intermedio (*Intermediate Language* o *Common Intermediate Language*). Independientemente del lenguaje en que haya sido creado, el ensamblado contendrá código CIL. Cuando un programa debe ser ejecutado, es cargado en memoria y a continuación es procesado por el compilador JIT (*Just in Time Compilation*), generando código máquina específico para la plataforma en que se ejecuta el binario. En algunas ocasiones, se puede realizar una especie de pre-compilación para acelerar el proceso de ejecución, a esto se le conoce frecuentemente como pre-JIT.

1.4.6. Base Class Library

Finalmente, otro de los componentes principales del *framework* es la *Base Class Library* (BCL), que es una enorme colección de clases que encapsulan muchas de las tareas de programación más comunes, tales como el manejo de hilos, operaciones de entrada y salida de archivos, interacción con dispositivos, acceso a orígenes de datos, diseño de interfaces de usuario tanto para escritorio como para web, cuestiones relacionadas a seguridad, etc. Todas estas funcionalidades se

encuentran organizadas a través de *namespaces*, que son agrupaciones lógicas de funcionalidades que tienen cierta relación entre sí.

Por ejemplo, el *namespace* raíz se denomina *System*, y si deseamos realizar alguna acción que tenga que ver con el manejo de archivos, podríamos utilizar las clases que se encuentran dentro del *namespace* llamado *IO*. Esto permite ubicar fácilmente las clases que nos pueden ser más útiles y evita que tengamos que reinventar la rueda al programar una aplicación. Muy frecuentemente (por no decir que siempre) cuando desarrollemos alguna aplicación dentro del *framework* nos encontraremos utilizando clases de la BCL.

Todo el código en .Net que hace uso de todas las partes mencionadas anteriormente, y que es manipulado por el CLR, se denomina código administrado (*managed code*). Lenguajes como Visual Basic.Net pueden generar únicamente código administrado. Sin embargo, si existe la necesidad de interactuar directamente con el sistema operativo (el escenario más común es la existencia de componentes que utilizan código heredado– *legacy* –que aún es de utilidad), podemos generar ensamblados que encapsulen esta funcionalidad. A este tipo de código se le conoce como código no administrado (*unmanaged code*). C++ es, por excelencia, un lenguaje que puede generar código no administrado y aprovechar al mismo tiempo de los beneficios del *framework*. Es necesario mencionar, sin embargo, que cuando uno desarrolla código administrado, debe encargarse por sí mismo de todas aquellas tareas realizadas por la CLR, como la liberación de objetos en memoria.

El .NET *framework* puede hospedarse en componentes no administrados que cargan Common Language Runtime en sus procesos e inician la ejecución de código administrado, con lo que se crea un entorno de *software* en el que se pueden utilizar características administradas y no administradas. En .NET *framework* no sólo se ofrecen varios *hosts* de motor en tiempo de ejecución sino que también se admite el desarrollo de estos hosts por parte de terceros.

Por ejemplo, ASP.NET hospeda el motor en tiempo de ejecución para proporcionar un entorno de servidor escalable para el código administrado. ASP.NET trabaja directamente con el motor en tiempo de ejecución para habilitar aplicaciones de ASP.NET y servicios *web* XML

1.5. Retos en la implementación de sistemas basados en agentes

La filosofía principal detrás del concepto de agentes se basa en la concepción de una arquitectura abierta de sistemas distribuidos en la cual cualquier agente desarrollado y ejecutado de forma independiente interactúe con otros con la finalidad de resolver problemas u ofrecer servicios a los usuarios finales.

1.5.1. ¿Por qué estandarizar?

Bajo la perspectiva de un escenario ideal de un ambiente con agentes interconectados que ejecutan tareas en nombre del usuario y para su beneficio, tal cómo podría ser la reservación de viajes o el manejo de procesos de negocio con mínima intervención del usuario, surgen diversas necesidades que deben ser satisfechas si se desea volverlo plausible.

La primera de ellas es la posibilidad de que los diferentes agentes que se encuentran en un determinado entorno sean capaces de interactuar entre sí y comunicarse efectivamente, posiblemente sólo conociendo cuál es el problema o servicio que debe resolverse y no quién lo necesita.

En este escenario ideal, muy probablemente los agentes que interactúen en un entorno distribuido serán desarrollados por diferentes proveedores y, por tanto, contarán con estrategias y restricciones de implementación distintas. Dichos agentes, además, se encontrarán distribuidos a lo largo de redes de gran escala, como lo es Internet, de tal forma que las propiedades generales del sistema multi-agentes al que pertenezcan serán el resultado de las interacciones realizadas de forma independiente por cada agente, teniendo como consecuencia un sistema un sistema robusto y flexible con un universo de agentes abierto y heterogéneo.

Por otro lado, actualmente el desarrollo de aplicaciones bajo el paradigma de agentes de *software* enfrenta las siguientes problemáticas:

- Cada equipo o empresa desarrolla aplicaciones multi-agentes de acuerdo a las necesidades y restricciones específicas de su propio proyecto, resultando en sistemas con capacidades y características distintas entre sí.

- La construcción de cada nuevo sistema empieza desde cero y sin aprovechar el conocimiento y la experiencia existente de los equipos que han enfrentado la misma tarea anteriormente, por lo que resulta imposible la reutilización de componentes y arquitecturas.
- La gran mayoría de las aplicaciones multi-agentes se realizan con propósitos académicos e investigativos, sin realizar una integración real con la infraestructura existente en las industrias, en donde es necesario abarcar aspectos como escalabilidad, seguridad y rentabilidad desde el punto de vista económico.

Ante este panorama, resulta evidente la necesidad de establecer mecanismos que permitan la interacción y comunicación entre agentes independientemente de la plataforma de desarrollo en que se realicen y que faciliten el desarrollo de sistemas abiertos; es decir, la definición de estándares que definan interfaces comunes entre agentes y otras entidades.

1.5.2. Iniciativas de estandarización

En el campo de los agentes de software se llevan a cabo las siguientes actividades de estandarización:

ARPA Knowledge Sharing Effort (KSE)

KSE es un consorcio cuyo objetivo es "desarrollar convenciones que faciliten el intercambio, la compartición y reutilización de bases de conocimiento y sistemas basados en conocimiento". Algunos de los productos más populares generados por dicho consorcio son el lenguaje para la manipulación y consulta de conocimiento (*Knowledge Querying Manipulation Language*, KQML) y el formato de intercambio de conocimiento (*Knowledge Interchange Format*, KIF), además de Ontolingua, una herramienta diseñada para la definición de ontologías.

Durante un tiempo, el lenguaje KQML fue considerado un estándar de facto para el intercambio de conocimiento entre sistemas multi-agente. No parece claro que el KSE siga todavía en activo aunque algunas de sus partes siguen trabajando como grupos de investigación independientes. Parece que nadie toma la responsabilidad de continuar con el desarrollo de estándar original.

OMG Mobile Agent Facility (MAF)

El Object Management Group es un consorcio integrado por empresas como IBM, General Electric, Crystaliz, GMD-Focus y The Open Group, y que en conjunto desarrollaron una propuesta denominada MAF, concebida inicialmente como una extensión al estándar CORBA (*Common Object Request Broker Architecture*) para incluir objetos que tuvieran la capacidad de migrar de un sistema a otro sin perder su estado a través de la definición de interfaces para la transferencia y localización de agentes en sistemas de agentes implementados sobre CORBA.

Esta propuesta generó diversos debates al interior de OMG, por lo que actualmente la situación en la que se encuentra esta propuesta no resulta clara.

The Agents Society

Esta iniciativa fue creada con la intención de "colaborar en el desarrollo generalizado y la aparición de tecnologías de agentes inteligentes así como en su colocación dentro del mercado". Básicamente se trata de un punto de encuentro común para el intercambio y recopilación de información sobre agentes de *software*, aunque el ritmo de actualización no resulta tan ágil como sería deseable.

FIPA (Foundation for Intelligent Physical Agents)

FIPA es una organización internacional que se dedica a promover la tecnología basada en agentes a través del desarrollo de especificaciones abiertas que soporten la interoperabilidad entre los agentes y las aplicaciones basadas en agentes. Esto ocurre a través de la colaboración abierta

entre sus miembros, entre los que se encuentran empresas y universidades que trabajan activamente en el campo de agentes. FIPA publica y hace disponibles los resultados de todas sus actividades a las partes interesadas con la intención de contribuir, a través de sus resultados, a la formación de estándares.

Las especificaciones FIPA se desarrollan mediante la participación directa de sus miembros, los cuales se encuentran, en lo individual y en lo colectivo, comprometidos a la creación de una competencia abierta en el desarrollo de aplicaciones y servicios basados en agentes.

Capítulo 2

Desarrollo de una biblioteca de clases para la simulación de tráfico vehicular basada en agentes

2. Introducción al capítulo

En los primeros días de la era de la PC, las herramientas con las que contaba un desarrollador para poder programar aplicaciones eran escasas: un compilador, un muy reducido conjunto de bibliotecas con funcionalidades mínimas, y una interfaz de programación de aplicaciones (Application Programming Interface, API) de bajo nivel, ligada directamente a funciones del sistema operativo.

2.1. Desarrollo de la API

A medida que los programadores desarrollaban aplicaciones con estas herramientas básicas, se hizo notorio el hecho de que cada vez con mayor frecuencia se reutilizaba el mismo código para realizar funciones comunes dentro de un programa. Bajo esta perspectiva, el siguiente paso lógico hacia una optimización era realizar una abstracción de este código repetitivo a través de APIs de alto nivel.

Los fabricantes de sistemas operativos pronto descubrieron que, si se incluían estas API de alto nivel como parte de las distribuciones, el desarrollo de las aplicaciones se llevaría a cabo con mayor velocidad y sin demandar tanto esfuerzo por parte del programador, lo que significaría un incremento en las aplicaciones desarrolladas para ese sistema operativo y que lo volvería, en

consecuencia, más atractivo para los usuarios finales que demandan una gran variedad de aplicaciones para diversas necesidades. Incluso, los fabricantes de componentes individuales pronto identificaron el potencial que representaba ofrecer API con un alto nivel de abstracción como una buena oportunidad de negocio.

En forma paralela, las tendencias en la industria del desarrollo de software comenzaron a converger hacia el diseño orientado a objetos con un especial énfasis en la extensibilidad y la reusabilidad. El concepto que ahora conocemos como *framework* surgió en el momento en el que los fabricantes de librerías reutilizables adoptaron el paradigma orientado a objetos para el desarrollo de sus bibliotecas de alto nivel.

A partir de este momento, el desarrollo de una nueva aplicación no implicaba comenzar a crear código desde cero; el *framework*, en cambio, proporcionaba los componentes principales necesarios, mismos que podían ser modificados y conectados entre sí para poder construir la aplicación.

Si bien el desarrollo de *frameworks* permitió, en un inicio, facilitar la creación de aplicaciones y disminuir los tiempos de desarrollo, a medida que se incrementaba el número de fabricantes que desarrollaban componentes reutilizables, surgió un nuevo problema: el nivel de complejidad y el esfuerzo necesario para poder interconectar distintos componentes (a menudo incompatibles entre sí) comenzó a impactar el nivel de productividad de los equipos de programación.

Otra consecuencia de esta situación fue que una gran cantidad de aplicaciones estaban constituidas por código fuente poblado en su gran mayoría de numerosas llamadas a API en lugar de las estructuras propias del lenguaje, lo que a su vez dificultaba la lectura y revisión del código, dando la impresión de que una misma sección contenía “varios lenguajes”: el lenguaje propio de la aplicación y las diferentes sintaxis utilizadas por los componentes para realizar las llamadas a las funciones de cada API. En este punto, resultó más que evidente la necesidad de contar con un conjunto estandarizado de reglas de diseño comunes que faciliten la integración entre componentes y que aseguren que los componentes reutilizables se adapten de manera consistente.

2.2. Factores que influyen en el diseño de una biblioteca de clases

Existe un conjunto de características entre las que se cuentan la seguridad, la confiabilidad, el rendimiento o la administración de dependencias, por mencionar algunos, que distinguen a un *framework* bien diseñado del resto. En realidad, la diferencia entre un *framework* y otros tipos de software radica en que los primeros están conformados por API reutilizables diseñadas utilizando ciertos principios básicos que inciden en la calidad del *software*. A continuación se enumeran los más importantes.

2.2.1. Simplicidad

Es una práctica común que los *frameworks* que se encuentran en fases iniciales de desarrollo cuenten con un conjunto reducido de funcionalidades. A medida que los requisitos definidos para su implementación se vuelven más claros, se van incorporando características que le agregan potencia. En el lado opuesto de la balanza, un vicio frecuente en el desarrollo de bibliotecas es sacrificar la simplicidad debido a distintos factores externos tales como la presión para liberar en alguna fecha determinada, un exceso de requerimientos por parte del cliente, o incluso el deseo de los desarrolladores por cubrir hasta el escenario más específico de los requerimientos. Ante la duda de incluir o no una característica que incremente la complejidad del diseño, la gran mayoría de las veces es mucho mejor eliminar la característica en cuestión de la versión actual y dedicar un mayor tiempo y esfuerzo para obtener un diseño óptimo que pueda incluirse en la siguiente versión. Una frase común entre los diseñadores de API es “Siempre puedes agregar, pero nunca quitar”. Si uno no se siente satisfecho con el diseño, y aun así se realiza una liberación, es muy probable que uno lo lamente después.

2.2.2. Diseño bien elaborado

Un *framework* bien diseñado no ocurre por arte de magia; es el resultado de un arduo trabajo que suele consumir una gran cantidad de tiempo y recursos. Para obtener un buen diseño es necesario estar dispuesto a invertir de verdad en ello.

El diseño de un *framework* debe ser una fase clara y explícita dentro del proceso de desarrollo. Debe ser explícito porque requiere de una adecuada planeación, asignación de recursos y ejecución, y no debe ser sólo una consecuencia de un proceso de implementación. Generalmente los *frameworks* mejor diseñados son aquellos que están desarrollados por un equipo donde se encuentran personas cuya responsabilidad explícita es el diseño de la biblioteca, o bien que sean capaces de "ponerse el sombrero" de diseñador en el momento adecuado dentro del proceso de desarrollo. Por el contrario, mezclar las responsabilidades de diseño y desarrollo de una biblioteca es un error que puede conducir a un diseño pobre que exponga las características de la implementación a los usuarios finales.

2.2.3. Diseño equilibrado

La verdad es que el "diseño perfecto" no existe. El diseño básicamente se trata de hacer concesiones y tomar las decisiones correctas, conocer y entender las opciones disponibles, sus ventajas e inconvenientes. A menudo estas decisiones se toman dependiendo de las situaciones específicas en las que se encuentre inmersa la biblioteca en cuestión.

2.2.4. Aprovechamiento de experiencias pasadas

Las bibliotecas más exitosas y mejor diseñadas suelen estar basadas en diseños ampliamente probados. Es posible, y con frecuencia deseable, introducir soluciones novedosas en el diseño de un *framework*; sin embargo, esto debe realizarse con mucho cuidado, ya que a medida que se incrementa el número de nuevos conceptos que se integran a una biblioteca, la probabilidad de obtener un diseño aceptable disminuye. Una pauta generalmente aceptada en estos casos es

evitar en la medida de lo posible innovar en el diseño de la biblioteca. La parte interesante de la API debe radicar en su funcionalidad, más que en su diseño.

2.2.5. Diseño planeado para evolucionar

Planear la forma en la que la biblioteca se desarrollará en el futuro tiene sus ventajas y desventajas. Por un lado, el diseñador del *framework* podría invertir mucho más tiempo y esfuerzo en el proceso de diseño, y en ocasiones se puede introducir complejidad adicional sólo "por si acaso". Pero, por otro lado, una planeación cuidadosa podría evitar que se incluyan características en la biblioteca que pueden degradarse con el paso del tiempo, o peor aún, que se incluyan características que sean incapaces de mantener la compatibilidad entre diferentes versiones. Como regla general, es mucho mejor mover la incorporación de una característica completa hacia una versión posterior, en lugar de incluir una funcionalidad inconclusa en la versión actual.

2.2.6. Integración

Los *frameworks* modernos necesitan ser diseñados teniendo en mente que trabajarán inmersos en un gran ecosistema de herramientas de desarrollo, diferentes lenguajes de programación y modelos de aplicaciones, por mencionar algunas variables. La llegada de conceptos tales como la computación distribuida significó el término de la era de las bibliotecas diseñadas para modelos específicos de aplicación.

2.2.7. Consistencia

La consistencia es una característica clave en el diseño de bibliotecas de clases y constituye uno de los factores más importantes que afectan la productividad y pueden definir el éxito o fracaso del desarrollo. Un *framework* coherente facilita la transferencia de conocimiento entre los módulos de la biblioteca que el desarrollador ya conoce y aquellos que está tratando de aprender. La consistencia también resulta fundamental para ayudar a los desarrolladores a reconocer de

manera rápida qué partes del diseño están enfocadas a situaciones o condiciones específicas y por lo tanto requieren de una mayor atención, y cuáles son simplemente adaptaciones de los patrones de diseño ya conocidos.

2.3. Fundamentos de diseño de bibliotecas de clases

Para que una biblioteca de clases, o *framework*, de propósito general pueda considerarse exitoso, debe necesariamente haber sido diseñado para una amplia gama de desarrolladores con requerimientos, habilidades y formación diferentes. Uno de los retos más grandes que los diseñadores de *frameworks* enfrentan hoy en día es ofrecer, al mismo tiempo, poder y sencillez en el diseño de cualquier código que será usado por diversos grupos de clientes.

Otro objetivo no menos importante es el poder ofrecer un modelo de programación unificado que sea independiente del tipo de aplicación que el desarrollador está construyendo o el lenguaje de programación que se utilice.

2.3.1. *Frameworks* Progresivos

Diseñar una API para un amplio grupo de desarrolladores, escenarios y lenguajes de programación es definitivamente una empresa difícil y costosa. Históricamente, los fabricantes de APIs han ofrecido productos dirigidos a grupos de desarrolladores y escenarios específicos.

Aunque este enfoque "*multi-frameworks*" ha resultado exitoso, dado que provee de APIs que resultan potentes y fáciles de utilizar para grupos de desarrolladores específicos, es importante resaltar que tiene desventajas importantes. La principal es que ante tal multitud de API disponibles, se vuelve difícil para un desarrollador que utiliza un *framework* específico transferir su conocimiento al siguiente nivel o a un escenario distinto (que muy a menudo requiere un *framework* diferente). Por ejemplo, cuando existe la necesidad de implementar una aplicación distinta que requiere de funcionalidades extendidas, los desarrolladores frecuentemente se enfrentan a una curva de aprendizaje muy pronunciada, ya que deben aprender un nuevo estilo de programación.

En este sentido, un mejor enfoque es proveer un *framework* progresivo, que es una API dirigida a un amplio espectro de desarrolladores y que permite la transferencia de conocimiento desde los escenarios más básicos hasta los más avanzados. El *framework* .Net de Microsoft es un ejemplo de este tipo de *frameworks* que ofrecen una curva de aprendizaje gradual.

Lograr una curva de aprendizaje gradual que cuente con un punto de entrada bajo es una tarea difícil, aunque no imposible. Es difícil debido a que requiere de un nuevo enfoque para el proceso de diseño de la API, demanda una gran disciplina y tiene un costo de diseño más alto.

Un aspecto importante a tener en cuenta es considerar que la comunidad de desarrolladores es muy amplia. Incluye desde empleados de oficina que graban macros para después editarlas hasta desarrolladores de controladores de bajo nivel de dispositivos. Cualquier *framework* que intente ser de utilidad para todos los usuarios corre el riesgo de terminar resultado un desastre que no satisfaga a todos los programadores. En este punto, es importante recordar que siempre existirán desarrolladores que requieran de API específicas y que cuentan con escenarios únicos y bien definidos.

2.3.2 Principios fundamentales de diseño de API

Proveer de una plataforma de desarrollo que sea a la vez poderosa y fácil de utilizar es uno de los principales objetivos del *framework* .Net, y como tal, debe ser uno de los objetivos de cualquier API que busque extender su funcionalidad. La primera versión del *framework* .Net contaba con un poderoso conjunto de API, pero un gran número de desarrolladores se enfrentaron con dificultades al utilizar secciones específicas del *framework*.

A través de encuestas de usuario y estudios de usabilidad, se encontró que un gran segmento perteneciente al grupo de desarrolladores de Visual Basic tuvo problemas para aprender Visual Basic .Net. Gran parte del problema se debió simplemente al hecho de que las librerías de .Net son diferentes a las de Visual Basic 6.0, pero además se detectaron varios problemas relacionados con usabilidad relacionados al diseño de la API. Solucionar estos problemas se convirtió en la principal prioridad para Microsoft durante el desarrollo del *framework* .Net 2.0

Los principios que se describen a continuación fueron desarrollados para solucionar la problemática comentada líneas arriba, y fueron creadas para ayudar a los diseñadores de *frameworks* a no incurrir los errores y vicios de diseño reportados más frecuentemente en los estudios realizados.

Principio de diseño basado en escenarios

Los *frameworks* a menudo contienen un gran conjunto de API. Esto es necesario debido a que de esta forma se pueden habilitar escenarios que requieren mayor poder y expresividad; sin embargo, gran parte del desarrollo se realiza en un pequeño conjunto de escenarios comunes que utilizan un relativamente pequeño subconjunto de todo el *framework*. Para optimizar la productividad del desarrollador que usa un *framework*, es crucial dedicar gran parte de los esfuerzos al diseño de APIs que sean utilizadas en los escenarios más comunes.

