UNIVERSIDAD AUTÓNOMA DE SAN LUIS POTOSÍ

FACULTAD DE INGENIERÍA
Centro de Investigación y Estudios de Posgrado
Posgrado en Computación

Titulo:

APRENDIZAJE DE LA INGENIERÍA DE SOFTWARE UTILIZANDO SIMULADORES BASADOS EN SISTEMAS MULTIAGENTE

Tesis que presenta:

ISC. Juan Manuel Barrientos Acosta

Para obtener el grado de:

Maestro en Ingeniería de la Computación

Director de Tesis:

Dr. Héctor Gerardo Pérez González

San Luis Potosí, S.L.P. Octubre de 2010

CENTRO DE INVESTIGACIÓN Y ESTUDIOS DE POSGRADO

Para Luna Julieta y Brenda Adriel con todo mi amor, nunca dejen de brillar, aprender y crecer.

Para Elba con todo mi amor, gracias por estar conmigo.

Gracias a mi asesor de tesis, Dr. Héctor Gerardo Pérez González, por su creatividad y su paciencia.

Gracias a todos los profesores y compañeros de la maestría, aprendí mucho de Ustedes, en especial a la maestra Sandra, por eso de los agentes.

Gracias a mis Padres por hacerme como soy.

Gracias a Doña Julia y Don Armando, por que cuidan mis tesoros.

CONTENIDO

Indice de figuras iii

Indice de tablas vi

Introducción 1

I. Ingeniería de software. 3

I.I. Definición de ingeniería de software. 3

I.II. Software y productos de software. 3

I.III. Procesos, personas, tecnologías. 4

I.III.I. Procesos. 5

I.III.II. Personas. 9

I.III.II.I. Ingeniero de software. 11

I.III.II.II. Cuerpo de conocimientos de la ingeniería de software. 11

I.III.II.II.I. Áreas del conocimiento de la ingeniería de software. 11

I.III.II.III. Roles de las personas en la ingeniería de software. 15

I.III.III. Tecnologías. 15

I.IV. Los modelos de procesos de software en la actualidad. 18

II. Enseñanza de la ingeniería de software. 20

II.I. Aproximaciones para mejorar la enseñanza en ingeniería de software. 20

II.II. La simulación en la enseñanza de la ingeniería de software. 22

II.II.I. La simulación d6tions (Decisiones). 24

II.III. Tesis propuesta. 26

III. Sistemas multiagente. 28

III.I. Agente. 28

III.II. Agentes computacionales. 29

III.II.I. Propiedades de un agente desde el punto de vista de la Inteligencia Artificial 29

III.II.II. Ambiente. 30

III.III. Sistemas multiagente. 31

III.III.I. FIPA (Foundatios for Intelligent Physical Agents) 32

III.III.I.I. Manejo de agentes (FIPA). 33

III.III.I.II. Comunicación entre agentes (FIPA). 34

III.IV. Tecnología utilizada 39

III.IV.I. Agentes BDI jadex 39

III.IV.I.I. Arquitectura BDI. 39

III.IV.I.II. Declaración de agentes JADEX 42

III.IV.I.II.I. Imports. 42

III.IV.I.II.II. Capabilities (capacidades). 42

III.IV.I.II.III. Beliefs (creencias). 43

III.IV.I.II.IV. Goals (objetivos). 44

III.IV.I.II.V. Plans (Planes). 47

III.IV.I.II.VI. Características generales de Jadex 49

IV. Simulador de ambientes de desarrollo de software. 52

IV.I. Simulador D6tionsx. 52

IV.I.I. Agentes implementados. 54

IV.I.I.I. Agente Ambiente. 54

IV.I.I.II. Agente Cliente. 54

IV.I.I.III. Agente Empresario. 55

IV.I.I.IV. Agente Desarrollador. 55

IV.I.I.V. Clases auxiliares. 57

IV.I.I.VI. Agentes Empresario Jugador y Desarrollador Jugador. 57

IV.II. Plataforma de ejecución de agentes. 58

IV.III. Ejecución de la simulación. 59

IV.III.I. Inicio de agentes. 59

IV.III.II. Simulación de la asignación de proyectos. 62

IV.III.III. Simulación de la contratación de desarrolladores. 64

IV.III.IV. Simulación de la producción de software. 65

IV.III.V. Monitoreo del estado interno de los agentes. 66

IV.III.VI. Monitoreo de eventos en el sistema. 68

IV.IV. Pruebas. 70

V. Evaluacion. 73

VI. Resultados, conclusiones y trabajo futuro. 75

Bibliografía 77

INDICE DE FIGURAS

Figura I-I El Airbus A380 utiliza software crucial para sus mecanismos de control monitoreo. 4