Dado lo anterior, el diseño de *frameworks* debe enfocarse en un conjunto de escenarios comunes hasta el punto en el que el diseño completo es basado en los escenarios. Una recomendación para este principio en particular es que los diseñadores de *frameworks* inicialmente escriban código que los usuarios utilizarán para la programación de varios escenarios, y después diseñar el modelo de objetos que soporte estos extractos de código construidos.

Principio de Desarrollo de Frameworks:

Los frameworks deben ser diseñados comenzando por un conjunto de escenarios de uso y ejemplos de código que implementen estos escenarios [CA1].

Un problema común en el diseño de *frameworks* es iniciar el desarrollo del modelo de objetos (a través de diversas metodologías de diseño) y continuar escribiendo ejemplos de código basados en la API resultante. El problema es que la mayoría de las metodologías (incluyendo las más comúnmente utilizadas en diseño orientado a objetos) están optimizadas para lograr una implementación robusta y sostenible, y no para que las APIs resultantes tengan buena usabilidad.

Su propósito principal es diseño arquitectónico interno y no el diseño de APIs para interfaces públicas como parte de un *framework*.

Durante la fase de diseño del *framework*, uno debe iniciar produciendo especificaciones para las API basadas en escenarios. Esta especificación puede realizar de manera separada a la especificación funcional, o bien como parte de un documento de especificación más detallado. En este último caso, la especificación de la API debe preceder a la especificación funcional en lugar y tiempo.

Esta especificación debe contener una sección de escenarios que liste los 5 a 10 escenarios principales para un área tecnológica específica, y debe incluir ejemplos de código que implementen dichos escenarios.

Adicionalmente, es importante que estos escenarios se codifiquen utilizando diferentes estilos de codificación que sean comunes entre los usuarios de ciertos lenguajes.

Principio de nivel de entrada bajo

De manera general, un desarrollador espera aprender las características básicas de un nuevo *framework* de manera rápida. Es muy común que este aprendizaje se realice a través de experimentar con diversas partes del *framework*, y dedicar tiempo para comprender a fondo la arquitectura completa sólo si se encuentran interesados en alguna característica importante o si requieren implementar escenarios más complejos. El primer encuentro con una API mal diseñada puede dejar una mala impresión de complejidad y puede provocar que el desarrollador pierda la motivación por seguir utilizando el *framework*. Es debido a esto que es muy importante que los *frameworks* tengan un punto de entrada muy básico para aquellos programadores que sólo desean experimentar con el *framework*.

Principio de diseño de frameworks:

Los frameworks deben ofrecer un nivel bajo de entrada basado para que los usuarios inexpertos puedan familiarizarse con el framework a través de la experimentación [CA1].

Muchos desarrolladores quieren experimentar con una API para descubrir lo que hace y luego ajustar su código gradualmente hasta hacer que su programa haga realmente lo que necesitan.

Para facilitar la experimentación en una API, es recomendable que se considere lo siguiente

- Permitir una fácil identificación del conjunto adecuado de tipos y miembros para las tareas de programación comunes. No sería fácil experimentar con un *namespace* (espacio de nombres) que tuviera más de 500 tipos, de los cuales sólo unos pocos son realmente importantes en los escenarios más comunes. De la misma manera, la experimentación se dificulta si los namespaces tienen una gran cantidad de miembros que están diseñados sólo para ayudar en los escenarios más complejos.

- Permitir a los desarrolladores su utilización inmediata, independientemente de que sirva a los propósitos requeridos por el programador. Un *framework* que requiere de extensos procesos de inicialización o instanciación de varios tipos antes de poder comenzar a implementar los escenarios más comunes dificulta la experimentación.

- De manera similar, una API que no ofrece métodos de sobrecarga con listas de parámetros cortas, o bien que cuenta con valores por default mal seleccionados para sus propiedades representan un alto nivel de complejidad para los desarrolladores que sólo buscan experimentar con una API.

- Permiten encontrar y corregir de manera fácil los errores provocados por un uso incorrecto de la API. Por ejemplo, las APIs debe lanzar excepciones que claramente describan lo que necesita hacerse para resolver el problema.

ADO.NET es un ejemplo de un área con características que los usuarios encontraban difíciles de utilizar debido a la extensa inicialización que requería. Incluso en los escenarios más sencillos, se esperaba que los usuarios comprendieran las complejas interacciones y dependencias entre varios tipos. Para utilizar las características de ADO.Net, incluso en los casos más básicos, los usuarios deberían crear y conectar instancias de varios objetos (instancias de DataSet, DataAdapter, SqlConnection y SqlCommand).Este problema se remedió en parte con la incorporación de clases adicionales en la versión 2.0 del *Framework*.

`System.Messaging.MessageQueue` es una buena ilustración de este concepto. El componente puede enviar mensajes después de pasar sólo una cadena de ruta para el constructor y llamar al método Send. La prioridad del mensaje, algoritmos de cifrado y otras propiedades de los mensajes se pueden personalizar añadiendo código al escenario simple

```
var ordersQueue = new MessageQueue(path);

ordersQueue.Send(order);  //Utiliza ajustes predeterminados de prioridad, encripción, etc.
```

Principio de documentación a través de modelos de objetos

La mayoría de los *frameworks* existentes en el mercado constan de cientos, si no miles, de tipos, y de un número mucho más elevado de miembros y parámetros. Los desarrolladores que los utilizan a menudo requieren invertir una gran parte de su tiempo en orientación y guías para su correcta utilización. Bajo estos escenarios, la documentación de referencia, por sí sola, no resulta suficiente. Si se requiere consultar la documentación incluso para responder a las preguntas más simples, es muy probable que se dedique mucho tiempo a ello y como consecuencia se interrumpa el flujo de trabajo del desarrollador. Por otra parte, como se ha comentado anteriormente, la mayoría de los programadores prefieren codificar a prueba y error, y recurren a la documentación sólo cuando resulta difícil intuir el uso correcto de la API.

Principio de diseño de frameworks:

En escenarios simples, las APIs deben poder utilizarse sin necesidad de recurrir a la documentación[CA1].

Por lo anterior, es muy importante diseñar APIs que no requieran que los desarrolladores tengan que leer la documentación cada vez que quieran llevar a cabo una tarea sencilla. Las siguientes directrices pueden ayudar en el diseño de API intuitivas que prácticamente se documentan por sí solas. Ahora bien, hay que recordar que no todas las APIs pueden explicarse por sí solas y que existen desarrolladores que prefieren entender a fondo la arquitectura y diseño de la API antes de comenzar a utilizarla.

Para diseñar un *framework* intuitivo, se debe tener cuidado con la selección de nombres, tipos, diseño de excepciones, etc. Las siguientes secciones describen algunas de las consideraciones más importantes relacionadas con este punto.

Nomenclatura

La manera más sencilla para diseñar un *framework* intuitivo, y paradójicamente la menos aprovechada, es reservar los nombres más comunes y descriptivos para los tipos que se espera que usen los desarrolladores para instanciar objetos en los escenarios más comunes. Los diseñadores de *frameworks* a menudo "queman" los mejores nombres para los tipos menos comunes, con los que la gran mayoría de los usuarios no tiene ninguna relación.

Otra recomendación es utilizar identificadores descriptivos que definan claramente lo que cada método realiza, y lo que cada tipo y parámetro representa.

Por ejemplo, `EventLog.DeleteEventSource(string source, string machineName)` podría lucir como un identificador demasiado extenso; pero desde la perspectiva de la usabilidad tiene un gran valor positivo.

Los nombres de métodos descriptivos sólo son posibles para métodos que tienen una semántica clara y definida. Esta es una de las razones por la cuales evitar semánticas complejas es un muy buen principio de diseño que debe ser replicado.

El mensaje principal es que los nombres que se definan para los identificadores deben ser seleccionados con sumo cuidado. La elección de los nombres de identificadores es una de las decisiones más importantes que se deben de tomar cuando se diseña un *framework*. Es extremadamente difícil y costoso realizar modificaciones a los nombres de los identificadores una vez que se ha liberado una API.

Excepciones

Las excepciones tienen un rol crucial en el diseño de *frameworks* intuitivos. Deben comunicar al desarrollador el uso correcto de la API a través de los mensajes de excepción. Por ejemplo, el siguiente segmento de código debe lanzar una excepción con el mensaje "La propiedad `source` no ha sido definida antes de realizar la escritura al `log` de eventos"

```
//C#
var log = new EventLog();
//No se ha definido previamente la propiedad source.
Log.WriteEntry("Hello World");
```

Definición estricta de tipos

La definición estricta de tipos es probablemente el factor más importante para determinar qué tan intuitiva resulta una API. Como ejemplo, basta mencionar que invocar `Customer.Name` es más sencillo que `Customer.Properties["Name"]`. Además, si la propiedad `Name` devuelve el nombre como una cadena de texto, tiene mayor utilidad que si la propiedad devolviera un objeto.

Existen ocasiones en las que los contenedores de propiedades, los llamados a funciones en tiempo de ejecución (*late-bound calls*), y otros métodos de definición de tipos más flexibles son necesarios

en una API; sin embargo, esto debe ser una excepción a la regla y no una práctica común. La práctica recomendada para los diseñadores de bibliotecas es que consideren la posibilidad de ofrecer definiciones estrictas de tipos para las operaciones más comunes que el usuario puede utilizar. Por ejemplo, un tipo de datos `Cliente` podría tener un contenedor de propiedades y de manera adicional implementar definiciones estrictas de tipos para las propiedades más comunes, como nombre, dirección, etcétera.

Coherencia

Mantener coherencia con API existentes que resultan familiares para el usuario es otra técnica muy valiosa para diseñar frameworks intuitivos. Esto también incluye la coherencia con otras API del *framework* .Net así como con API heredadas.

La coherencia es una característica importante en temas de usabilidad. Si un usuario está familiarizado con alguna parte de un *framework* que resulta similar a la API que uno está desarrollando, existen grandes posibilidades de que el diseño de ésta última le resulte más natural e intuitivo. Nuestra API debe diferenciarse de las demás únicamente en aquellas secciones donde existe alguna característica distintiva y particular de nuestra API.

Abstracciones limitadas

Las implementaciones de los escenarios más comunes en la API no deben utilizar demasiadas abstracciones, sino que más bien deben corresponder a las partes físicas, o las secciones lógicas mejor reconocidas de un sistema. Como se ha mencionado anteriormente, los modelos estándar orientados a objetos están dirigidos a producir diseños optimizados para facilitar el mantenimiento de la base de código. Esto tiene sentido si consideramos que los costos de mantenimiento representan una de los mayores gastos asociados al desarrollo de un producto de software. Una forma de mejorar la manutención del código es a través del uso de abstracciones, esto se debe a que las metodologías de diseño modernos tienden a producir una gran cantidad de ellas.

El problema se presenta cuando los *frameworks* que cuentan con una gran cantidad de abstracciones obligan a los usuarios a convertirse en expertos en la arquitectura de la API antes de comenzar a implementar incluso los escenarios más sencillos; sin embargo, la mayoría de los desarrolladores no tienen el deseo de volverse expertos en todas las API que proporciona un *framework*. Para los escenarios más simples, los desarrolladores demandan API que sean lo suficientemente sencillas e intuitivas como para ser utilizadas sin necesidad de comprender cómo funcionan en conjunto todas las partes del *framework*.

Es importante resaltar que las abstracciones tienen un lugar muy importante en el diseño de *frameworks*. Por ejemplo, las abstracciones pueden ser muy útiles para mejorar la extensibilidad de un *framework*. A menudo, dicha extensibilidad es posible gracias a un diseño de abstracciones bien elaborado.

Principio de Arquitectura por niveles

No todos los desarrolladores requieren resolver los mismos problemas. Distintos desarrolladores a menudo requieren y esperan diferentes niveles de abstracción y de control de los *frameworks* que utilizan. Algunos desarrolladores que utilizan típicamente C++ o C# podrían buscar APIs que sean más expresivas y que tengan mayor capacidad. Este tipo de API son conocidas como API de bajo nivel, debido a que ofrecen un bajo nivel de abstracción. Por otro lado, otros desarrolladores que típicamente utilizan C# o VB.Net valoran más las API que les permiten optimizar la productividad y privilegian la simplicidad. A este tipo de API se les conoce como API de alto nivel, ya que ofrecen un nivel de abstracción más elevado. Mediante el uso de un diseño en capas es posible construir un único *framework* que cumpla con estas diversas necesidades.

Principio de Diseño de Frameworks:

El diseño en capas permite proporcionar potencia y facilidad de uso en un mismo framework [CA1].

La recomendación general para la construcción de un único *framework* dirigido a un amplio grupo de desarrolladores es "re-factorizar" la API en un conjunto de tipos de bajo nivel que expongan toda la riqueza y el poder de la biblioteca, así como un conjunto de tipos de alto nivel que sirvan para encapsular la capa de bajo nivel y así facilitar su uso.

El término refactorización se usa a menudo para describir la modificación del código fuente sin cambiar su comportamiento, lo que se conoce informalmente por limpiar el código. La refactorización se realiza a menudo como parte del proceso de desarrollo del *software*: los desarrolladores alternan la inserción de nuevas funcionalidades y casos de prueba con la refactorización del código para mejorar su consistencia interna y su claridad. Las pruebas aseguran que la refactorización no cambia el comportamiento del código.

La refactorización es la parte del mantenimiento del código que no arregla errores ni añade funcionalidad. El objetivo, por el contrario, es mejorar la facilidad de comprensión del código o cambiar su estructura y diseño y eliminar código muerto, para facilitar el mantenimiento en el futuro. Añadir nuevo comportamiento a un programa puede ser difícil con la estructura dada del programa, así que un desarrollador puede re-factorizarlo primero para facilitar esta tarea y luego añadir el nuevo comportamiento.

Esta recomendación representa una técnica muy ponderosa para simplificar el diseño de la API. En una API de una sola capa, a menudo uno se ve forzado a decidir entre contar con un diseño más complejo y dejar de soportar algunos escenarios. Contar con una capa de bajo nivel proporciona la libertad necesaria para implementar una API de alto nivel para los escenarios principales.

Es importante tener en cuenta que en algunos casos podría no requerirse alguna de las capas. Por ejemplo, en algunas situaciones particulares las API expuestas podrían ser sólo de bajo nivel.

ASP.NET (el *namespace* `System.Web`) es un ejemplo de este tipo de diseño en capas. Para los desarrolladores enfocados en la potencia y la expresividad, ASP.NET ofrece una capa de bajo nivel HTTP que permite a los programadores codifiquen directamente sobre las peticiones lanzadas por el servidor web, con muy poco nivel de abstracción. Sin embargo, ASP.NET también ofrece un amplio conjunto de controles *web* que permiten a los desarrolladores programar contra conceptos de alto nivel a través del uso de propiedades y métodos sin tener que preocuparse sobre los

protocolos *web*. De esta manera, ASP.NET ofrece un único *framework* que cuenta con capas dirigidas a audiencias y escenarios distintos.

Existen dos enfoques principales para la factorización de espacios de nombres (*namespaces*) en las distintas capas de una API:

- Exponer las capas en espacios de nombres separados
- Exponer las capas en el mismo espacio de nombres.

Exposición de capas en espacios de nombres separados

Una forma de re-factorizar un *framework* es colocando los tipos de alto y de bajo nivel en *namespaces* diferentes, pero relacionados entre sí. Esta alternativa tiene la ventaja de ocultar los tipos de bajo nivel de los escenarios principales, sin alejarlos demasiado de los desarrolladores que necesitan implementar escenarios más complejos.

ASP.Net está factorizado de esta manera. Las APIs de bajo nivel `HttpRuntime` se encuentran en un espacio de nombres distinto que el que contiene los controles de página *web* de alto nivel (ubicados en el *namespace* `System.Web.UI`). La API encargada de realizar el manejo de las páginas está construida a partir de la estructura de bajo nivel que implementa la interfaz `IHttpHandler`, y proporciona un modelo de programación sencillo y abstracto para la gran mayoría de los escenarios habituales.

La gran mayoría de los *frameworks* deberían seguir este enfoque de factorización de *namespaces*.

Exposición de capas en el mismo espacio de nombres

Otra manera de factorizar un *framework* es colocar las API de alto y de bajo nivel en el mismo espacio de nombres. Esta alternativa tiene la ventaja de proveer un acceso más sencillo a funcionalidades más avanzadas o sofisticadas de acuerdo al tipo de funcionalidad requerida. Por otro lado, esta misma característica podría implicar la existencia de tipos complejos dentro del espacio de nombres que podrían dificultar la implementación de algunos escenarios.

Este tipo de factorización funciona mejor para características sencillas. Por ejemplo, el espacio de nombres `System.Net` incluye tipos de bajo nivel como `Dns`, así como tipos de alto nivel como la clase `WebClient`. Es importante notar que incluso las clases conectoras de bajo nivel están factorizadas fuera en el *namespace* `System.Net.Sockets`.

2.4. Simulación basada en agentes

La simulación es generalmente reconocida como una de las mejores herramientas tecnológicas de soporte en el diseño de sistemas de naturaleza compleja, dinámica y estocástica, además de ser una ayuda valiosa durante el proceso de toma de decisiones tácticas y estratégicas.

Un modelo de simulación está compuesto por un conjunto de reglas que definen la manera en que un sistema cambia a través del tiempo a partir de un estado determinado. A diferencia de los modelos analíticos, un modelo de simulación no se resuelve, sino que es ejecutado a efectos de observar los cambios de estado del sistema en un momento determinado. Esto nos permite obtener una idea de la dinámica del sistema en lugar de sólo predecir la salida del mismo basado en entradas específicas. La simulación no es en estricto sentido una herramienta para la toma de decisiones, sino que es una herramienta de soporte a las decisiones, que permite la toma de las mismas de manera más informada.

Debido a la complejidad existente en los sistemas del mundo real, los modelos de simulación presentan sólo una aproximación del sistema objetivo pero lo suficientemente completo como para apreciar el funcionamiento o evolución del sistema. La esencia del modelado de sistemas reside en la abstracción y la simplificación. Sólo aquellas características que son importantes para el estudio y análisis del sistema objetivo deben ser incluidas en el modelo de simulación.

El propósito de la simulación es obtener una mayor comprensión sobre la operación de un sistema objetivo, o bien realizar predicciones acerca del rendimiento de un sistema objetivo. Puede ser visto como un laboratorio de experimentación que no sólo permite obtener conocimiento, sino también probar nuevas teorías y prácticas sin interrumpir la rutina diaria de un modelo organizativo.

Una de las principales ventajas que ofrece un estudio de simulación es que, si una teoría que ha sido propuesta para un sistema se comprueba, y si esta teoría ha sido adecuadamente trasladada a un modelo diseñado por computadora, esto podría permitirnos responder las siguientes interrogantes:

- ¿Qué tipo de comportamiento se puede esperar bajo una determinada combinación de parámetros definidos y ciertas condiciones iniciales?
- ¿Qué tipo de comportamiento se reflejará en el sistema en el futuro?
- ¿Qué estado tomará el sistema objetivo en el futuro?

El nivel de precisión requerida por un modelo de simulación depende en gran medida de las preguntas que uno está tratando de contestar. Para poder responder a la primera de las interrogantes listadas anteriormente, el modelo de simulación requiere ser un modelo explicativo. Este tipo de modelo requiere una precisión menor en los datos. En contraste, el modelo de simulación requerido para responder las dos últimas preguntas tiene que ser de naturaleza predictiva y por tanto requiere de información con un alto nivel de precisión para poder producir salidas fidedignas. Esta naturaleza predictiva implica que el modelo deberá producir como resultado tendencias más que predicciones absolutas y precisas del rendimiento del sistema objetivo.

Los resultados numéricos de un experimento de simulación generalmente no son útiles por sí mismos y necesitan ser analizados de manera rigurosa a través de métodos estadísticos. Estos resultados deben ser considerados en el contexto de los sistemas reales y por tanto deben ser interpretados de manera cualitativa para definir recomendaciones significativas o bien para recopilar buenas prácticas. La persona que analiza los resultados de un modelo requiere de un alto nivel de comprensión del comportamiento del sistema objetivo para poder explotar todo el conocimiento obtenido a través de los modelos de simulación.

2.4.1. Técnicas de Simulación

Existen tres tipos de modelado de simulación que son utilizados para poder colaborar en la comprensión del comportamiento de los sistemas organizacionales:

- Simulación discreta de eventos (*Discrete Event Simulation*, DES)
- Dinámica de sistemas (*System Dynamics*, SD)
- Simulación basada en agentes (*Agent Based Simulation*, ABS)

La simulación discreta de eventos modela sistemas como un conjunto de entidades que son procesadas y evolucionan a través del tiempo de acuerdo con la disponibilidad de recursos y disparadores de eventos específicos. El simulador mantiene una cola ordenada de eventos. Este tipo de simulación es utilizado ampliamente para el soporte a decisiones relacionadas a manufactura y procesamiento en serie, así como para las industrias de servicios.

La dinámica de sistemas modela los cambios en un sistema a través del tiempo en un enfoque *top-down*. El analista tiene que identificar las variables de estado principales que definen el comportamiento del sistema y cómo se relacionan con las demás a través de ecuaciones diferenciales. La dinámica de sistemas se aplica en entornos donde los individuos no cuentan con elementos diferenciados y existe información disponible en niveles agregados. Dos ejemplos típicos son el modelado de población y de sistemas económicos y ecológicos.

En una simulación basada en agentes, el investigador describe de forma explícita los procesos de decisión de los actores simulados en un nivel micro. Las estructuras emergen a niveles macro como resultado de las acciones de los agentes y su interacción con otros agentes y el entorno. Mientras que los dos primeros métodos de simulación cuentan con un alto grado de madurez y cuentan con probada efectividad tanto en los ámbitos académicos como empresariales, este último es utilizado principalmente como una herramienta de investigación académica, especialmente en ciencias sociales, economía y ecología (donde es conocida también bajo el nombre de modelado basado en individuos). En la siguiente tabla podemos ver algunas de las aplicaciones que se le dan a este tipo de modelado:

Campo	Ejemplos de aplicaciones
Ciencias sociales	Sociedades de insectos, dinámica de grupos en vuelo, propagación de epidemias, desobediencia civil
Economía	Mercado de valores, comportamiento de consumidores
Ecología	Dinámica de la población del salmón y la trucha, dinámica de uso de tierras, crecimiento de la selva
Ciencias políticas	Orígenes y patrones de violencia política, derecho del agua en países en desarrollo

Tabla I. Ejemplos de aplicaciones de simulación basada en agentes. [SA1]

A pesar de que la simulación por computadora se ha utilizado ampliamente desde la década de 1960, la simulación basada en agentes se volvió popular hasta principios de la década de 1990. Actualmente, es considerada una herramienta de modelado de simulaciones reconocida por la academia y en proceso de obtener el mismo nivel de reconocimiento en la industria. La historia del modelado basado en agentes no está bien documento, esto, probablemente se deba al hecho de que no existe un consenso generalizado sobre una definición de lo que es un agente y por tanto las opiniones con respecto a los inicios del modelado basado en agentes difieren.

La simulación basada en agentes cuenta con un enfoque *bottom-up* que es utilizado en situaciones en donde la variabilidad entre los distintos agentes no puede ser ignorada. Permite comprender cómo la dinámica de varios sistemas surge a partir de los rasgos de los individuos y de su entorno. Permite el modelado de poblaciones heterogéneas donde cada agente puede tener motivaciones e incentivos personales, además de facilitar la representación de grupos y sus interacciones.

2.4.2. Sistemas Multi-agentes

Existe un amplio conjunto de dominios de aplicación existentes que aprovechan el paradigma de agentes para el desarrollo de sistemas basados en ellos. Luck [LM1] realiza una distinción entre dos tipos principales de paradigmas de sistemas multi-agente:

- Sistemas de decisión multi-agente
- Sistemas de simulación multi-agente

En los sistemas de decisión multi-agente, los agentes participantes en el sistema deben tomar decisiones conjuntas como un grupo. Los mecanismos para la toma de decisiones conjunta pueden estar basados en procedimientos económicos, como las subastas, o en procedimientos alternativos, como la argumentación.

En los sistemas de simulación multi-agentes, el sistema es utilizado como un modelo para simular algún dominio del mundo real. Un uso típico es en los dominios que involucran varios componentes diferentes que interactúan de maneras diversas y complejas y donde las propiedades a nivel sistema no pueden ser fácilmente inferidas de las propiedades de los componentes individuales.

Un sistema multi-agente puede definirse de manera simple como una colección de agentes inteligentes diversos y heterogéneos que interactúan entre sí y entre el entorno que los contiene. Dicha interacción puede ser cooperativa, donde los agentes tratan de alcanzar un objetivo como un equipo, o bien puede ser competitiva, donde cada agente individual trata de maximizar su propio beneficio a expensas de los demás. Los agentes reciben información de otros agentes y de su entorno y contienen reglas internas que sirven para representar los procesos cognitivos de decisión a efectos de poder determinar la manera en que deben responder.

Las reglas pueden ser tan simple como una función de las entradas recibidas, o bien puede ser sumamente compleja e incorporar varios parámetros de estado internos que pudieran incluir un modelo que represente el punto de vista del agente sobre una parte del entorno, o incluso modelos psicológicos que incluyan algún tipo de personalidad y características emocionales que

influyen en la producción de comportamientos diferentes del agente bajo diferentes circunstancias. Incluso, estas reglas pueden ser previamente definidas o bien pueden modificarse durante el transcurso de la simulación para representar aprendizaje. Adicionalmente, el entorno del sistema es susceptible de recibir presión externa.

2.4.3. Arquitecturas de agentes

Existen varias maneras de diseñar la estructura interna de un agente, y para este fin, se han desarrollado diversas arquitecturas de agentes a lo largo del tiempo. De manera general, podemos clasificar las arquitecturas para agentes inteligentes en cuatro grupos principales:

- Agentes basados en lógica
- Agentes reactivos
- Creencia-Deseo-Intención
- Arquitecturas por capas

Los agentes basados en lógica son especificados a través de un lenguaje declarativo, y el proceso de toma de decisiones se lleva a cabo a través de deducciones lógicas. El razonamiento se realiza fuera del sistema, durante la compilación, en lugar de durante la ejecución del sistema. La lógica utilizada para especificar a este tipo de agentes es en esencia conocimiento de lógica modal.

La principal desventaja de esta arquitectura es que las limitaciones teóricas de este enfoque no son claramente identificables, ya que al traducir el mundo real a través de expresiones de lógica pura se corre el riesgo de dejar fuera expresiones que dada su complejidad puedan no replicarse de manera fidedigna.

Los agentes reactivos ofrecen una arquitectura basada en la descomposición de los elementos de un comportamiento inteligente de alto nivel de complejidad en varios módulos de comportamiento simple, que son organizados en capas. Cada capa implementa un objetivo

particular del agente, de tal manera que las capas superiores cuentan con un nivel de abstracción mucho mayor que las inferiores. El objetivo de cada capa engloba los objetivos de las capas inferiores. Por ejemplo, la decisión de la capa diseñada para buscar alimento toma en cuenta la decisión de la capa inferior para librar obstáculos. Esta arquitectura se utiliza a menudo cuando se requiere que los agentes sean capaces de emular comportamientos adaptativos complejos.

La arquitectura Creencia-Deseo-Intención (*Belief-Desire-Intention* o BDI por sus siglas en inglés) tiene sus orígenes en el razonamiento práctico sobre los objetivos que deben ser alcanzados y la manera de lograrlo. Las creencias son representadas por la información de cada estado, o en otras palabras, sus creencias acerca del mundo, que pueden no necesariamente ser ciertas y pueden ser modificadas en el futuro. El deseo es representado por los objetivos o situaciones que el agente debe cumplir o alcanzar. Las intenciones se representan a través de estados deliberados del agente que han sido preseleccionados como objetivos alcanzables. Esta arquitectura es frecuentemente utilizada cuando los objetivos no son constantes y por tanto se utiliza el deseo como una mejor alternativa para expresarlos.

La idea principal detrás de la arquitectura por capas es la separación de conductas sociales reactivas y proactivas. Estas capas pueden organizarse ya sea de forma horizontal, donde las entradas y las salidas se encuentran conectadas a cada capa, o bien de manera vertical donde cada entrada/salida es manejada por una única capa.

2.5. Simulación de tráfico

Champion [CH1] describe a la simulación como una herramienta eficaz utilizada para la reproducción y análisis de una gran variedad de problemas complejos, que de otra manera resultarían difíciles, caros o peligrosos de estudiar. El tráfico puede ser visto como un sistema complejo, por lo que la simulación es la herramienta indicada para el análisis de sistemas de tráfico. La simulación de tráfico es uno de los métodos más populares en la actualidad para la evaluación de nuevos esquemas de transporte enfocadas a reducir los congestionamientos. En lugar de implementar un esquema sin conocer si el resultado será o no exitoso, dicho esquema puede ser implementado a través de una simulación para determinar su efectividad. Las mejoras a

la infraestructura vial en una zona urbana son costosas, por lo que cada nueva modificación o mejora debe ser evaluada cuidadosamente en términos del impacto al flujo de tráfico. La simulación de tráfico por computadora es un método costo-efectivo para la realización de estas evaluaciones.

Un estudio realizado por la Universidad de Leeds, en Inglaterra, encontró que entre las organizaciones que utilizan las simulaciones de tráfico se pueden encontrar los centros de investigación, las autoridades gubernamentales en temas de vialidades y los constructores. La aplicación más frecuente para la simulación de tráfico es para el diseño y prueba de estrategias de control y medición de grandes esquemas de transporte.

En este mismo estudio, se identificaron cuatro usos principales de la simulación de tráfico:

- Simulación de redes que incluyen interacciones entre vehículos
- Pronósticos del flujo de tráfico en el corto plazo
- Mejoras en los modelos actuales
- Suministro de información para los simuladores de conducción de automóviles.

Entre los escenarios simulados con más frecuencia se encuentran los corredores (áreas que contienen las vías vehiculares principales) e intersecciones, seguidas de simulaciones de caminos individuales y ciudades. La gran mayoría de los usuarios utilizan la simulación para obtener información sobre las condiciones futuras en el corto plazo (1 a 5 años) a partir de una situación actual.

2.5.1 Clasificaciones de Simulación

La simulación es una rama muy activa de las ciencias de la computación, que consiste en el análisis de las propiedades de modelos teóricos o del mundo real para su estudio en la computadora. Existen varios tipos de simulación por computadora. A continuación se mencionan las más relevantes.

La simulación estocástica utiliza generadores de números aleatorios para modelar el cambio y la aleatoriedad. Es muy poco probable que dos ejecuciones de una simulación estocástica generen los mismos resultados, pero varios generadores utilizan valores “semilla” que pueden ser definidos para producir el mismo conjunto de números cada vez, lo que hace posible que los resultados sean reproducibles.

Las simulaciones determinísticas son aquellas que son predecibles, y que siempre generan los mismos resultados a partir de una entrada determinada. Los modelos determinísticos son útiles para aquellos experimentos donde los resultados necesitan ser reproducibles; sin embargo, la gran mayoría de los fenómenos del mundo real, como el tráfico, tienen cierto nivel de cambio y por tanto requieren de simulaciones del tipo estocástico.

La simulación también puede ser clasificada como continua o discreta. Los modelos continuos se representan a través de ecuaciones que utilizan variables que se corresponden con valores reales. A través de la resolución de dichas ecuaciones se puede calcular el estado del modelo en algún punto determinado.

La simulación discreta representa la realidad a través del modelado del estado del sistema y de los cambios de estado una vez que ha pasado el tiempo o ciertos eventos. Existen dos tipos de simulación discreta: modelos de tiempo discretos y modelos de eventos discretos. Los modelos de tiempo discretos son aquellos que segmentan la simulación entre ciertos intervalos de tiempo definidos. En cada intervalo, el estado del modelo es actualizado a través de funciones que describen sus interacciones. Los modelos de eventos discretos son aquellos que mantienen una cola de eventos programados para ocurrir en cierto orden definido. Cada evento representa un cambio de estado de un elemento del modelo. El simulador procesa los eventos en orden, y cada uno puede alterar la cola de eventos.

2.5.2. Simulación macroscópica y microscópica

Los simuladores de tráfico pueden ser de tipo microscópico o macroscópico dependiendo del nivel de detalle requerido.

Los simuladores macroscópicos modelan el flujo de tráfico utilizando modelos matemáticos de alto nivel provenientes de la dinámica de fluidos, por lo que son considerados como simulaciones continuas. En estos modelos, cada vehículo es tratado de la misma manera, y entre las variables utilizadas se consideran comúnmente la velocidad y la densidad de tráfico. Este tipo de simuladores carecen de la capacidad de modelar vías vehiculares complejas, características detalladas del tráfico o comportamientos diferenciados en los conductores. Las simulaciones macroscópicas son utilizadas con frecuencia para la simulación de sistemas de tráficos en áreas amplias que no requieren de un modelado detallado, como las autopistas. Este enfoque no es considerado realista debido a que en la vida real existen distintos tipos de vehículos conducidos por individuos con características diferenciadas como estilos de manejo y comportamiento frente al volante; sin embargo, tiene un tiempo de implementación menor.

Los simuladores microscópicos modelan entidades individuales de forma separada con un alto nivel de detalle, y son considerados como simulaciones de tipo discreto. Cada vehículo es monitoreado a medida que interactúa con otros vehículos y con el entorno. Usualmente, estas interacciones son dirigidas en lógica de cambio de carril (*lane-changing*) y de vehículo siguiente (*car-following*). Las reglas son definidos para controlar lo que puede y no puede hacerse en la simulación, por ejemplo, los límites de velocidad, el derecho de paso, la velocidad de los vehículos y su aceleración.

La simulación microscópica puede modelar flujos de tráfico de manera más realista que la realizada por los modelos macroscópicos, debido al nivel de detalle adicional incluido en el modelado de los vehículos individuales. Los simuladores microscópicos son utilizados de manera frecuente para evaluar nuevos controles de tráfico, así como para la realización de análisis del tráfico existente.

2.5.3. Elementos de modelado

La definición del modelo es una de las primeras etapas en la construcción de una simulación de tráfico, que implica decidir la manera en que se representarán los objetos (por ejemplo, vehículos, semáforos y peatones) en la simulación, y qué parámetros requerirá cada tipo de objeto. También

incluye determinar la manera en que se representará el entorno (caminos e intersecciones) y el efecto que tiene sobre el resto de los objetos.

Modelado de vehículos

En una simulación, los vehículos y los conductores generalmente son modelados como una sola entidad. Aunque esto es distinto en el mundo real, depende del propósito de la simulación la decisión sobre si pueden manejarse como entidades distintas o separadas. El modelado del vehículo es relativamente sencillo, ya que sólo se requieren de unos cuantos parámetros para describir sus características y comportamiento: velocidad máxima, aceleración máxima del vehículo y desaceleración. La aceleración es especialmente importante ya que afecta la manera en que se libera una cola de vehículos. Las dimensiones del vehículo a menudo son incluidas en la implementación, a efectos de poder incluir camiones y autobuses y que estos sean fácilmente distinguibles de los automóviles. Durante la simulación, la posición actual y la dirección son requeridas para mantener un monitoreo del estado del vehículo. Generalmente no se añaden detalles adicionales ya que pueden añadir complejidad al modelo sin aportar mayor realismo al mismo; sin embargo, algunos simuladores incluyen información adicional que puede ser de utilidad cuando se realizan investigaciones enfocadas a algún aspecto en particular, como las emisiones de partículas y su impacto en el ambiente.

Modelado de conductores

Algunos investigadores sugieren que los procesos de decisión que utilizan los conductores pueden clasificarse en dos tipos: *micro objetivos* y *macro objetivos*.

Los *macro objetivos* son el destino y la ruta que debe tomarse, mientras que los *micro objetivos* se refieren a las decisiones que un conductor debe tomar en cada punto de su ruta en búsqueda de alcanzar un *macro objetivo*. Los *macro objetivos* incluyen la planeación diaria y la funcionalidad asociada a la generación de una ruta. Los *micro objetivos* se refieren a decisiones relacionadas con el control del vehículo, como la velocidad deseada, los rebases y los cambios de dirección.

Cada conductor tiene diferentes estilos de manejo, que se rigen por características individuales, como la agresividad, la confianza y la experiencia de manejo. Los estilos de conducción son diseñados utilizando parámetros para representar las características más importantes del comportamiento del conductor y son utilizados como parte del proceso de razonamiento para la toma de decisiones que un conductor debe elegir en cada punto dentro de su trayecto. Algunos ejemplos de estos parámetros incluyen:

- Velocidad preferente, la velocidad que el conductor desea mantener en un camino vacío
- Tasa de aceleración preferente, que no se refiere al valor máximo que puede ofrecer el vehículo, sino a una tasa influenciada por la agresividad del vehículo y el deseo de comodidad, entre otras características
- Tasa preferente de desaceleración, contraparte de la tasa comentada anteriormente
- Espacio entre vehículos aceptable, que se refiere a la distancia que un conductor desea dejar entre el vehículo de enfrente y él mismo.

Estos parámetros son suficientes para modelar una gran cantidad de características de los conductores. Un conductor agresivo tendría una velocidad preferente alta, una tasa de aceleración y desaceleración altas, y un espacio entre vehículos mínimo. Un conductor precavido tendría valores más bien bajos y opuestos a los del conductor agresivo. Muchos simuladores incorporan todos estos parámetros para poder representar comportamientos de conductores más realistas. Para poder convertir estos parámetros en acciones, debe existir un proceso de razonamiento que es evaluado en cada punto de la simulación.

Modelado del entorno

En una simulación de tráfico, el entorno a través del cual circulan los vehículos es una red de caminos construida a partir de segmentos enlazados, nodos (cruces e intersecciones) y características de control que son usualmente parte de cada nodo. Cada enlace puede tener uno o

más carriles, y puede operar en una o dos direcciones. Los nodos pueden ser clasificados en los siguientes tipos básicos:

- Cruces no controlados sin prioridad
- Cruces con prioridad
- Glorietas
- Cruces controlados por semáforo
- Cruces a desnivel

Pueden existir enlaces especiales como las incorporaciones donde dos vías convergen en una sola. Salidas donde una vía se divide en dos, y secciones donde una salida es seguida de una incorporación.

Los modelos generalmente representan la red como una colección de enlaces unidos a través de nodos. Dichos nodos tienen propiedades tales como la ubicación (coordenadas) y el tipo de nodo. Los enlaces cuentan con propiedades como la longitud, el número de carriles, el límite de velocidad, etc.

2.5.4. Elementos de Simulación

El modelado de los vehículos, los conductores y el entorno en el que habitan no tienen ninguna utilidad a menos que puedan ser manipulados durante la simulación. Una simulación implica el uso de reglas de comportamiento para modificar el modelo a través del tiempo. Esto incluye mover cada vehículo basado en sus parámetros y las decisiones del conductor. Simular un único vehículo a lo largo de un camino recto es una tarea sencilla, ya que el vehículo aceleraría hasta la velocidad preferente definida por el usuario. A medida que se incrementa el número de vehículos, carriles y caminos, la complejidad del modelo se incrementa. Para poder controlar estos elementos se utilizan los modelos de vehículo siguiente y de cambio de carriles.

Vehículo siguiente

Los modelos de vehículo siguiente se refieren a aquellos que intentan describir la manera en que se comporta un vehículo bajo condiciones normales. Estos modelos reciben este nombre debido a que se enfocan principalmente en la relación entre un vehículo y el que le antecede en una misma vía (vehículo líder). Existe una gran variedad de modelos de vehículo siguiente en la literatura, y la gran mayoría se basan en el concepto de anti-colisión. Uno de los modelos más reconocidos es el modelo de Gipps, desarrollado en 1981. Este modelo asume que un vehículo determinado siempre busca estar en posibilidades de frenar de manera segura si el vehículo líder realiza un frenado de emergencia. Este modelo calcula la nueva velocidad del vehículo utilizando la velocidad actual de ambos vehículos (el vehículo en cuestión y su líder), la velocidad preferente del vehículo siguiente, la aceleración máxima y la desaceleración, así como el espacio entre vehículos, de tal forma que se calculan dos valores posibles para definir la velocidad del vehículo: uno basado en la velocidad preferente del conductor limitada por las capacidades del vehículo, y la otra basada en la seguridad del mismo, es decir, una velocidad que asegure que no existan colisiones. Entre estas dos opciones, se elige la menor de ellas para asegurar que ambas limitaciones son aplicadas al modelo.

Cambio de carril

Los modelos de cambio de carril se caracterizan por describir cuándo y bajo qué condiciones un vehículo cambia de carril

Los modelos de cambio de carril son aquellos que describen cuando y de qué manera un vehículo cambia de carril en una vía de varios carriles (autopista). Un escenario de cambio de carril puede representarse de acuerdo a la siguiente imagen:

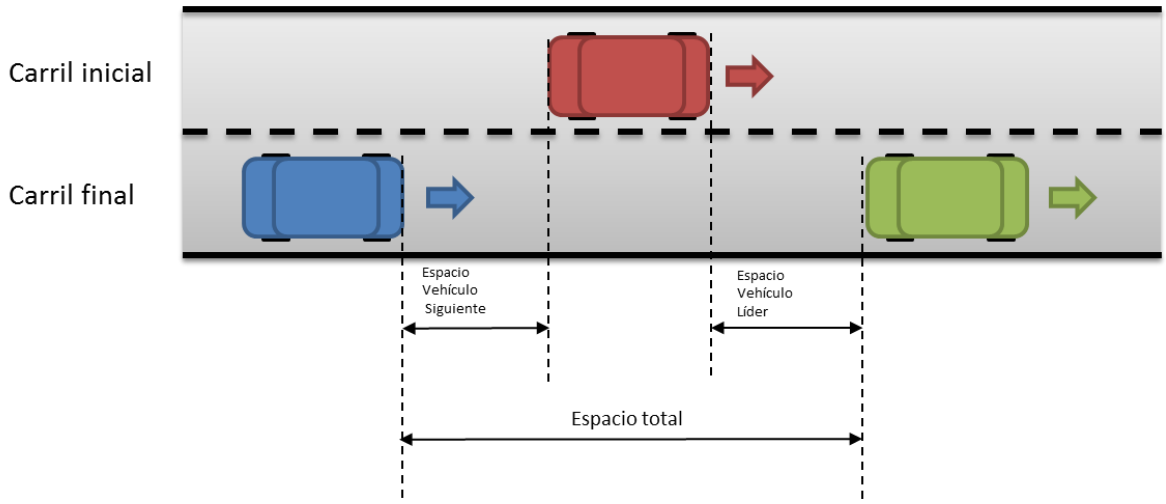


Figura 8. Consideraciones para el cambio de carril.
Elaboración propia a partir de [LA1]

El comportamiento de cada vehículo depende de los vehículos que se encuentran a su alrededor, y cada vehículo debe tomar decisiones basado en el conocimiento que posea del ambiente. Para que un vehículo decida realizar un cambio de carril, son necesarias dos condiciones:

- Que el conductor tenga un incentivo. Un incentivo puede ser que el otro carril sea más rápido, o que el conductor eventualmente tenga que realizar un giro.
- Que el cambio de carril sea seguro, es decir, que exista suficiente espacio en el carril destino para que pueda realizarse un cambio sin colisionar con los otros vehículos.