Figura I-II Triángulo de Hierro en ingeniería de software. 5

Figura I-III Variables del Triángulo de Hierro en ingeniería de software. 5

Figura I-IV Desarrollo en cascada (Waterfall). 7

Figura I-V Desarrollo en paralelo. 7

Figura I-VI Desarrollo por fases. 8

Figura I-VII Desarrollo de prototipos. 9

Figura I-VIII Desarrollo ágil. 9

Figura I-IX Áreas del conocimiento de la ingeniería de software (a). 10

Figura I-X Áreas del conocimiento de la ingeniería de software (b). 10

Figura I-XI Áreas de conocimiento de disciplinas relacionadas a la ingeniería de software. 10

Figura I-XII Visual Studio 2008 IDE para trabajar con las tecnologías .Net (Microsoft, 2010) 16

Figura I-XIII NetBeans, IDE para diversas tecnologías, integra herramientas para el modelado, para productividad y para trabajo colaborativo (Oracle Corporation, 2010)...17

Figura I-XIV PSP Flujo del proceso tomado de (Self-Study PSP Material, 2010) 18

Figura I-XV Herramienta web para el seguimiento de RUP permite personalizar el proceso siguiendo las Fases y flujos de trabajo, tomado de (IBM, 2010)..19

Figura I-XVI Gráfico de Flujo de Extreme Programming tomado de (Wells, Extreme Programming Project, 2000). 19

Figura II-I Materia en ingeniería de software. 22

Figura II-II Ejemplos de tarjetas de decisión D6tions. 26

Figura II-III Ejemplos de tarjetas de paridad soft/punto D6tions. 26

Figura II-IV Clases que intervienen en la producción de software. 27

Figura III-I Anatomía básica de un agente. 28

Figura III-II Modelo de referencia FIPA para el manejo de agentes. 33

Figura III-III Ejemplo de un mensaje FIPA-ACL con el acto comunicativo “request”. 35

Figura III-IV FIPA - Request interaction protocol 36

Figura III-V FIPA - Cancel Meta Protocol 37

Figura III-VI FIPA Contract Net Interaction Protocol 38

Figura III-VII Arquitectura BDI. 39

Figura III-VIII Arquitectura abstracta de un agente JADEX....40

Figura III-IX Ejemplo del ADF del agente Ambiente (Ambiente.agent.xml) 41

Figura III-X Imports. 42

Figura III-XI Implementación de capability (Df capability de JADEX). 42

Figura III-XII Capabilities....43

Figura III-XIII Beliefs. 44

Figura III-XIV Ciclo de vida del objetivo. 45

Figura III-XV Goals. 47

Figura III-XVI Plans (Plans Heads). 48

Figura III-XVII Implementación de planes (Plan Bodies). 49

Figura III-XVIII Expresiones. 50

Figura III-XIX Configuraciones. 50

Figura IV-I Agentes implementados en el simulador d6tionsx 52

Figura IV-II Flujo general de actividades en la simulación d6tionsx. 53

Figura IV-III Interfaz gráfica del agente ambiente. 54

Figura IV-IV Interfaz gráfica del agente cliente 54

Figura IV-V Interfaz gráfica del agente empresario. 55

Figura IV-VI Interfaz gráfica de una empresa. 55

Figura IV-VII Interfaz gráfica del agente desarrollador. 56

Figura IV-VIII Clases Auxiliares para manejo de información en los agentes d6tionsx (a). 56

Figura IV-IX Clases Auxiliares para manejo de información en los agentes d6tionsx (b). 57

Figura IV-X Evaluación para los participantes. 58

Figura IV-XI JADEX Control Center. 59

Figura IV-XII DF Browser. 60

Figura IV-XIII Agentes desarrollador y empresario buscando el agente ambiente. 60

Figura IV-XIV Agentes d6tionsx iniciados. 61

Figura IV-XV Uso del "Contract Net Protocol" para simular la contratación entre cliente y empresario. 62