La utilización de este modelo produce un sistema de cambio de carriles regido por dos reglas: incentivos y seguridad. Si ambas reglas se cumplen, entonces ocurre el cambio de carril. El incentivo se puede cumplir si el espacio enfrente del vehículo ubicado en el carril destino es lo suficientemente amplio como para permitir un incremento inmediato de la velocidad después del cambio de carril. La seguridad puede ser determinada validando si el espacio detrás del vehículo líder en el carril destino es lo suficientemente amplio como para que pueda incorporarse al carril sin provocar que el vehículo siguiente tenga que frenar inmediatamente.

Capítulo 3

Desarrollo de la solución

Para el desarrollo de este proyecto se utilizó la metodología conocida como *Extreme Programming* (XP), la cual es una metodología ágil de desarrollo de *software* enfocada en proveer resultados al usuario de la manera más rápida posible.

Bajo el enfoque de *Extreme Programming*, en lugar de llevar a cabo las fases de planeación, análisis y diseño de manera lineal, un desarrollador XP realiza estas actividades a nivel micro durante toda la fase de desarrollo. Este proceso se lleva a cabo de forma iterativa para cada una de las partes que constituyan el sistema a construir. La mayor ventaja de este enfoque es la facilidad que proporciona para poder incorporar cambios.

3.1. Metodología de desarrollo Extreme Programming (XP)

La metodología XP está compuesta por 12 prácticas fundamentales [AE1]:

Planificación

Es la práctica que define la forma general de trabajar. Está compuesta por la planificación del "release" y por la planificación de la iteración.

En la planificación del "release" se define qué es lo que se pretende tener como producto en un periodo de 4 o 6 meses. El cliente define una gran cantidad de requerimientos, llamadas historias,

que son los que le dan mayor valor al negocio y que deben ser implementados dentro de ese periodo. Una vez definido el conjunto de historias, éstas son analizadas y estimadas por el grupo de programadores para que finalmente el cliente las ordene en función de su valor. Cuando se tienen ordenadas las historias, se procede a elegir aquellas cuya suma del tiempo de desarrollo no supere el periodo del “release".

En la planificación de la iteración, se definen las actividades para las siguientes 3 o 4 semanas. Teniendo en cuenta la capacidad productiva del grupo de desarrollo para la iteración, denominada velocidad, el cliente elige el conjunto de historias de mayor valor para que sean implementadas en la iteración planeada. A continuación, los programadores dividen las historias en tareas más pequeñas, denominadas tareas de ingeniería. Luego, cada programador elige las tareas que desea implementar, las analiza en mayor detalle y realiza una estimación de su tiempo de desarrollo. Finalmente, el cliente ordena en función de sus necesidades las historias estimadas, dejando para iteraciones posteriores aquellas que sobrepasen la capacidad productiva de la iteración.

Además de la mecánica del proceso explicada anteriormente, es importante hacer notar el aprendizaje que tienen los programadores en la realización de estimaciones al tener que repetirlas en periodos cortos de tiempo. También es importante resaltar el aprendizaje que tiene el cliente en la definición de las historias, para que sean más claras para los programadores.

Entregas frecuentes.

Las entregas frecuentes permiten que el sistema empiece con algo simple y se ponga en producción rápidamente, para luego evolucionar a través de actualizaciones e incorporación de funcionalidad frecuente. Estas actualizaciones son realizadas en base a las prioridades establecidas por el cliente durante la planificación de la iteración. Las entregas frecuentes se dividen en periodos denominadas iteraciones. Se recomienda que las iteraciones sean cortas y que no duren más de 3 o 4 semanas.

Metáfora

La metáfora es una descripción general del sistema, que se establece al comenzar el proyecto, que fortifica la integridad conceptual, ayuda a guiar el proceso de desarrollo y mantiene una visión unificada entre los actores. La metáfora determina un estándar en el vocabulario que será utilizado por los programadores y el cliente, que luego ayudará a establecer las clases y métodos del sistema.

Diseños simples

Los diseños simples hacen que los sistemas desarrollados con XP sean creados de la manera más sencilla, pero cumpliendo con la funcionalidad que el cliente especificó en el juego de la planificación. XP le resta importancia a las necesidades desconocidas y especulativas del futuro y sólo atiende las necesidades actuales del cliente. Cabe aclarar que esto no quiere decir que los diseños sean de baja calidad, sino que se empieza por lo más sencillo que funcione y luego se transforma en algo más complejo si el diseño demuestra insuficiencias. La complejidad innecesaria debe ser eliminada ni bien se descubra.

Testing continuo

El testing continuo exige que los equipos XP validen el funcionamiento del software en todo momento. XP define dos tipos de test. Por un lado, los programadores diseñan y ejecutan los test de unidad previo a la implementación, mientras que el cliente diseña y ejecuta los test de aceptación. Los test de aceptación le permiten al cliente asegurarse de que ha desarrollado la funcionalidad negociada durante el juego de la planificación. Cada funcionalidad del sistema (historia) debe tener por lo menos un test de aceptación asociado.

Refactoring

Se define como “el proceso de alterar un sistema computacional para mejorar su estructura interna sin alterar el comportamiento externo”. La incorporación de esta práctica permite que los diseños del sistema se vayan perfeccionando continuamente durante todo el proceso de desarrollo, sin atarse a un diseño preliminar rígido como en el caso de las metodologías tradicionales. A diferencia de otras metodologías, XP acepta que en realidad lo único constante es el cambio y se adapta a coexistir junto a él. Aplicando esta práctica de forma continua, XP apunta a que el software se pueda mejorar y modificar con facilidad.

Pair Programming

La programación en pares exige que toda la programación y los tests se realicen a través de un esquema maker-checker; es decir, que una persona valide la programación de otro, y viceversa.

Propiedad colectiva del código

Esta propiedad hace que ninguna porción del código tenga programadores “dueños”. Esto aumenta la velocidad de desarrollo ya que cuando se necesita algún cambio, cualquier programador lo puede hacer sin depender de otros.

Integración continua

Indica que los equipos XP deben integrar el software construido diariamente. Esto minimiza el riesgo de enfrentar severos problemas de integración, vistos en proyectos que no integran con frecuencia.

Salud laboral

Para mantener al equipo saludable, descansado y aumentar la productividad y efectividad, XP propone semanas de 40 horas de trabajo.

Presencia del cliente On-Site

Permite que el proyecto sea guiado por un individuo dedicado, con el poder de decisión necesario para determinar los requerimientos y las prioridades de entrega. El efecto de la presencia continua en el lugar de desarrollo, hace que la comunicación sea fluida, con menos necesidades de documentación por escrito y permite resolver rápidamente las dudas y decisiones que puedan aparecer.

Estilo de programación estándar

Para que un equipo pueda trabajar de forma efectiva y pueda compartir el código de forma colectiva, los programadores deben ponerse de acuerdo en establecer un estilo en común mediante una serie de reglas que permitan estandarizar el estilo de programación. Esto incluye la estandarización de nomenclaturas de variables, formato común para comentarios dentro del código, etc.

Según Kent Beck, uno de los creadores de XP, todas estas prácticas interactúan y se refuerzan mutuamente, de manera que el éxito del proceso estará dado por la puesta en marcha de todas las prácticas.

A diferencia de otras metodologías, XP no requiere que se genere ningún tipo de documentación formal a lo largo del proceso de desarrollo. Se menciona que la documentación del sistema es el propio código fuente junto a los tests que validan su funcionamiento.

La flexibilidad de esta metodología permite adaptarla a las necesidades de la organización. De esta forma, se pueden incorporar las prácticas que den resultado y modificar aquellas cuya especificación original no funcione de forma adecuada.

En XP se pueden realizar estimaciones cada vez más acertadas, debido al proceso iterativo utilizado. Esto permite predecir con mayor exactitud que otras metodologías las fechas de entrega, ayudando a los programadores y al cliente a manejar mejor los tiempos de desarrollo.

La siguiente figura compara XP con otras metodologías de desarrollo de software, destacando que en XP el análisis, el diseño, la implementación y las pruebas se realizan de forma continua.

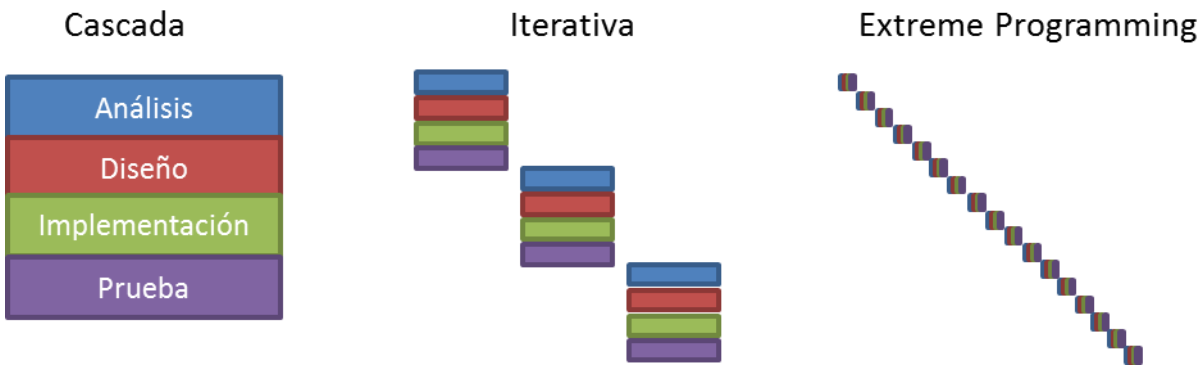


Figura 9. Estructura general de la API. Elaboración propia basada en [AE1]

Roles en la metodología XP

En XP se definen los siguientes roles:

- El **cliente** determina la funcionalidad que se pretende en cada iteración y define las prioridades de implementación según el valor de negocio que aporta cada historia. El cliente también es responsable de diseñar y ejecutar las pruebas de aceptación.

- El **programador** es responsable de implementar las historias solicitadas por el cliente. Además, estima el tiempo de desarrollo de cada historia para que el cliente pueda asignarle prioridad dentro de alguna iteración. Cada iteración incorpora nueva funcionalidad de acuerdo a las prioridades establecidas por el cliente. El programador también es responsable de diseñar y ejecutar las pruebas de unidad del código que se ha implementado o modificado.

- Una de las tareas más importantes del **tracker** consiste en seguir la evolución de las estimaciones realizadas por los programadores y compararlas con el tiempo real de desarrollo. De esta manera, puede brindar información estadística en lo que refiere a la calidad de las estimaciones que puedan ser mejoradas. Otra de las tareas que merece ser señalada, consiste en visitar a todos los programadores durante la iteración y analizar cuánto tiempo de trabajo le falta para implementar sus historias y cuánto es que se había estimado para ellas. Con esta información, pueden tener una idea global del progreso de la iteración y evaluar las acciones que se deben tomar. Con la información mencionada y muchos otros datos que el tracker colecta, se generan variados reportes estadísticos con el objetivo de mejorar el proceso.

- El **coach** es el responsable del proceso en general. Se encarga de iniciar y de guiar a las personas del equipo en poner en marcha las 12 prácticas mencionadas anteriormente. Es usualmente una persona con mucha experiencia en el desarrollo utilizando XP.

- El **mánager** o gerente se encarga de organizar las reuniones (por ejemplo planificación de la iteración, planificación del “release") , se asegura que el proceso de desarrollo se esté cumpliendo y registra los resultados de las reuniones para ser analizados en el futuro. Es de alguna forma el que responde al inversionista en lo que respecta a la evolución del desarrollo.

3.2. Planificación de la solución

Para identificar y definir los requerimientos de este proyecto, se utilizó la técnica de historias de usuario.

Una historia de usuario es una representación de un requisito de *software* escrito en una o dos frases utilizando el lenguaje común del usuario. Las historias de usuario son utilizadas en las metodologías de desarrollo ágiles para la especificación de requisitos (acompañadas de las discusiones con los usuarios y las pruebas de validación).

Cada historia de usuario debe ser limitada, esta debería poderse escribir sobre una nota adhesiva pequeña. Dentro de la metodología XP las historias de usuario deben ser escritas por los clientes.

Una práctica recomendada en la generación de historias de usuario que resulten efectivas para la definición de requisitos es utilizar la siguiente fórmula para la redacción:

"Como [QUIÉN] quiero [QUÉ] para [POR QUÉ]"

De esta manera, se obtiene mayor información sobre el requerimiento, como el rol o el tipo de usuario que lo necesita, además de una justificación que facilite la priorización de tareas una vez que se llegue a la fase de desarrollo.

Las historias de usuario son una forma rápida de administrar los requisitos de los usuarios sin tener que elaborar gran cantidad de documentos formales y sin requerir de mucho tiempo para administrarlos. Las historias de usuario permiten responder rápidamente a los requisitos cambiantes.

3.2.1. Historias de usuario

Las historias de usuario generadas para este proyecto se pueden observar en la tabla siguiente:

Como…	Quiero…	Para…
Desarrollador	Que la API sea compatible con lenguajes del Framework .Net	Poder explotarla en desarrollos que utilicen esta plataforma
Usuario de la API	Que pueda realizar la configuración a través de un archivo externo	Facilitar la modificación de los parámetros para un escenario determinado
Usuario de la API	Poder configurar los parámetros de los vehículos	Modelar vehículos con características diferentes y generar escenarios más realistas
Usuario de la API	Poder configurar los parámetros referentes a los semáforos	Generar escenarios de simulación más fidedignos
Usuario de la API	Poder configurar el volumen de tráfico por cada vía existente en el modelo	Poder representar modelos con características diferentes
Usuario de la API	Poder configurar la ubicación de los nodos que generan las vías	Generar modelos que emulen a vialidades del mundo real
Usuario de la API	Poder definir el sentido de las vías	Diseñar escenarios de simulación más realistas

Tabla 2. Historias de usuario para la API. Elaboración propia

3.2.2. Requerimientos de la aplicación

A partir de las historias de usuario, se definieron los siguientes requerimientos para este proyecto:

1. La API debe ser elaborada en un lenguaje de programación que permita integración y uso con el *framework* .Net y sus lenguajes compatibles. Para cumplir este requerimiento se utilizará el lenguaje C#, por la potencia y facilidad para el desarrollo que ofrece.
2. La API permitirá el modelado de 3 elementos básicos:
    - Una red de nodos enlazados para representar vialidades
    - Vehículos que circularán por dichas vialidades
    - Semáforos para representar elementos de control de tráfico
3. La API permitirá definir el sentido de las vialidades
4. La API permitirá definir el tipo de vehículo (automóvil sedan, vehículo de carga, autobús)
5. La API permitirá definir semáforos. Para facilitar la implementación, este semáforo tendrá solo dos estados: verde (avanzar) y rojo (alto). La duración de la luz en rojo podrá ser configurada.
6. La API permitirá definir el volumen de tráfico que transitará por cada vialidad.
7. Para la configuración de los elementos de la API se utilizará un archivo XML con cierto formato específico. La API leerá los valores del archivo XML y los utilizará como insumo para la simulación.

3.2.3. Requerimientos de *hardware* y *software*

Para la elaboración de este proyecto, se utilizaron los siguientes programas de cómputo:

Herramienta	Versión	Propósito
Microsoft Office Word	2010	Redacción del documento
Microsoft Office Power Point	2010	Elaboración de diagramas
Microsoft Visual Studio	2010 Professional	Desarrollo de la aplicación

Tabla 3. Programas de cómputo utilizados. Elaboración propia

Para el desarrollo de aplicaciones que hagan uso de esta API, se requiere contar con un ambiente de desarrollo para lenguajes compatibles con el *framework* .Net, tal como C#, Visual Basic .Net, C++, J++, etc.

Para la ejecución de aplicaciones que utilicen esta API, se necesita cubrir los siguientes requisitos mínimos [MS1]:

Software

- Sistema Operativo Windows 98 o superior
- Microsoft Internet Explorer 6.0 with Service Pack 1.
- Microsoft Data Access Components (MDAC) 2.8 for data access applications.
- Windows Installer version 3.0.

Procesadores

Escenario	Mínimo
.NET Framework redistributable	400 mega Hertz (MHz)
Kit de desarrollo de software de Windows (SDK)	600 MHz

Tabla 4. Requisitos mínimos de procesadores. Extraído de [MS1].

Memoria RAM

Escenario	Mínimo
.NET Framework	96 megabytes (MB)
Kit de desarrollo de software de Windows (SDK)	128 MB

Tabla 5. Requisitos mínimos de Memoria RAM. Extraído de [MS1].

Espacio en disco duro

Escenario	Mínimo
32-bit	280 MB
64-bit	610 MB

Tabla 6. Requisitos mínimos de Espacio en disco duro. Extraído de [MS1].

3.3. Diseño

Dado que uno de los objetivos de este trabajo de tesis es el de documentar el proceso de elaboración de la API, más que convertirse en un inventario de archivos, se mostrarán de manera general los componentes más importantes elaborados dentro de la fase de diseño. Asimismo, se presentará una visión de alto nivel de los módulos más representativos de la API.

Si el lector desea ahondar más en los detalles específicos del diseño, podrá encontrar, en los anexos de este documento, la información sobre la ubicación de los archivos fuente para un análisis más detallado.

3.3.1. Estructura general de la API

La biblioteca de clases cuenta con cuatro módulos principales que permiten el modelado de los distintos elementos necesarios para poder construir una simulación, además de un conjunto de clases de ayuda que facilitan la operación de la API. La siguiente figura nos muestra de manera general los elementos que componen esta API:

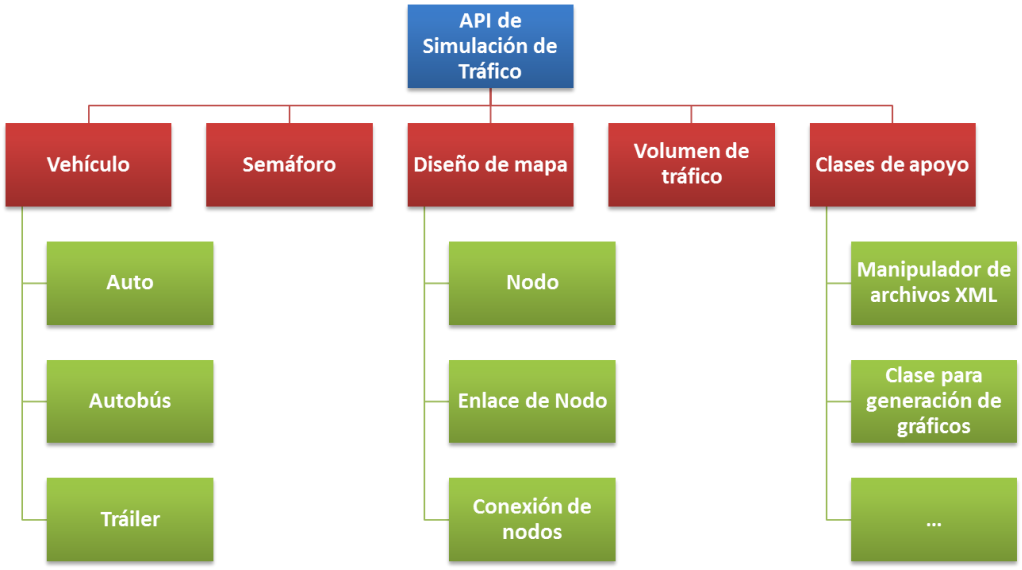


Figura 10. Estructura general de la API. Elaboración propia

La biblioteca de clases se encuentra constituida por los siguientes módulos

- Módulo de diseño de mapa. Este módulo incluye a su vez elementos para representar nodos, los enlaces entre ellos y definiciones de conexiones entre nodos para poder representar una red de vialidades con direcciones definidas a nivel camino.

- Módulo de vehículos. Permite la representación de vehículos en movimiento que circulan por las vialidades generadas por el módulo de diseño de mapa. Cuentan con parámetros que permiten definir distintos tipos de vehículos, como automóviles sedan, autobuses y camiones de carga o tráiler.

- Módulo de semaforización. Permite establecer elementos para el control de flujo vehicular a través de la colocación de objetos que obstaculizan la circulación por una vía de manera temporal, a efectos de emular la operación de un semáforo.

- Módulo de volumen de tráfico. Define las distintas rutas que puede tomar un grupo de vehículos, además de la cantidad de vehículos que circularán por cada vía.

- Clases auxiliares. Son el soporte para la operación de la API. Entre las más importantes se incluyen funciones para la lectura, procesamiento y guardado de archivos XML, clases de apoyo para el manejo de los elementos gráficos utilizados para dibujar el mapa de vialidades, los vehículos, semáforos, etc., clases para el control de la simulación, entre otras.