Figura IV-XVI Resultado de la negociación de proyectos en la simulación. 63

Figura IV-XVII Uso del "Contract Net Protocol" para simular la contrataciones de desarrolladores por empresarios. 63

Figura IV-XVIII Resultado de la negociación para contratar desarrolladores. 64

Figura IV-XIX Simulación de avance en el desarrollo. 65

Figura IV-XX Simulación de trabajo de desarrollo para participantes jugadores. 66

Figura IV-XXI Monitoreo de Creencias con la herramienta Introspector del JCC. 67

Figura IV-XXII Monitoreo de Objetivos con la herramienta Introspector del JCC. 67

Figura IV-XXIII Monitoreo de Planes con la herramienta Introspector del JCC. 67

Figura IV-XXIV BDI Tracer. 68

Figura IV-XXV Rastreo de eventos en la "Trace Table". 69

Figura IV-XXVI Rastreo gráfico de eventos de un agente. 69

Figura IV-XXVII Rastreo de mensajes entre agentes. 70

Figura IV-XXVIII Pruebas de ejecución en Windows 7 Profesional. 71

Figura IV-XXIX Prueba de Ejecución en Ubuntu 9.10 (a). 72

Figura IV-XXX Prueba de Ejecución en Ubuntu 9.10 (b). 72

INDICE DE TABLAS

Tabla III-I Propiedades de un agente. 29

Tabla III-II estándares emitidos por la FIPA. 32

Tabla III-III Parámetros de un mensaje FIPA-ACL. 34

Tabla III-IV Actos comunicativos FIPA 35

Tabla III-V Atributos de la etiqueta <goal>. 46

Tabla IV-I Hardware de prueba. 70

Tabla IV-II Configuraciones de prueba. 70

Tabla V-I Otras simulaciones en la enseñanza de la ingeniería de software. 73

INTRODUCCIÓN

Para dar soporte a la enseñanza de la ingeniería de software se han propuesto desde mecanismos meramente ilustrativos dentro de las aulas, como juegos y dinámicas entre los alumnos, hasta intentos de incluir la vinculación temprana de los estudiantes con empresas e instituciones, con el fin de acercarlos a una problemática real de lo que es el desarrollo de software.

Los simuladores han ganado terreno como herramienta auxiliar para presentar y enseñar a los alumnos temas que resultan demasiado abstractos cuando únicamente se presentan en formas tradicionales como las lecturas o exposiciones en clase, ya que el enseñar ingeniería de software implica dejar el entendimiento y conocimiento de cómo procesos, personas y tecnologías se interrelacionan formando el ambiente donde se produce software.

Esta tesis presenta la investigación y el trabajo desarrollado para incluir una simulación basada en sistemas multiagente en el proceso de enseñanza aprendizaje de la ingeniería de software. Para esto se desarrollo el simulador d6tionsx; actualmente es un prototipo de una simulación donde agentes de software representan a los actores en un ambiente de desarrollo de software. En este simulador agentes en el rol de cliente, empresarios y desarrolladores, interactúan para mostrar un escenario básico de producción de software donde un cliente solicita un proyecto de software a los empresarios quienes contratan desarrolladores para producir el software solicitado.

Si bien existen aproximaciones previas al uso de simuladores en la enseñanza de la ingeniería de software, que se analizan en este trabajo, d6tionsx propone el uso de tecnología multiagente, la posibilidad de que uno o más usuarios participen en la simulación, la posibilidad de que se participe como actor en diferentes roles y la evaluación de conocimientos de ingeniería de software de los participantes en una dinámica de preguntas y respuestas, como adición a propuestas anteriores.

Este documento se estructura de la siguiente manera:

El tema I, presenta las definiciones y conceptos básicos que se deben de conocer acerca de ingeniería de software, así como una breve descripción de lo que los procesos, las personas y las tecnologías representan en un ambiente de desarrollo de software, que es lo que se pretende simular.

El tema II, presenta las aproximaciones que se han tomado para mejorar la enseñanza de la ingeniería de software, aquí se analizan propuestas para el uso de la simulación como herramienta auxiliar que sirven como referencia para la simulación propuesta en esta tesis.

El tema III, presenta definiciones y estándares sobre tecnología multiagente, así como la plataforma multiagente BDI JADEX, que es la tecnología con la que está implementado el simulador d6tionsx

El tema IV, presenta la implementación del simulador desarrollado para el apoyo a esta tesis, su diseño conceptual general e información sobre su operación.

El tema V, presenta la evaluación del simulador d6tionsx basada en un comparativo con otras propuestas presentadas en la enseñanza de la ingeniería de software.

El tema V, presenta los resultados generales, las conclusiones y lineamientos para trabajo futuro, respecto al simulador d6tionsx.

Junto a este documento se incluye un CD que contiene el código fuente y la documentación técnica del prototipo desarrollado, también herramientas auxiliares necesarias para su ejecución o para continuar con su desarrollo.

I. INGENIERÍA DE SOFTWARE.

Este tema presenta los conceptos generales que se deben conocer para poder entender el simulador desarrollado. Aquí se presentan las definiciones de ingeniería de software, software e ingeniero de software. Finalmente se presenta un resumen de cómo los procesos, las personas y las tecnologías se interrelacionan para formar un ambiente en el que se produce el software.

I.I. DEFINICIÓN DE INGENIERÍA DE SOFTWARE.

Ingeniería. (Real Academia Española, 2010)
1. f. Estudio y aplicación, por especialistas, de las diversas ramas de la tecnología.
2. f. Actividad profesional del ingeniero.

En la actualidad el término ingeniería de software es ampliamente usado por profesionales y académicos, sin embargo aún existen algunas diferencias de opinión acerca de lo que este significa. Las siguientes definiciones proveen diferentes puntos de vista, a través del tiempo:

- El establecimiento y uso de principios de ingeniería (métodos) que nos permitan obtener software económicamente viable, que trabaje en máquinas reales (Bauer, 1972).

- Ingeniería de software es la forma (rama) de la ingeniería que aplica los principios de las ciencias de la computación y las matemáticas para conseguir soluciones a problemas de software con un costo efectivo (Software Engineering Institute, 1990).

- (1) La aplicación de aproximaciones sistemáticas, disciplinadas y cuantificables al desarrollo, operación, y mantenimiento del Software; esto es, la aplicación de la ingeniería al software. (2) El estudio de estas aproximaciones (IEEE, 1990).

- Ingeniería de Software, es una disciplina de ingeniería que se ocupa de todos los aspectos de la producción de software (Sommerville, 2006).

Independientemente de su origen, todas las definiciones presentan un punto en común, proponen o implican fuertemente que la ingeniería de software es más que la simple codificación de programas; esta implica el conocimiento y aplicación de diversos principios y disciplinas, para producir software de calidad dentro de tiempos y costos establecidos.

I.II. SOFTWARE Y PRODUCTOS DE SOFTWARE.

Software (Real Academia Española, 2010).
(Voz inglesa).
m. Inform. Conjunto de programas, instrucciones y reglas informáticas para ejecutar ciertas tareas en una computadora.

Existe mayor consenso en cuanto a la definición actual de software:

- Programas de computadora y su documentación asociada tales como requerimientos, modelos de diseño y manuales de usuario (Sommerville, 2006).

- (1) El conjunto completo de programas de computadora, procedimientos, y posiblemente documentación asociada e información designada para entregar a un usuario. (2) Cualquier elemento individual que pertenezca a la definición (1) (IEEE, 1990).

Figura I-I El Airbus A380 utiliza software crucial para sus mecanismos de control monitoreo.

El software es construido utilizando ingeniería de software, para resolver un problema de un cliente o un usuario o una organización, y es usado por esta persona u organización para operar alguna parte de su negocio, entendiendo "negocio" en el más amplio sentido de la palabra: finanzas, inventarios, monitoreo de pacientes, control de tráfico aéreo (Figura I-I), etcétera. En la actualidad importantes actividades dependen del correcto funcionamiento de los sistemas de Software que las soportan y un error en estos puede tener impactos graves en términos de pérdidas de negocio, pérdidas financieras, o pérdidas de prestigio e incluso pérdida de vidas (Finkelstein & Dowell, 1996), (Gleick, 1996), (Leveson, 1995).