La siguiente figura muestra de manera ilustrativa una sección del diagrama de clases de la API que muestra algunos de los elementos mencionados:

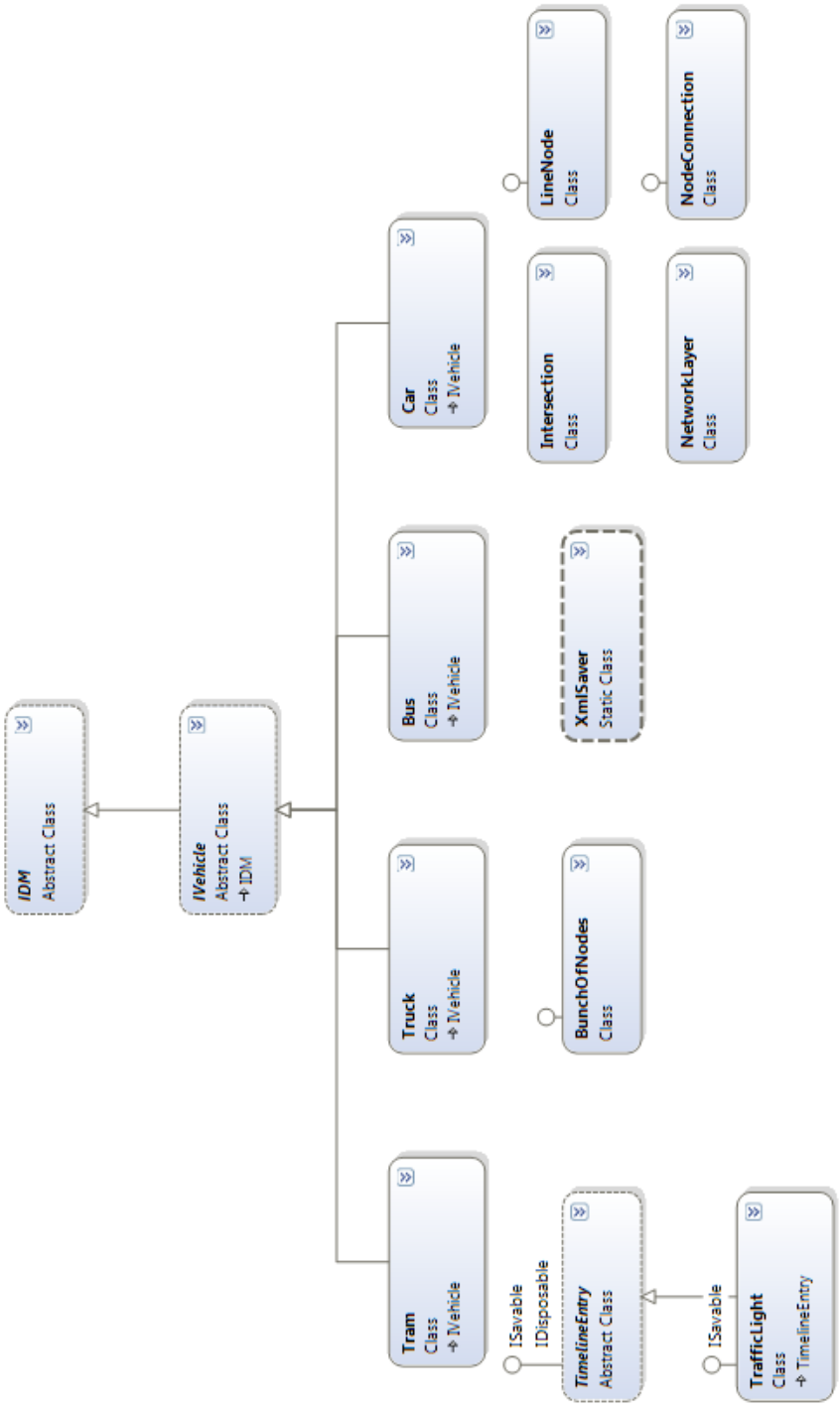


Figura 11. Sección del diagrama de clases de la API. *Elaboración propia*

3.3.2. Modelado de Procesos de Negocio

Los elementos descritos con anterioridad se comunican constantemente durante la generación de los modelos. El siguiente diagrama explica las principales interacciones:

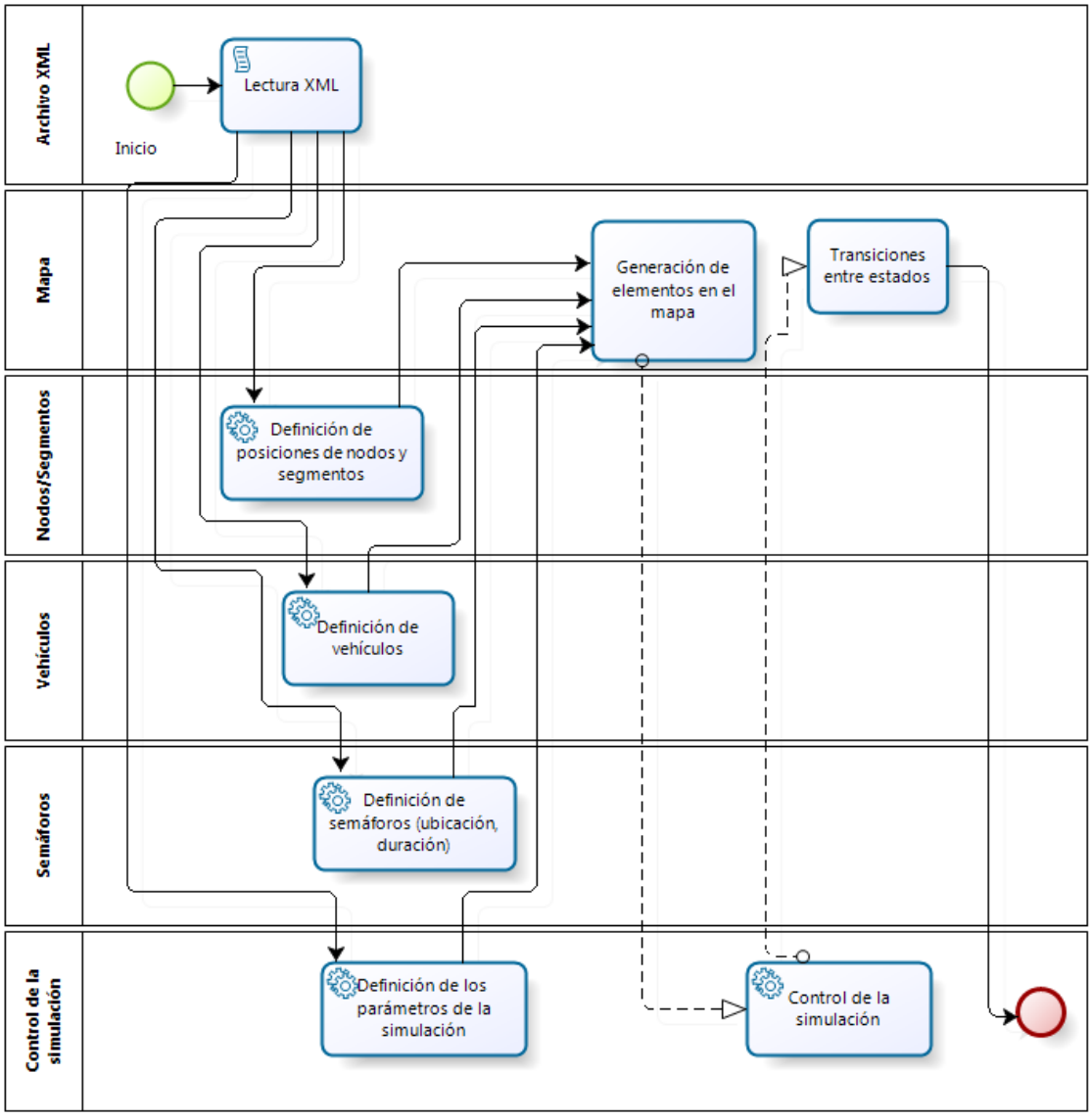


Figura 12. Interacciones entre los elementos de la API. Elaboración propia.

En términos generales, la operación de la API funciona de la siguiente manera:

- Se realiza de manera inicial un parsing sobre un archivo XML que contiene toda la información de definición y configuración de una red vehicular (nodos de la red, caminos, volúmenes de tráfico, etc.)

- Utilizando la información contenida en este XML, se crean los distintos objetos que constituyen la red vehicular: nodos de red, conexiones entre nodos, vehículos, semáforos, y se definen los atributos respectivos.

- De la misma manera, a partir de la información del archivo XML, se obtienen los parámetros generales a ser aplicados para la simulación (duración de la simulación, velocidad, flujo de vehículos por minuto para cada camino, etc.)

- Una vez que se han creado los objetos respectivos para cada elemento de la simulación, se crean elementos gráficos para representar los nodos, los caminos, los semáforos y los vehículos que estarán participando en la simulación.

- A través de un objeto timer (contador de tiempo) se lleva el control de la secuencia de pasos de la simulación y se realizan las transiciones entre estados que permiten emular el movimiento de los vehículos a través de la red de nodos (red vehicular).

3.3.3. Estructura del Archivo de Configuración

El diseño de la API considera la utilización de un archivo de configuración en formato XML cuyo propósito principal es suministrar los valores de inicio de los elementos a ser modelados durante la simulación.

Previo a la generación del modelo, se procesa el contenido del archivo XML y se utiliza la información contenida en él para definir los siguientes elementos principales:

- Configuraciones generales del modelo de simulación.
    - Duración de la simulación en Segundos.
    - Nivel de zoom inicial.
    - Escala de color para demostrar el cambio de velocidad de los vehículos
- Parámetros del Mapa de vialidades
    - Ubicación de los nodos en el plano
    - Ubicación de las líneas conectoras
    - Información sobre las conexiones entre nodos
    - Sentido de las vialidades
- Información sobre semáforos
    - Ubicación de los semáforos
    - Duración de la luz roja
- Volumen de tráfico
    - Cantidad de vehículos por tipo
    - Definición de rutas que utilizarán los vehículos para desplazarse sobre la red de nodos

Una vez que se ha procesado el archivo, se realiza la instanciación de objetos para representar vehículos, nodos, enlaces y semáforos utilizando como valores iniciales los que se encuentran en el XML.

Un archivo XML con el contenido mínimo debería contener los tags mostrados en la figura siguiente:

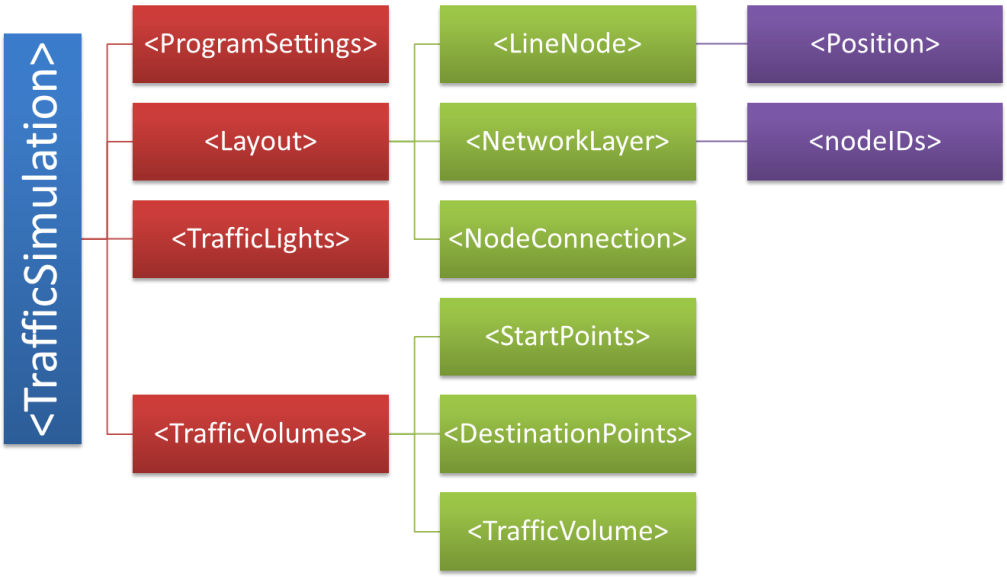


Figura 13. Estructura del archivo de configuración XML. Elaboración propia

A continuación se describe brevemente el propósito de cada tag:

Tag	Propósito
<TrafficSimulation>	Tag raíz
<ProgramSettings>	Especifica los parámetros generales que se utilizarán para la simulación, como la duración o en nivel de zoom
<Layout>	Parámetros generales del mapa
<LineNode>	Define propiedades para cada uno de los nodos que se utilizarán para crear la red vial
<Position>	Define las coordenadas de cada nodo
<NetworkLayer>	Define los enlaces entre nodos
<nodeIds>	Define los identificadores para cada nodo
<NodeConnection>	Define los segmentos que unen los nodos, así como información adicional para definir el sentido
<TrafficLights>	Define qué nodos serán utilizados para representar semáforos
<TrafficVolumes>	Define las rutas y el número de vehículos que circularán por cada vía
<StartPoints>	Define los nodos iniciales de las rutas
<DestinationPoints>	Define los nodos de destino de las rutas
<TrafficVolume>	Define el número de vehículos que circularán en cada ruta

Tabla 7. Descripción de los tags del archivo de configuración XML.. Configuración propia

En el apartado de anexos de este trabajo se podrá revisar un archivo XML de ejemplo que presenta más detalle sobre las etiquetas utilizadas.

3.4. Codificación de la API

Para la escritura del código de la API, se han utilizado los siguientes estándares de estilo de codificación para código fuente escrito en C#. La adhesión a una norma de estilo de codificación es considerada una de las mejores prácticas de la industria para hacer más eficiente el desarrollo y el mantenimiento de aplicaciones. Aunque no es exhaustiva, las directrices que se muestran a continuación representan el nivel mínimo de normalización aplicado al código de este proyecto y se encuentran basadas en las recomendaciones de MSDN [MS2], de Clint Edmonson [FR1] así como en la experiencia propia.

3.4.1. Pautas generales

Las siguientes pautas son aplicables para todos los aspectos de desarrollo en C#:

- Se debe seguir el estilo de código existente. Debe existir un esfuerzo consciente por mantener la coherencia dentro de la base de código de la aplicación.
- Escriba código de la manera más sencilla y clara posible. Asuma que otra persona siempre revisará el código.
- Prefiera clases y métodos pequeños y cohesivos antes que grandes piezas monolíticas.
- Utilice un archivo separado para cada clase, estructura, interfaz, enumeración y delegado, a excepción de aquellos anidados dentro de otra clase.
- Escriba los comentarios primero. Al escribir un nuevo método, escriba los comentarios para cada paso que el método realizará antes de escribir la primera línea de código. Estos comentarios se convertirán en los encabezados de cada bloque de código que se implementa.
- Utilice comentarios amplios y significativos dentro de cada clase, método y bloque de código para documentar el propósito del mismo.
- Marque el código incompleto con la etiqueta de comentarios `// TODO:`. Cuando se trabaja de manera simultánea con varias clases, es fácil perder el control de todas las modificaciones pendientes.
- Siempre prefiera `while` y `foreach` a otras estructuras de repetición en cuanto sea posible, ya que son más simples y sencillos de depurar.
- Use la clase `Stringbuilder` así como los métodos `Append()`, `AppendFormat()` y `ToString()` en lugar de la concatenación de cadenas (+=), ya que cuentan con un manejo más eficiente de la memoria.

- Nunca muestre información de depuración, ya sea para usted o para el usuario, a través de la interfaz de usuario (p.e. `MessageBox`). Utilice las herramientas de tracing y logging existentes en el framework para visualizar este tipo de información.

3.4.2. Diseño de clases

Las clases deben organizarse en regiones dentro de la misma aplicación utilizando el criterio definido por el arquitecto de software. Dicha organización puede basarse en el nivel de accesibilidad, tipo o funcionalidad.

Ejemplo:

```
// Organización de clases basada en nivel de acceso
class Purchasing
{
	#region Main

	#region Public

	#region Internal

	#region Protected

	#region Private

	#region Extern

	#region Designer Generated Code
}
```

Recomendaciones:

- Se debe utilizar el mismo criterio de organización para todas las clases en la aplicación.
- Si los elementos de clase asociado no son necesarios es mejor omitir la región
- La región `Designer Generated Code` creada por el diseñador de Visual Studio nunca debe ser modificada manualmente. Sólo debe contener código generado por el diseñador.

3.4.3. Indicación del alcance

Indique siempre el alcance cuando acceda a todos los miembros de clase estáticos y no estáticos. Esto asegura que se utiliza el elemento apropiado para realizar alguna función específica. La característica *intellisense* de Visual Studio se habilita automáticamente al realizar esta práctica, proporcionando un listado completo de todos los miembros de clase disponibles. Esto también ayuda a reducir el riesgo de cometer errores tipográficos.

Ejemplo:

```
string connectionString = DataAccess.DefaultConnectionString;
float amount = this.CurrentAmount;
this.discountedAmount = this.CalculateDiscountedAmount( amount,
this.PurchaseMethod );
```

Recomendaciones:

- Incluya la palabra `this` antes de todos los campos de clase, propiedades y métodos.
- Incluya el nombre de la clase antes de todos los campos estáticos, constantes, campos de clase y métodos

3.4.4. Indentación y llaves

Las líneas de código deben ser indentadas (utilizando caracteres de tabulación) en bloques que muestren el alcance relativo de la ejecución. Debe utilizarse la misma longitud de tabulación de manera consistente para todas las indentaciones dentro del código de la aplicación. Las llaves, cuando sean necesarias, deben colocarse en el renglón siguiente y alineado con la línea de código que marca el inicio del nuevo alcance de ejecución. Visual Studio .Net incluye un atajo en el teclado que permite aplicar de manera automática este formato al bloque de código seleccionado.

Ejemplo:

```
float CalculateDiscountedAmount( float amount, PurchaseMethod
purchaseMethod )
{
    // Calculate the discount based on the purchase method
    float discount = 0.0f;
    switch( purchaseMethod )
    {
        case PurchaseMethod.Cash:
            // Calculate the cash discount
            discount = this.CalculateCashDiscount( amount );
            Trace.Writeline( "Cash discount of {0} applied.", discount );
            break;

        case PurchaseMethod.CreditCard:
            // Calculate the credit card discount
            discount = this.CalculateCreditCardDiscount( amount );
            Trace.WriteLine( "Credit card discount of {0} applied.",
discount );
            break;

        default:
            // No discount applied for other purchase methods
            Trace.WriteLine( "No discount applied." );
            break;
    }

    // Compute the discounted amount, making sure not to give money away
    float discountedAmount = amount - discount;
    if( discountedAmount < 0.0f )
    {
        discountedAmount = 0.0f;
    }
    LogManager.Publish( discountedAmount.ToString() );

    // Return the discounted amount
    return discountedAmount;
}
```

3.4.5. Líneas de código extensas

Los comentarios y las líneas de código que excedan una longitud de 80 caracteres en un mismo renglón deben ser divididas e indentadas para una mejor legibilidad. Se debe tener especial cuidado para asegurar una correcta legibilidad y representación del alcance de la información en las líneas divididas. Cuando es necesario listar un alto número de parámetros, es aceptable agrupar los parámetros relacionados en una misma línea.

Ejemplo:

```
string Win32FunctionWrapper(
    int    arg1,
    string arg2,
    bool   arg3 )
```

```
{
    // Perform a PInvoke call to a win32 function,
    // providing default values for obscure parameters,
    // to hide the complexity from the caller
    if( Win32.InternalSystemCall(
        null,
        arg1, arg2,
        Win32.GlobalExceptionHandler,
        0, arg3,
        null )
    {
         return "Win32 system call succeeded.";
    }
    else
    {
         return "Win32 system call failed.";
    }
}
```

Recomendaciones:

- Cuando se divida un listado de parámetros en múltiples líneas, aplique indentación a cada nueva línea con un carácter de tabulación adicional con relación a la línea que inicia la instrucción.
- Agrupe parámetros similares en la misma línea si lo considera apropiado.
- Cuando divida comentarios en múltiples líneas, asegúrese que la indentación aplicada es igual a la aplicada en el código al que hace referencia.

3.4.6. Comentarios

Comentarios Intellisense

Se recomienda utilizar el comentarios de triple diagonal '///' para documentar las interfaces públicas de cada clase. Esto permite que Visual Studio .Net reconozca la información de cada método para construir la información de *intellisense*. Estos comentarios son requeridos para cada miembro de clase público, interno y protegido, y es opcional para los miembros privados.

Comentarios de final de línea

Utilice comentarios de fin de línea sólo para declaraciones de variables y campos de clase, a fin de documentar el propósito de la variable que se está declarando.

Ejemplo:

```
private string name = string.Empty; // Name of control (defaults to blank)
```

Comentarios de una línea

Utilice comentarios de una línea antes de cada bloque de código relacionado a una tarea particular dentro de un método que realiza una operación significativa, o bien cuando una condición significativa ha sido cumplida. Los comentarios de este estilo siempre deben comenzar con dos caracteres de diagonal seguidos por un espacio.

Ejemplo:

```
// Compute total price including all taxes
float stateSalesTax = this.CalculateStateSalesTax( amount, Customer.State
);
float citySalesTax  = this.CalculateCitySalesTax( amount, Customer.City );
float localSalesTax = this.CalculateLocalSalesTax( amount,
Customer.Zipcode );
float totalPrice    = amount + stateSalesTax + citySalesTax +
localSalesTax;
Console.WriteLine( "Total Price: {0}", totalPrice );
```

Comentarios // TODO:

Utilice los comentarios `// TODO:` para marcar secciones de código que requieren de trabajo adicional antes de ser liberados. Se deben revisar estos comentarios de manera previa a cada liberación.

Comentarios estilo C

Utilice los comentarios estilo C `/*…*/` sólo para eliminar temporalmente secciones de código grandes durante el desarrollo y la depuración. Preferentemente, evite liberar piezas que

contengan secciones de código comentado con este estilo. Si el código no es necesario, debe ser eliminado.

3.4.6. Uso de mayúsculas y nomenclatura

Siga los siguientes estándares de nomenclatura definidos por el equipo de desarrollo del Framework .Net, utilizando sólo los siguientes estilos de notación

- Notación Pascal
- Notación Camel
- Notación Upper casing

Ejemplo:

Tipo de identificador	Estilo de notación	Ejemplo
Abreviaturas	Upper	ID, REF
Namespaces	Pascal	AppDomain, System.IO
Clases y estructuras	Pascal	AppView
Constantes y enumeraciones	Pascal	TextStyles
Interfaces	Pascal	IEditableObject
Valores de enumeración	Pascal	TextStyles.BoldText
Propiedades	Pascal	BackColor
Variables y atributos	Pascal (public) Camel (private, protected, local)	WindowSize windowWidth, windowHeight
Métodos	Pascal (public, private, protected) Camel (parameters)	ToString() SetFilter(string filterValue)
Variables locales	Camel	recordCount

Tabla 8. Estilos de notación por tipo de identificador. Elaboración propia a partir de [FR1]

Recomendaciones:

- En la notación Pascal, la primera letra de un identificador, así como la primera letra de cada palabra concatenada se escribe en mayúsculas. Este estilo es utilizado para los identificadores públicos dentro de una biblioteca de clases, incluyendo namespaces, clases, estructuras, propiedades y métodos.

- En la notación Camel, la primera letra de un identificador se escribe en minúsculas, pero la primera letra de cada palabra concatenada se escribe en mayúsculas. Este estilo es utilizado para identificadores privados o protegidos dentro de una clase, parámetros enviados a métodos y variables locales dentro de un método.

- La notación Upper casing, o mayúscula, se utiliza para identificadores abreviados de cuatro caracteres o menos.

3.4.7. Elementos de programación

Clases y estructuras

Las clases y estructuras representan las "entidades" o "sujetos" de un sistema. Como tales, deben ser declaradas utilizando el formato "Sujeto" + "Calificador(es)". Las clases y las estructuras deben ser declaradas incluyendo calificadores que permitan reflejar las posibles derivaciones a partir de una clase base cada que sea posible.

Ejemplo:

```
CustomerForm : Form
CustomerCollection : CollectionBase
```

Recomendaciones:

- Utilice notación Pascal para nombrar clases y estructuras.

- Los valores predeterminados de un determinado campo deben ser definidos en la misma línea en la que son definidos. Estos valores son asignados en tiempo de ejecución justo antes de que el constructor sea creado. Esto mantiene el código para asignación de valores predeterminados en un solo lugar, lo cual resulta especialmente útil cuando una clase contiene múltiples constructores.

Interfaces

Las interfaces especifican contratos de comportamiento que deben ser implementados por clases derivadas. Para nombrar interfaces se deben utilizar sustantivos, nombres de entidades o adjetivos que expresen claramente el comportamiento declarado.

Ejemplo:

```
IComponent
IFormattable
ITaxableProduct
```

Recomendaciones:

- Utilice el prefijo "I" para los nombres de interfaces
- Utilice notación Pascal para nombrar interfaces.

Constantes

Las constantes y las variables estáticas de sólo lectura deben ser declaradas utilizando el siguiente formato: Adjetivo(s) + Sustantivo + Calificador(es)

Ejemplo:

```
public const int DefaultValue = 25;
public static readonly string DefaultDatabaseName = "Membership";
```

Recomendaciones:

- Utilice notación Pascal para nombrar constantes y variables estáticas.
- Prefiera el uso de `static readonly` en lugar de `const` para constantes públicas cada que sea posible. Las constantes declaradas utilizando son sustituidas dentro del código que las utiliza en tiempo de compilación. Utilizar `static readonly` asegura que los valores constantes son accesados en tiempo de ejecución. Esto es mucho más seguro y menos propenso a riesgos, especialmente cuando se acceden a constantes alojadas en un ensamblado distinto.

Variables, campos y parámetros

Las variables, campos de clase y parámetros deben ser declarados utilizando notación Camel, y procurando que la declaración se encuentre lo más cercano posible a su primer uso.

Ejemplo:

```
int    lowestCommonDenominator = 10;
float  firstRedBallPrice       = 25.0f;
```

Recomendaciones:

- Declare cada variable en una línea de código independiente. Esto permite la utilización de comentarios de fin de línea para documentar el propósito de cada elemento.
- Asigne valores iniciales cada que sea posible. El Framework .Net asigna en tiempo de ejecución el valor 0 o nulo para todas las variables no inicializadas de manera automática. Asignar un valor inicial ayuda a aligerar la carga de procesamiento del compilador al evitar verificar por una correcta asignación de la variable en el resto del código.
- Evite utilizar nombres como `i, j, k, temp`. Tome el tiempo que considere necesario para poder describir el uso o propósito correcto de cada variable (p.e., utilice `índice` en lugar de `i`, `swaptInt` en lugar de `tempInt`)

Manejadores de eventos

Los manejadores de eventos (Event Handlers) deben declararse utilizando el formato NombreObjeto_NombreEvento

Ejemplo:

```
private HelpButton_Click( object sender, EventArgs e )
```

Capítulo 4

Implementación de la solución

En este capítulo se documenta el resultado del proceso de codificación e implementación de la biblioteca de clases para el manejo de tráfico vehicular, a través de la revisión de un caso de estudio consistente en un escenario de simulación (un cruce vehicular controlado por un conjunto de semáforos).

En primera instancia, se realiza una descripción del visor construido para observar las simulaciones. A continuación se mostrarán un par de casos de estudio: un cruce vehicular simple de un carril, y un cruce vehicular con varios carriles y varios semáforos

4.1. Visor de simulaciones

Con el fin de poder validar el correcto funcionamiento de la API para la simulación de agentes de tráfico basado en agentes se diseñó, de manera paralela, un visor que permitiera crear elementos gráficos a partir de los elementos involucrados en la simulación.

A continuación se realiza un breve recorrido por los principales elementos de este visor.

Pantalla principal

El visor de simulaciones es una aplicación de escritorio para Windows que cuenta con un aspecto similar al siguiente:

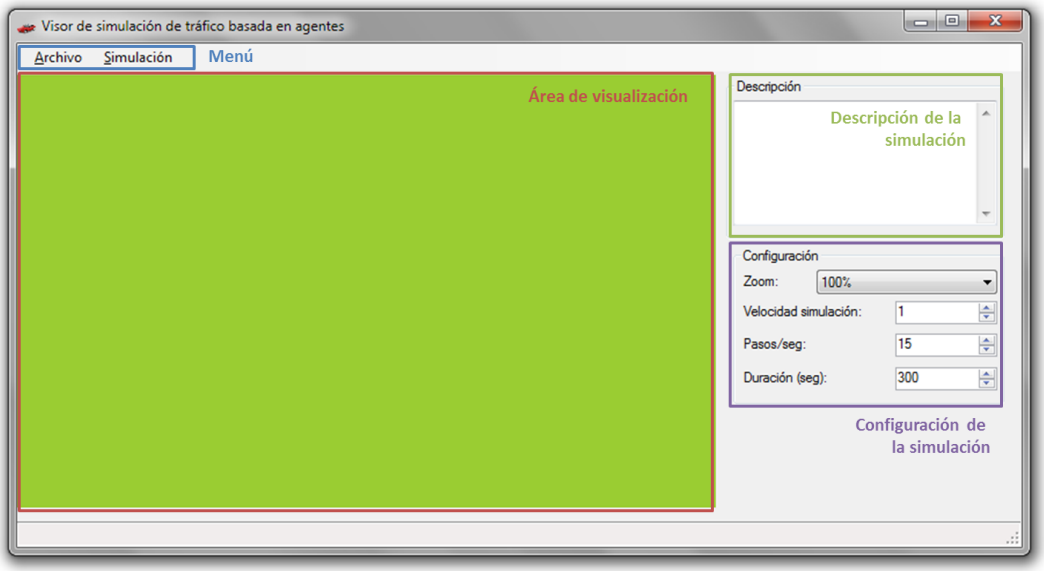


Figura 14.Pantalla principal del simulador. Elaboración propia.

La pantalla principal del visor de simulación cuenta con las siguientes secciones:

- **Menú**. A través de las opciones de menú disponibles es posible abrir los archivos XML que contienen la definición para la creación del escenario de simulación, así como para poder guardar directamente en el archivo XML las modificaciones que se realicen de manera gráfica desde el visualizador a la ubicación de los nodos y a la forma que tienen los enlaces entre nodos. Las opciones disponibles son:
    - **Archivo**. En esta opción del menú aparecen las opciones para abrir un nuevo archivo XML, así como para guardar las modificaciones que se realicen a la ubicación de los nodos de red.

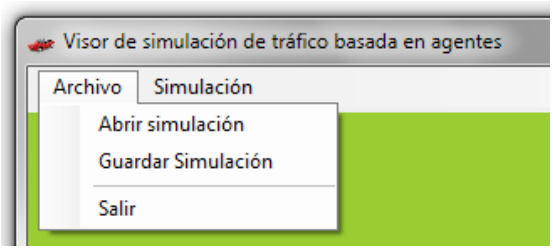


Figura 15.Menú Archivo del visor de simulación. Elaboración propia.

- Simulación. Esta opción del menú permite iniciar, detener, reiniciar o ejecutar paso por paso la simulación, así como eliminar todos los vehículos generados.

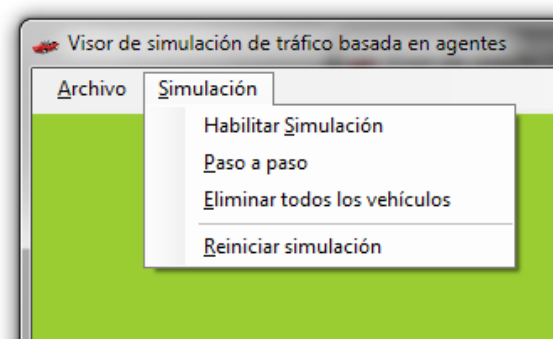
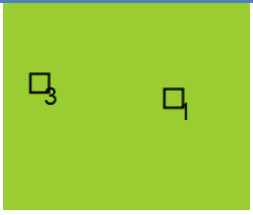


Figura 16.Menú Simulación del visor de simulación. Elaboración propia.

- **Área de simulación**. En esta sección se muestran los siguientes elementos gráficos relacionados a los componentes que participan en el escenario de simulación.

Elemento de la simulación	Gráfico
Nodos de la red **Representados por pequeños cuadros numerados que indican cada uno de los nodos de la red**	

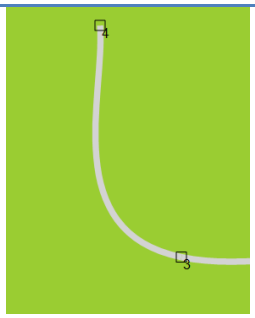
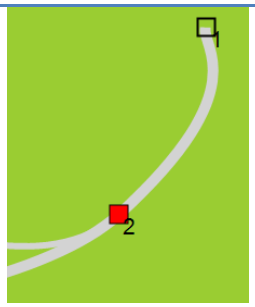
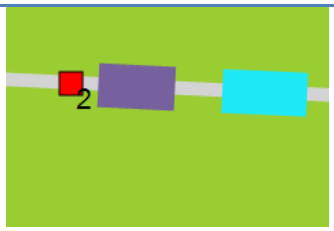
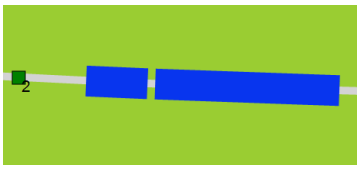
Enlaces entre nodos Representados por líneas curvas que conectan dos nodos. Representan un camino de un solo sentido por el que transitan vehículos	
Semáforos Son representados por nodos que cuentan con un atributo especial que impide el paso de vehículos durante un determinado intervalo de tiempo en el que se muestra de color rojo. **Cuando el nodo permite el paso de vehículos, se muestra de color verde.**	
Vehículos (automóviles, camiones) Son representados a través de figuras rectangulares de diversos colores que se desplazan a través de las conexiones entre nodos. Los automóviles son representados por una figura rectangular, mientras que los camiones (tráiler) son representados por dos figuras rectangulares contiguas de un mismo color que se desplazan juntas al mismo ritmo.	 

Tabla 9. Elementos gráficos de la simulación. Elaboración propia.

Una vez que termina el procesamiento del archivo fuente en formato XML que contiene las definiciones y parámetros del escenario de simulación, se generan en este panel todos los elementos gráficos anteriormente mencionados..

- **Descripción de la simulación**. Es un área de texto en la que se despliega una breve descripción del escenario en cuestión. La información que se despliega en este campo de texto se extrae del tag <infoText> contenido en el archivo XML.

- **Configuración de la simulación**. En esta sección se pueden configurar algunas de las opciones generales de comportamiento de la simulación:

    - **Zoom**. Por medio de este cuadro desplegable es posible modificar el nivel de zoom en el área de visualización en la que se despliega la red vehicular. También es posible modificar este valor utilizando el scroll del mouse.

    - **Velocidad de la simulación**. Establece la velocidad a la que se ejecuta la simulación en una escala numérica del 1 al 10.

    - **Pasos por segundo**. Define el número de pasos que se ejecutan en la simulación por cada segundo transcurrido. Para efectos de esta simulación, un paso se refiere a la transición entre un estado y el estado subsecuente para cada vehículo o semáforo involucrado en la simulación.

    - **Duración**. Establece la duración de la simulación en segundos.

4.1. Caso de estudio 1: Cruce vehicular de una vía

Para poder validar la funcionalidad de la librería, se diseñó un primer escenario que contara con las características mínimas necesarias para poder implementar una red vehicular funcional.

El escenario más sencillo considerado fue el de un cruce vehicular de dos caminos con un solo carril cada uno, como el de la figura siguiente:

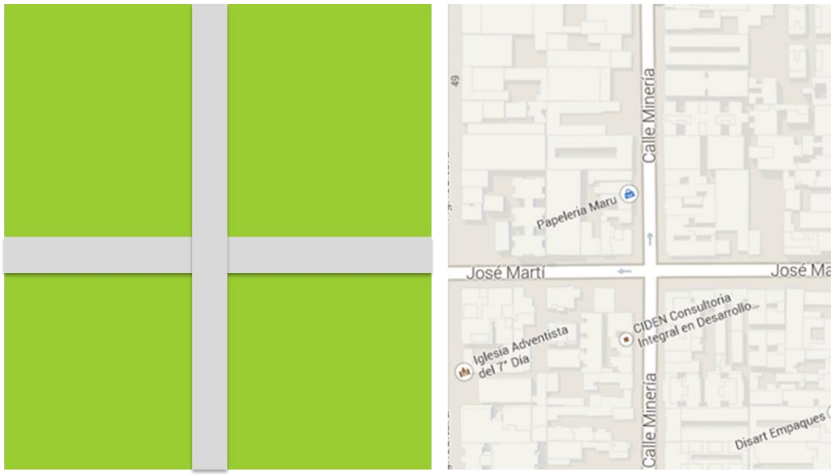


Figura 17.Ejemplo de cruce vehicular simple. Elaboración propia utilizando imágenes de Google Maps.

Para simplificar la implementación de la red vehicular y poder probar las características más importantes de la librería, se implementó un solo semáforo para una de las vialidades, y se permitieron sólo 3 rutas posibles:

- A → B
- C → D
- A → D

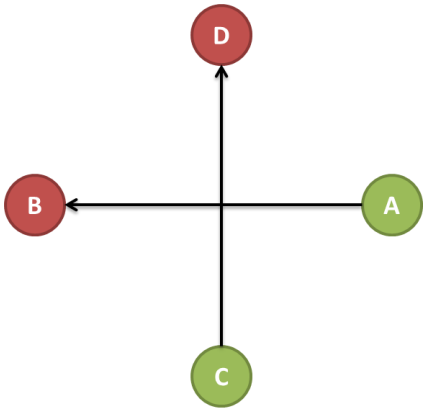


Figura 18.Rutas posibles del caso de estudio 1. Elaboración propia.

El diseño de la red vehicular, una vez que ha sido procesado por el visor de simulaciones, presentó un aspecto similar al de la siguiente figura:

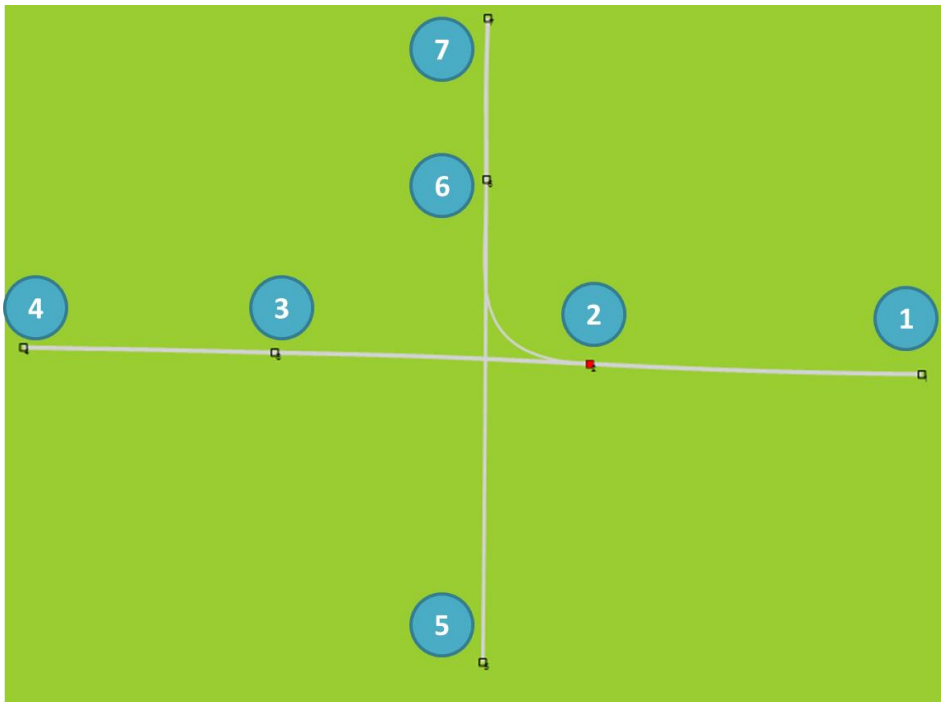


Figura 19.Red vehicular del caso de estudio 1. Elaboración propia.

Para la elaboración de esta red vial, se utilizaron 7 nodos distintos:

- Los nodos {1, 2, 3,4} para representar la vía principal.
- Los nodos {5, 6, 7} para representar la vía secundaria

Para poder representar la vuelta de los vehículos hacia la derecha (Ruta A → D) se utilizó un enlace adicional entre los nodos {2,6} para poder incluir una vía que permitiera hacer el "cambio" entre caminos vehiculares.

Adicionalmente, el nodo 2 fue habilitado como un semáforo, para poder validar el funcionamiento de esta señalización dentro de la simulación.

Para validar que los vehículos no colisionaran durante la ejecución de la simulación, se decidió incrementar la frecuencia de vehículos por minuto en cada vía a un número considerable a efectos de “saturar” las vías vehiculares e incrementar las probabilidades de un choque.

Una vez dispuestos todos los elementos anteriores, se realizó la ejecución de la simulación. La siguiente captura de pantalla muestra los primeros instantes de la ejecución del escenario de simulación:

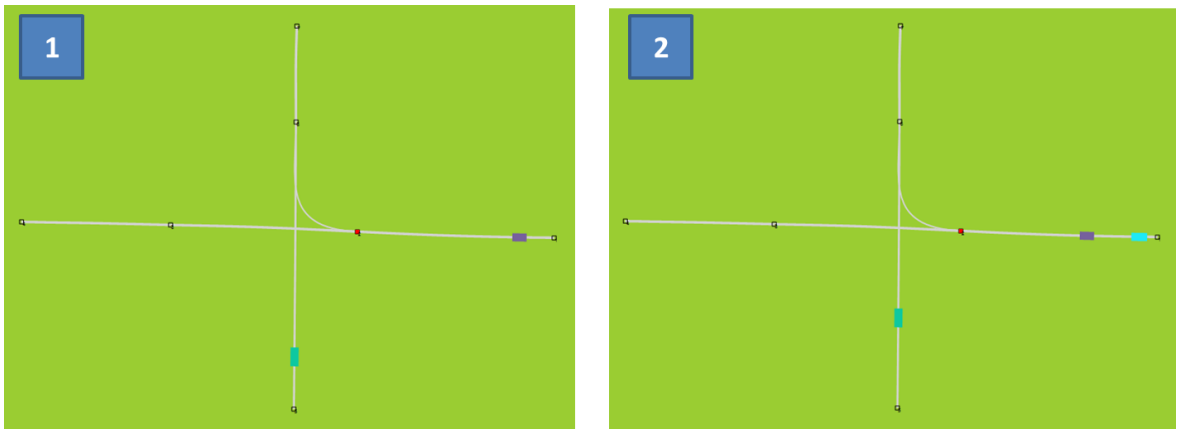


Figura 20.Capturas de pantalla del inicio de la simulación del caso de estudio 1. Elaboración propia.

A medida que continúa la ejecución del escenario, se pueden apreciar más vehículos circulando por el cruce vehicular. A continuación podemos ver un par de capturas de pantalla obtenidas aproximadamente a la mitad de la simulación:

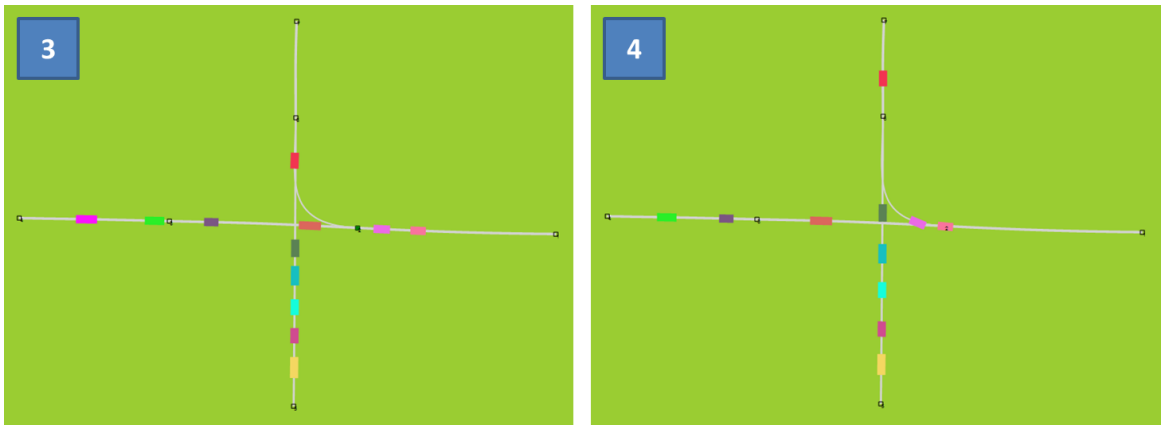


Figura 21.Capturas de pantalla a la mitad de la simulación del caso de estudio 1. Elaboración propia.

Durante la ejecución de la simulación no se presentó ninguna colisión entre los vehículos y se pudo observar que los vehículos que transitaban por la ruta indicada por los nodos 1 a 4 respetaban el semáforo del nodo 2 de acuerdo a lo esperado.

4.2. Caso de estudio 2: Cruce vehicular con múltiples carriles

El caso de estudio 1 nos permitió validar el funcionamiento utilizando un escenario con características mínimas. A efectos de poder demostrar el potencial para el diseño y la construcción de vías vehiculares con que cuentan tanto el simulador como la biblioteca de clases, se elaboró un segundo escenario con un nivel de complejidad mayor al del primer caso de estudio que nos permitiera emular de una manera más realista un escenario de una red vehicular.

El escenario propuesto para el caso de estudio 2 es, al igual que el anterior, un cruce vehicular; sin embargo, cuenta con elementos adicionales que incrementan la complejidad de la simulación:

- La vía principal cuenta con dos carriles (implementados utilizando dos vías paralelas con enlaces entre nodos que cuentan con la misma dirección) que nos permitirán observar los cambios de carril
- Se incorporan 2 grupos de semáforos para controlar el flujo de vehículos
- Una de las vías secundarias no cuenta con semaforización. Esto nos permitirá observar el comportamiento de los vehículos y de manera más específica la manera en que evitan colisiones entre otros objetos que se desplacen por las vías que cruzan.

El siguiente diagrama muestra las vías definidas en el nuevo escenario

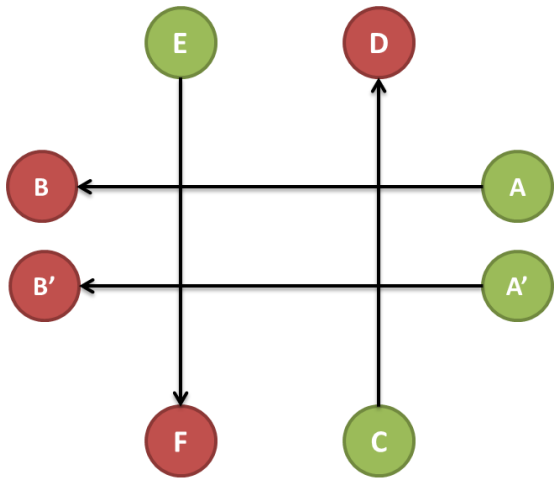


Figura 22.Rutas posibles del caso de estudio 2. Elaboración propia.

Las rutas posibles para el escenario descrito en la figura 21 son las siguientes:

- A/A' → B/B'
- A/A' → D
- C → D
- E → F
- E → B/B'

La siguiente figura muestra la red vehicular tal como aparece en el visor de simulaciones:

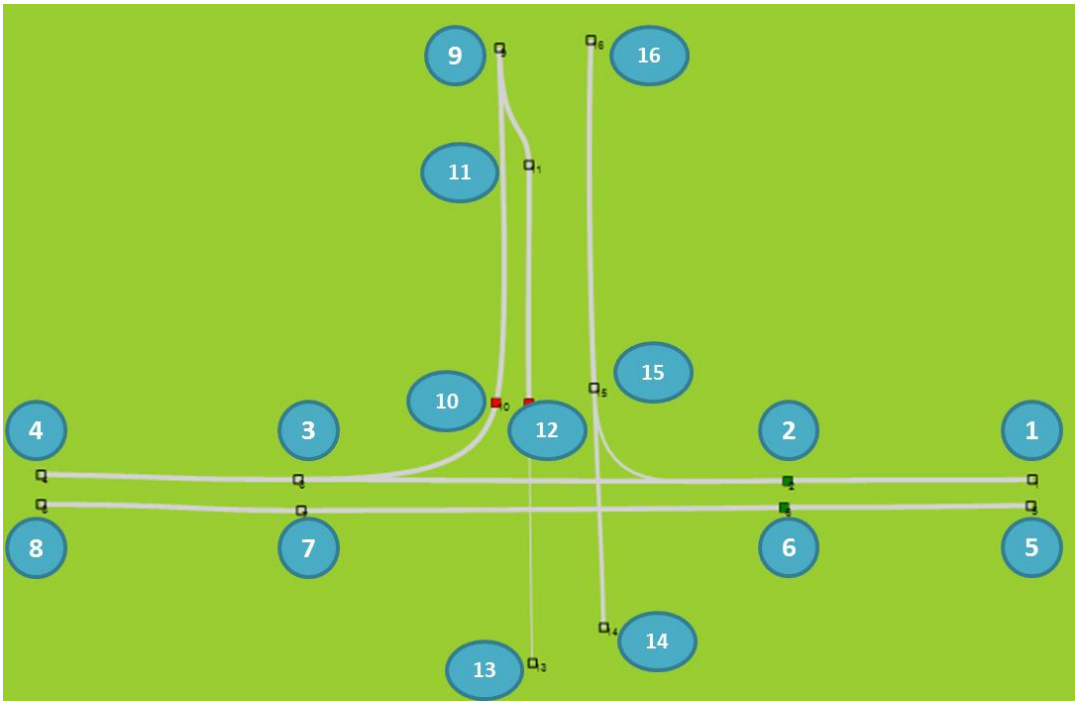


Figura 23.Red vehicular del caso de estudio 2. Elaboración propia.

Para la construcción de esta red vehicular se utilizaron 16 nodos

- Los grupos de nodos {1, 2, 3,4} y {5, 6, 7,8} representan dos vías paralelas de un mismo sentido que representan una vía principal con dos carriles.
- El grupo de nodos {9, 11, 12, 13} representa una vía de un sentido que cruza la vía principal.
- El grupo de nodos {14, 15, 16} representa una vía de un sentido que cruza la vía principal en sentido opuesto a la vía anterior.
- Se habilitaron conexiones entre los nodos {10, 3} y {2, 15} para simular cambios de vía en el cruce.

Los nodos {2, 6} y {10, 12} fueron habilitados como semáforos para controlar el flujo de vehículos en el cruce.

En las siguientes imágenes se pueden observar los instantes iniciales de la ejecución de la simulación, en las que de inmediato se podrá percibir cómo los semáforos controlan el flujo vehicular que circula por ambas vías

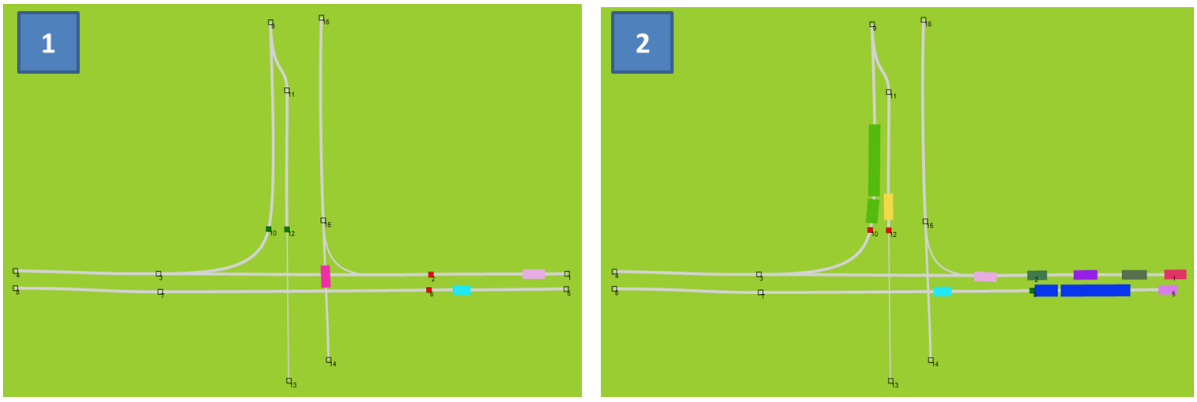


Figura 24.Capturas de pantalla del inicio de la simulación del caso de estudio 2. Elaboración propia.

En las siguientes imágenes podemos observar cómo se desarrolla la simulación.

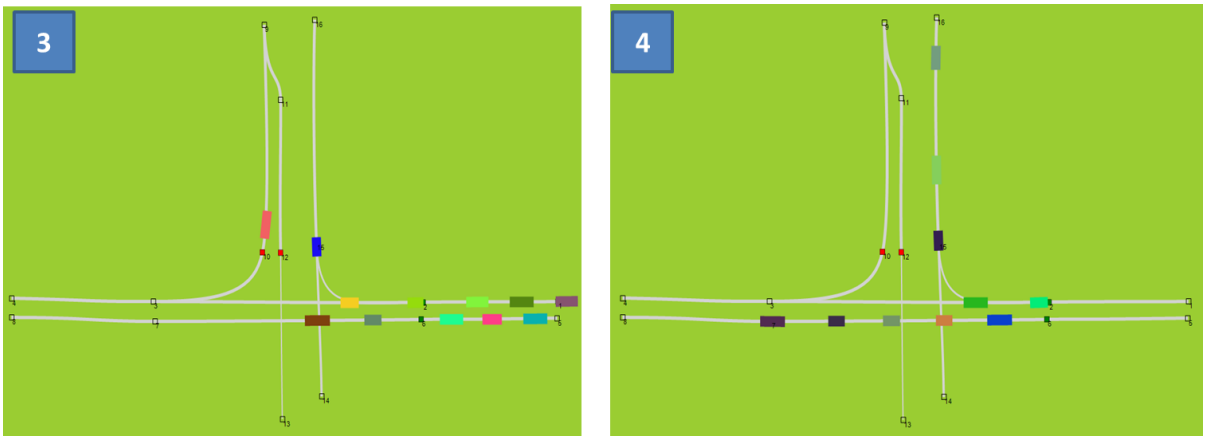


Figura 25.Capturas de pantalla a la mitad de la simulación del caso de estudio 2. Elaboración propia.

Una característica interesante en este escenario de simulación es el comportamiento de cambio de carril que presentan algunos vehículos. En la imagen siguiente, podemos observar cómo un vehículo, ante la luz roja, busca una mejor posición en la vialidad cambiando de carril.

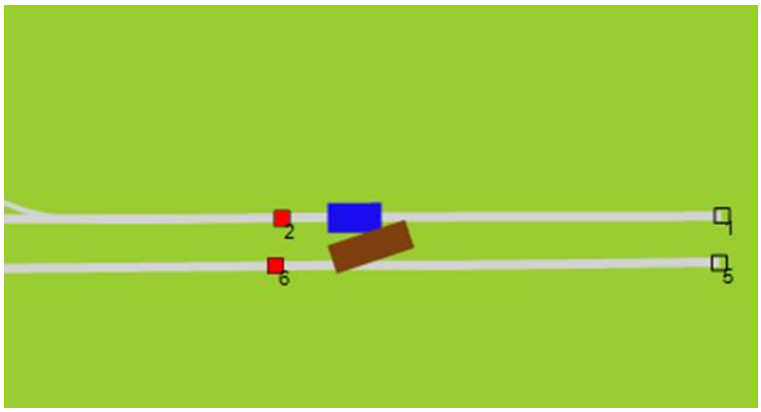


Figura 26.Cambio de carril de un vehículo. Elaboración propia.

4.3. Caso de estudio 3. Autopista con incorporaciones, salidas y cruces en vías secundarias

Los escenarios presentados en los dos casos de estudio anteriores nos permitieron validar la funcionalidad de la biblioteca de clases a través de la implementación de redes vehiculares de tamaño pequeño con una estructura de baja complejidad y que contaban con condiciones controladas.

Sin embargo, la biblioteca, así como el visor, son capaces de implementar escenarios de mayor complejidad de acuerdo a las necesidades del usuario. En este sentido, la API puede ser utilizada como la base sobre la cual se pueden construir simulaciones con una complejidad que puede ir desde la implementación de cruces viales muy sencillos (como los implementados en los dos casos de estudio previos) hasta la construcción de mapas vehiculares completos correspondientes a ciudades de tamaño pequeño a mediano.

Para poder ejemplificar lo anterior, el caso de estudio 3 se centrará en la construcción de un escenario constituido por una sección de una red de vías vehiculares que tenga un nivel de similitud mayor a aquellas que podríamos encontrar en ciudades o áreas urbanas.

De manera más específica, el escenario de este caso de estudio se basará en la construcción de una red vehicular que represente una sección de una red vehicular con los siguientes elementos principales:

- Una sección de una vía principal de cuatro carriles (Vía A).
- Un cruce secundario de 4 carriles que se incorpora a la vía principal seguido de una salida de 2 carriles (Vía B).
- Un conjunto de vías secundarias conformadas por una vía recta de dos carriles que viaja en sentido opuesto a la vía principal y que tiene una intersección con la salida de ésta (Vía C).
- Un camino de un solo carril que se incorpora a la vía secundaria (Vía D)

Al igual que en los casos anteriores, se han incluido elementos de semaforización para poder controlar el flujo vehicular. Las incorporaciones secundarias, por el contrario, no cuentan con semáforos a fin de que podamos apreciar cómo es que la lógica incluida dentro de los vehículos permite que éstos circulen por las vías y lleguen a su destino evitando colisiones.

La siguiente figura muestra un diseño a alto nivel de las vialidades anteriormente mencionadas:

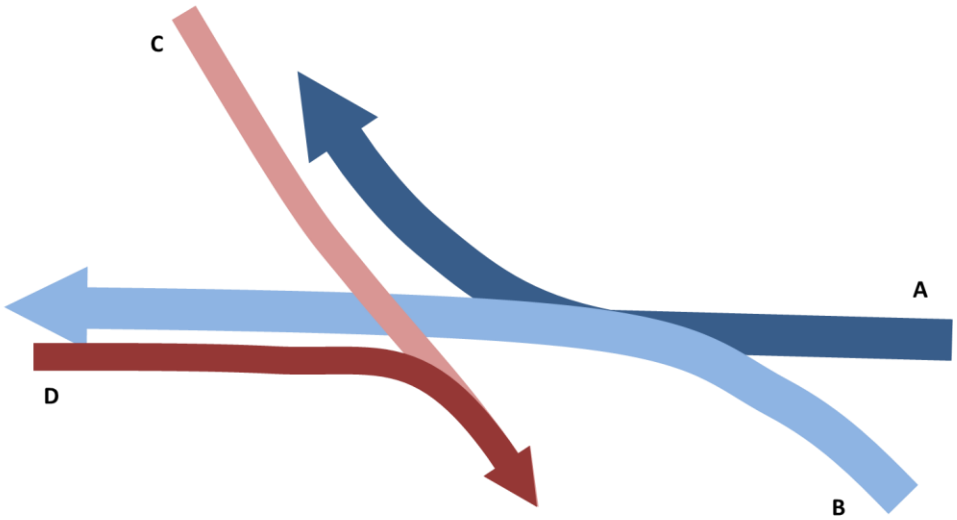


Figura 27.Diagrama de alto nivel de las vialidades del caso de estudio 3. Elaboración propia.

Se han considerado 6 grupos de semáforos que se encuentran dispuestos en las siguientes ubicaciones:

- Sobre la vía A, justo antes de la incorporación de la vía B (control de la incorporación a la vía A).
- Sobre la vía B, justo antes de la incorporación a la vía A (control de la incorporación a la vía A).
- Sobre la vía B, después de la salida de la vía A y antes del cruce con la vía C (control de la salida de la vía B y cruce con vía C).
- Sobre la vía C, antes del cruce con la vía B (control de la salida de la vía B y cruce con vía C).
- Sobre la vía D, antes de la incorporación a la vía C (control de la incorporación a la vía C).
- Al final de la vía C.

Para efectos de esta simulación, se han definido las siguientes rutas, de acuerdo a la figura siguiente:

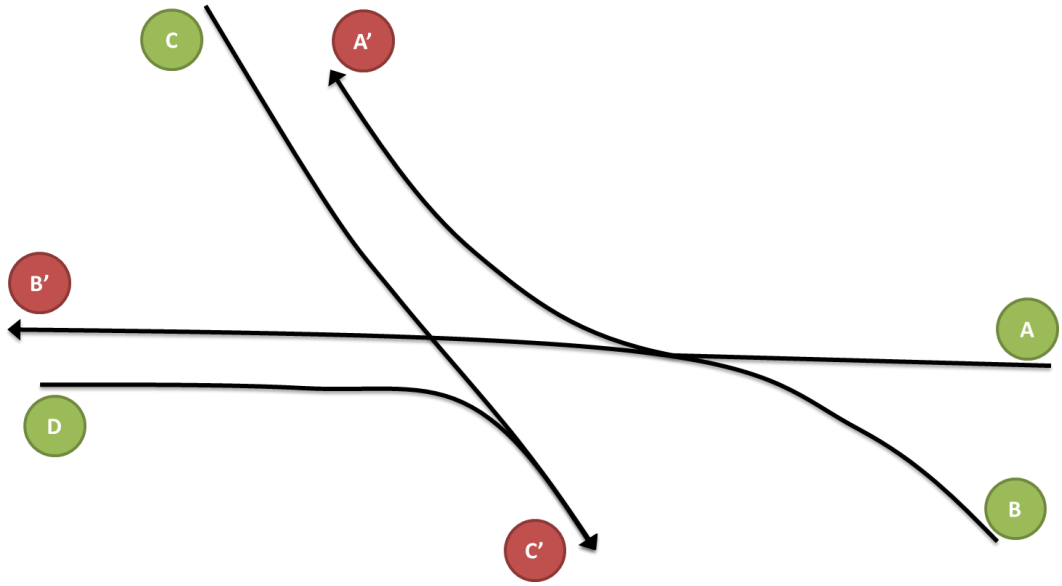


Figura 28.Rutas posibles del caso de estudio 3. Elaboración propia.

- Ruta 1: A → A'
- Ruta 2: A → B'
- Ruta 3: B → A'
- Ruta 4: B → B'
- Ruta 5: C → C'
- Ruta 6: C → B'
- Ruta 7: D → C'

En la siguiente figura podemos apreciar la red vehicular en referencia tal como se puede apreciar en el simulador:

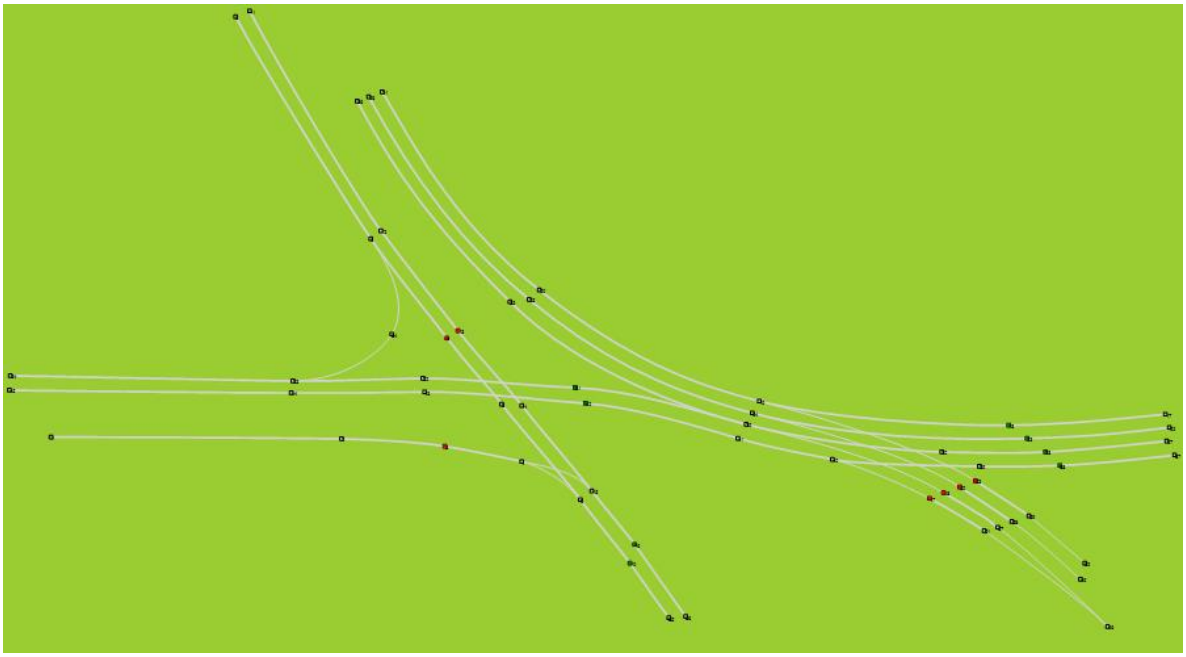


Figura 29.Red vehicular del caso de estudio 3. Elaboración propia.

Para su construcción se utilizaron 59 nodos, dispuestos de la siguiente manera:

- 16 nodos para la vía A
- 19 nodos para la vía B
- 5 nodos comunes entre las vías A y B
- 12 nodos para la vía C
- 1 nodo para la incorporación de la vía C a la vía B
- 4 nodos para la vía D
- 2 nodos comunes entre las vía C y D

Cabe mencionar que 15 del total de 59 nodos utilizados para la construcción de la vía vehicular han sido marcados como semáforos para poder controlar el flujo vehicular.

En las siguientes imágenes podremos observar dos capturas de pantalla efectuadas en dos instantes distintos dentro de la ejecución de la simulación, en donde podremos apreciar la circulación de los vehículos por la red vehicular:

Figura 30.Captura de pantalla a la mitad de la simulación del caso de estudio 3. Elaboración propia.

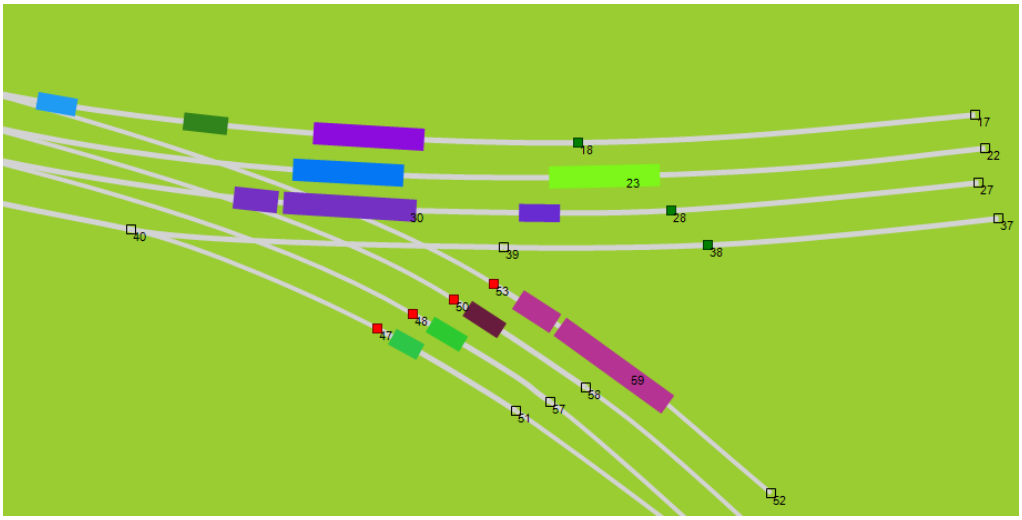


Figura 31.Captura de pantalla a la mitad de la simulación del caso de estudio 3. Elaboración propia.

CONCLUSIONES

Durante las últimas décadas hemos sido testigos de un progreso sin precedentes en el campo de la informática. La miniaturización y el dramático descenso en los costos del *hardware*, la evolución de las tecnologías de conectividad y la masificación del uso y consumo de teléfonos inteligentes (*smartphones*), entre otros varios avances, literalmente han sacado a la computación de los grandes laboratorios reservados para científicos y personal técnico altamente especializado que eran tan comunes en la década de 1970 y la han colocado al alcance de cualquier persona que ahora cuenta con la capacidad de poder interactuar a través de Internet con todo el mundo utilizando tan sólo un dispositivo que cabe en la palma de la mano.

Bajo este contexto, en el que el cómputo móvil se ha filtrado prácticamente en todos los ámbitos de la vida cotidiana, es posible comenzar a dilucidar dos escenarios claramente distinguibles entre sí:

- Un primer escenario ubicado en el mundo convencional de las bases de datos corporativas, los servidores, las aplicaciones centralizadas y las redes de computadoras estáticas, en el que los servicios de *software* y los distintos componentes continuarán implementándose utilizando tecnologías convencionales tal como lo hemos visto desde hace varios años.

- El segundo escenario basado en conexiones inalámbricas en donde las redes se conecten de forma espontánea y los servicios de internet sean siempre cambiantes.

- Los consumidores de los servicios descritos en este último escenario incluirán no sólo a seres humanos, sino a agentes de *software* que sean capaces de:

- Representar a su vez a consumidores y proveedores humanos y de *software*

- Encapsular la funcionalidad de los sistemas legados de software

- Representar a dispositivos itinerantes que enfrentan entornos con características fácilmente modificables

- Encapsular protocolos de coordinación entre pares, incluidos otros agentes.

Debido a lo extenso del campo de los agentes de software, este trabajo de tesis se centró de manera particular en la simulación de tráfico vehicular basada en agentes de software, a través de la implementación de un modelo de simulación sencillo que sirviera para mostrar las bondades de este enfoque para la construcción de aplicaciones de *software*.

Para facilitar la implementación, se creó una biblioteca de clases con el propósito de encapsular la funcionalidad del modelo y facilitar la configuración y parametrización del mismo a través de un archivo externo de configuración.

La biblioteca de clases, así como los demás elementos necesarios para hacer funcional el modelo se programaron utilizando C#, un lenguaje compatible con el marco de desarrollo de aplicaciones de Microsoft conocido como .Net Framework, y que conjuga potencia y facilidad de uso, además de facilitar tareas como la manipulación de archivos XML o la creación de elementos gráficos para la representación visual del modelo.

Adicionalmente, al realizar el desarrollo con un lenguaje compatible, aseguramos que la biblioteca de clases para la simulación pueda ser reutilizado y explotado por cualquier otro lenguaje compatible con el Framework .Net, como Visual Basic .Net, Visual C++ o Delphi, por mencionar algunos.

Para la validación de la construcción del modelo, se construyeron tres escenarios de complejidad creciente:

- Un cruce vehicular de dos vías de un solo carril y un solo sentido.

- Un cruce vehicular de una vía principal con 2 carriles y dos vías de un solo carril.

- Una sección de una autopista de 4 carriles con una incorporación de 4 carriles, una salida de 2 carriles y un par de vías secundarias.

Para los escenarios anteriormente descritos, se definieron objetos para representar vehículos particulares y camiones de carga, y se definieron rutas por las cuáles debían transitar estos vehículos. Además, se incorporaron elementos de semaforización que permitieran proporcionar un cierto nivel de control sobre el flujo vehicular.

Una vez dispuestos dichos elementos, se ejecutaron las simulaciones de forma continua y sin interrupciones a efectos de poder apreciar el comportamiento de dichos vehículos y la manera en que se desplazaban sobre las vías definidas para tal fin a través de la creación de enlaces entre nodos.

Durante la ejecución de todas las simulaciones, fue posible observar varias características de comportamiento importantes:

- Los vehículos siempre se desplazaron sobre los enlaces entre nodos (utilizados para representar carriles)
- Los vehículos respetaron las señales de semaforización, avanzando siempre que se encontraban en verde y deteniéndose al cambiar a luz roja
- El desplazamiento de los vehículos se realizó sin que ocurrieran colisiones entre otros vehículos, e incluso en los cruces no controlados por semáforos.

Dichas características nos permiten afirmar que los escenarios construidos con esta biblioteca de clases cumplen con las características mínimas que se requieren para poder representar con cierto nivel de “realismo” los escenarios de la vida real a través de redes vehiculares.

Si bien no están presentes elementos más sofisticados que se pueden encontrar en otros simuladores de tráfico vehicular, es posible considerar esta API, así como el visor de simulaciones que lo acompaña, como la base sobre la cual se pueden construir escenarios de prácticamente cualquier nivel de complejidad, y poder incorporar las características y variables requeridas para satisfacer las necesidades del usuario.

Dado lo anterior, el funcionamiento correcto del simulador nos permite afirmar que la construcción de aplicaciones bajo el enfoque basado en agentes de *software* es una alternativa real para la resolución de problemas relacionados con la simulación de tráfico vehicular y proporciona resultados efectivos que pueden ser de gran utilidad tanto en los ámbitos académicos como en la industria.

Trabajos futuros.

El desarrollo de este modelo de simulación representa la base para diversos proyectos que podrían ser implementados extendiendo las capacidades del mismo. A continuación se comparten un par de ideas para ampliar el alcance de este trabajo.

- Se podría implementar el uso de mapas geográficos disponibles a través de diversas APIs (Google Maps, OpenStreetMap, Bing Maps) para poder automatizar la creación de la red vehicular basada en vialidades reales. Bajo este contexto, incluso se podrían agregar capas de visualización para poder "mostrar" imágenes satelitales de la red vial e imprimir mayor realismo al modelo.

- Los vehículos implementados a través de la biblioteca de clases cuentan con lógica incorporada de modelos que permiten prevenir la colisión entre los mismos. Si bien esta característica permite imprimir mayor "realismo" a la simulación, la incorporación de comportamientos correctivos, como avanzar en reversa ante un bloqueo inminente, podrían ayudar a mejorar la calidad general de la librería.

Bajo esta misma línea, podría utilizarse este proyecto como base para la simulación de propuestas de nuevas vialidades, y proporcionar parámetros de entrada más acordes con la realidad para poder obtener un pronóstico de las cargas vehiculares existentes y probables utilizando estas nuevas vialidades, como un método para poder evaluar la funcionalidad de las propuestas antes de que sean construidas.

BIBLIOGRAFÍA

[AL1] ALGERS, S; et-al. *Review of Micro-Simulation Models*. Leeds, Reino Unido: Institute for Transportation Studies, University of Leeds. 1997

[BA1] BALL, G; et al. “Lifelike Computer Characters: the Persona project at Microsoft Research”. En: *Software agents*. Estados Unidos: MIT Press Cambridge, 1997

[BR1] BRADSHAW, Jeffrey. “An introduction to software agents”. En: *Software agents*. Estados Unidos: MIT Press Cambridge, 1997

[BU1] BURMEISTER, B et al. “Agent-Oriented Techniques for Traffic and Manufacturing Applications: Progress Report”. En: *N.R. Jennings and M.J. Wooldridge (eds.), Agent Technology*. Springer-Verlag: Berlin, Alemania, 1998, p. 161-174.

[CA1] CWALINA, Krzysztof; ABRAMS, Brad. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .Net Libraries*. 2a. ed. Estados Unidos: Pearson Education, 2009. 421 p.

[CH1] CHAMPION, A; et-al. “Traffic Generation with the Scanner II simulator: towards a multi-agent architecture”. En: *DSC’99: Proceedings of the first Driving Simulation Conference*, 1999

[VP1] VAN DYKE PANURAK, H. “Manufacturing Experience with the Contract Net”. En: *Distributed Artificial Intelligence*. Londres, 1987, p. 285 – 310

[FG1] FRANKLIN, Stan; GRAESSER, Art. "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents". En: *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*, Springer-Verlag, 1996.

[GA1] GARCÍA ALONSO, Daniel. *Introducción al estándar FIPA*. Madrid, España: Universidad Complutense de Madrid, Departamento de Sistemas Informáticos y Programación, 2000. UCM-DSIP 98-00 Versión 1.0. 52 p.

[GD1] GILBERT, D; et al. *IBM Intelligent Agent Strategy*. Estados Unidos. IBM Corporation working paper, 1995

[HA1] HAYES-ROTH, B; et al. "Guardian: A Prototype Intelligent Agent for Intensive-Care Monitoring". En: *Proceedings of the Twelfth National Conference on Artificial Intelligence*. Washington, Estados Unidos, 1994

[JE1] JENNINGS, N.R.; et al. *ADEPT: Managing Business Processes using Intelligent Agents*. Londres, Inglaterra, 1996

[LA1] LANSDOWNE, Andrew. Traffic Simulation using Agent-Based Modelling. Inglaterra: University of the West of England, 2006

[LL1] LJUNBERG, Magnus; LUCAS, Andrew. "The OASIS Air Traffic Management System". En: *Proceedings of the Second Pacific Rim International Conference on Artificial Intelligence*, PRICAI 92. Seúl, Corea

[LM1] LUCK, M; MCBURNEY,P; et al. *Agent technology: computing as interaction (a roadmap for agent based computing)*. Liverpool, Reino Unido. 2005.

[LM2] LEAL, L; MACÍAS, D; VILARIÑO, D. "Teoría de Agentes y sus Aplicaciones". En: *Semana de la Informática 2007 tu contacto de actualización*. Puebla, México, 2007

[MA1] MAES, P. "Agents that Reduce Work and Information Overload". En: *Communications ACM, 37*, Nueva York, Estados Unidos. Julio, 1994. P. 31 – 40.

[CM1] CASTELLS, Manuel. *The Rise of the Network Society*. 2ª ed. Reino Unido: Wiley-Blackwell, 2010.

[MP1] MAES, P.; CHAVEZ, A.; et al. "A Real-Life Experiment in Creating an Agent Marketplace." En: *Software Agents and Soft Computing, Towards Enhancing Machine Intelligence*, editado por Hyacinth S. Nwana y Nader Azarmi, Springer, 1996.

[NW1] NWANA, Hyacinth S. *Software agents: an overview*. Reino Unido: BT Laboratories, Martlesham Heath, Advanced Applications & Technology Department, Intelligent Systems Research, 1996.

[RN1] RUSSELL, Stuart; NORVIG, Peter. *Artificial Intelligence A Modern Approach*. 3ª ed. Estados Unidos: Pearson Education, 2010. 1132 p.

[SA1] SIEBERS, Peer-Olaf ; AICKELIN, Uwe Aickelin. "Introduction To Multi-Agent Simulation". En: *Encyclopedia of Decision Making and Decision Support Technologies*. Reino Unido: University of Nottingham, School of Computer Science & IT (ASAP), 2008.

[WJ1] WOOLDRIDGE, Michael; JENNINGS, Nick. Software Agents. *IEEE Review*, Enero 1996, p. 17-20

[JN1] JENNINGS, Nicholas; SYCARA, Katia; WOOLDRIDGE, Michael. "A Roadmap of Agent Research and Development." En: *Autonomous Agents and Multi-Agent Systems.* Estados Unidos, Boston, Kluwer Academic Publishers, 1998

[GV1] GARCIA-VALDECASAS, José Ignacio. “La simulación basada en agentes: una nueva forma de explorar los fenómenos sociales”. En: *Reis. Revista Española de Investigaciones Sociológicas*, núm. 136, octubre-diciembre, 2011, pp. 91-109, Centro de Investigaciones Sociológicas, España.

[AE1] ERLIJMAN, Ariel; GOYÉN, Alejandro. *Problemas y soluciones en la implementación de extreme programming. Memoria de Grado.* Universidad Católica del Uruguay Dámaso Antonio Larrañaga, Facultad de Ingeniería. Montevideo, Uruguay, 2001

[FE1] FENN, Jackie. *When to leap on the hype cycle*. Estados Unidos: Advanced Technologies and Applications, Gartner Research, 1995

[CH2] CHEN, Bo; CHEN, Harry H. “A Review of the Applications of Agent Technology in Traffic and Transportation Systems”. En: *IEEE Transactions on Intelligent Transportation Systems*, vol 11, núm 2. Estados Unidos, Junio, 2010.

Referencias de Internet

[MC1] Succeeding with Agile - Mike Cohn's Blog:
Advantages of the “As a user, I want” user story template
http://www.mountaingoatsoftware.com/blog/advantages-of-the-as-a-user-i-want-user-story-template

[MR1] Microsimulation of Road Traffic Flow
http://www.traffic-simulation.de/

[MS1] MSDN. Requisitos del sistema para .Net Framework 2.0
http://msdn.microsoft.com/es-mx/library/ms229070(v=vs.90).aspx

[UE1] Understanding the Extreme Programming Life Cycle Phases
http://www.brighthubpm.com/methods-strategies/88996-the-extreme-programming-life-cycle/

[WK1] Wikipedia. Microsoft .Net
http://es.wikipedia.org/wiki/Microsoft_.NET

[WK2] Wikipedia. Historias de Usuario
http://es.wikipedia.org/wiki/Historias_de_usuario

[WK3] Wikipedia. Software Framework
http://en.wikipedia.org/wiki/Software_framework

[WK4] Wikipedia. Application Programming Interface
https://en.wikipedia.org/wiki/Application_programming_interface

[WK5] Wikipedia. Refactorización
https://es.wikipedia.org/wiki/Refactorizaci%C3%B3n

[MO1] Mono Project. Compatibility
http://www.mono-project.com/Compatibility

[LS1] Curso .Net: El .Net Framework – parte II
http://lasoto.wordpress.com/2011/08/01/curso-net-el-net-framework-parte-ii/

[MS2] MSDN. Instrucciones de diseño para desarrollar bibliotecas de clases
http://msdn.microsoft.com/es-es/library/ms229042(v=vs.85).aspx

[FR1] Free C# and VB Coding Standards Reference Documents
http://www.notsotrivial.net/blog/post/2008/12/Holiday-Goodie-Bag-Free-C-and-VB-Coding-Standards-Reference-Documents.aspx

ANEXO I. Ubicación del código fuente.

El proyecto de Visual Studio 2010 que contiene:

- Clases de la API para simulación de tráfico
- Front End para la implementación de la simulación
- Archivos XML de ejemplo

Se encuentra disponible en la siguiente dirección url:

http://lasoto.wordpress.com/academia/maestria/

ANEXO II. Archivo XML de ejemplo.

Este archivo XML sirve únicamente para fines ilustrativos. Para obtener un archivo XML funcional que pueda ser utilizado en la simulación, puede referirse al anexo I.

Archivo XML "Dummy"

```
<?xml version="1.0" encoding="utf-8"?>
<TrafficSimulation>
  <ProgramSettings>
    <simDuration>300</simDuration>
    <simRandomSeed>42</simRandomSeed>
    <zoomLevel>5</zoomLevel>
    <velocityMappingColorMap>
      <colors>
        <SerializableColor>
          <rgbColor>-7667712</rgbColor>
        </SerializableColor>
        <SerializableColor>
          <rgbColor>-256</rgbColor>
        </SerializableColor>
        <SerializableColor>
          <rgbColor>-16751616</rgbColor>
        </SerializableColor>
      </colors>
    </velocityMappingColorMap>
  </ProgramSettings>
  <Layout>
    <title>Intersección</title>
    <infoText>Incorporación a una avenida principal</infoText>
    <LineNode>
      <nodeId>1</nodeId>
      <stopSign>false</stopSign>
      <position>
        <X>1613</X>
        <Y>1808</Y>
      </position>
    </LineNode>
    <NetworkLayer>
      <nodeIds>
        <int>1</int>
      </nodeIds>
      <title>Layer 1</title>
      <visible>true</visible>
    </NetworkLayer>
    <NodeConnection>
      <startNodeId>1</startNodeId>
      <endNodeId>2</endNodeId>
      <priority>5</priority>
      <carsAllowed>true</carsAllowed>
      <busAllowed>false</busAllowed>
      <enableOutgoingLineChange>true</enableOutgoingLineChange>
      <enableIncomingLineChange>true</enableIncomingLineChange>
      <targetVelocity>14</targetVelocity>
    </NodeConnection>
  </Layout>
  <TrafficLights cycleTime="50">
    <TimelineGroup>
```

```
      <title>Cruce1</title>
      <collapsed>false</collapsed>
      <entries>
        <TimelineEntry>
          <events>
            <TimelineEvent>
              <eventTime>0</eventTime>
              <eventLength>6</eventLength>
              <rgbColor>-16744448</rgbColor>
            </TimelineEvent>
            <TimelineEvent>
              <eventTime>37</eventTime>
              <eventLength>13</eventLength>
              <rgbColor>-16744448</rgbColor>
            </TimelineEvent>
          </events>
          <name>Incorporacion</name>
          <maxTime>50</maxTime>
          <rgbColor>-65536</rgbColor>
          <Idcode>-1</Idcode>
          <assignedNodesIds>
            <int>10</int>
            <int>12</int>
          </assignedNodesIds>
        </TimelineEntry>
      </entries>
    </TimelineGroup>
  </TrafficLights>
  <TrafficVolumes>
    <StartPoints>
      <BunchOfNodes>
        <Idcode>1</Idcode>
        <nodeIds>
          <int>1</int>
          <int>2</int>
        </nodeIds>
        <title>Start1</title>
      </BunchOfNodes>
    </StartPoints>
    <DestinationPoints>
      <BunchOfNodes>
        <Idcode>2</Idcode>
        <nodeIdes>
          <int>2</int>
          <int>1</int>
        </nodeIdes>
        <title>Destination2</title>
      </BunchOfNodes>
          </DestinationPoints>
          <TrafficVolume>
            <startId>1</startId>
            <destinationId>2</destinationId>
            <trafficVolumeCars>828</trafficVolumeCars>
            <trafficVolumeTrucks>72</trafficVolumeTrucks>
            <trafficVolumeBusses>0</trafficVolumeBusses>
          </TrafficVolume>
        </TrafficVolumes>
      </TrafficSimulation>
```