Por esto el software debe tener propiedades básicas de calidad, el estándar ISO/IEC 9126-1 (ISO/IEC, 2001) enuncia seis atributos básicos para medir la calidad del software, y se definen de la siguiente manera:

- Funcionalidad. La capacidad para proporcionar funciones que satisfagan necesidades expresadas o implícitas, cuando se utiliza el software.
- Fiabilidad. La capacidad de mantener un determinado nivel de rendimiento.
- Usabilidad. La capacidad de ser entendido, aprendido y usado.
- Eficiencia. La capacidad para proporcionar un rendimiento apropiado en relación a la cantidad de recursos que utiliza.
- Capacidad de Mantenimiento. La capacidad de ser modificado a efectos de realizar correcciones, mejoras o adaptaciones.
- Portabilidad. La capacidad para ser adaptado para diferentes entornos, sin aplicar acciones o medios distintos de los proveídos para tal efecto en el producto.

I.III. PROCESOS, PERSONAS, TECNOLOGÍAS.

El tema fundamental de la ingeniería de software es: sistemáticamente desarrollar software que satisfaga las necesidades de algún usuario o cliente. Para direccionar este tema se han propuesto muchas variantes acerca de cómo proceder en el desarrollo de software, también se reconocen distintos roles que las personas tienen que asumir y existen muchas tecnologías en el mercado que, usadas por estas personas, soportan o auxilian los procesos para producir software.

Actualmente, el sistema de producción de cualquier proyecto o producto de software se compone de tres elementos clave: procesos, personas y tecnologías, si estos tres elementos trabajan bien en su conjunto nos

permiten alcanzar mayor productividad y calidad (Figura I-II). Esta relación es llamada por algunos autores "Triángulo de Hierro" [1] (Jalote, 2005).

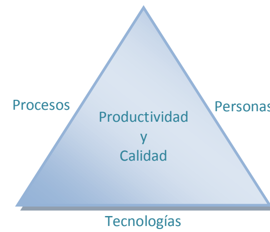


Figura I-II Triángulo de Hierro en ingeniería de software.

El triangulo de hierro presenta una vista sencilla de lo que es la producción de software, sin embargo existen diversas variantes para cada uno de los elementos mencionados, como lo muestra la Figura I-III.

Figura I-III Variables del Triángulo de Hierro en ingeniería de software.

El análisis detallado de cada una de estas variables sale del alcance de esta tesis, es por eso que en las siguientes tres secciones solamente se exponen a manera de resumen por ser conceptos que se pretenden modelar en el simulador propuesto.

I.III.I. PROCESOS.

Proceso (Real Academia Española, 2010).
(Del lat. processus).
1. m. Acción de ir hacia adelante.
2. m. Transcurso del tiempo.
3. m. Conjunto de las fases sucesivas de un fenómeno natural o de una operación artificial.

El uso del término "aproximaciones sistemáticas" en la definición de ingeniería de software, implica el uso de metodologías o procesos definidos para desarrollar software. Generalmente un método o proceso consiste en la sucesión de varias fases o etapas que agrupan un conjunto de actividades, cada fase termina con un conjunto definido de resultados. Las fases se ejecutan en un orden que es especificado por el modelo que describe el proceso que se está siguiendo.

[1] Esto puede estar relacionado al concepto de "The Iron Triangle" que se maneja en el contexto de Project Management.

La IEEE define proceso de desarrollo de software como:

- El proceso por el cual las necesidades de un usuario son traducidas en productos de software. Este proceso involucra transformar las necesidades del usuario en requerimientos de software, transformar los requerimientos en diseño, implementar el diseño en código, probar el código y en ocasiones instalar y verificar el software para su uso. Estas actividades pueden traslaparse o ser desempeñadas de forma iterativa (IEEE, 1990).

Y define el ciclo de desarrollo de software como:

- El periodo de tiempo que comienza con la decisión de desarrollar un producto de software y termina cuando el software es entregado. Este ciclo típicamente incluye una fase de requerimientos, una fase de diseño, una fase de implementación, una fase de pruebas, y algunas veces, una fase de instalación y verificación (IEEE, 1990).

En estas definiciones se pueden ver las fases típicas en prácticamente cualquier proceso y ciclo de desarrollo:

- Fase de requerimientos. Se ejecuta con el fin de entender el problema que el software resuelve. El énfasis en esta fase, se pone en identificar qué es lo que se necesita del sistema, análisis de requerimientos, y no en como el sistema va a alcanzar sus objetivos. El objetivo de las actividades en esta fase es recolectar y entender todo lo que se requiere del software y ponerlo en un documento de especificación de requerimientos.

- Fase de diseño. Se ejecuta con el fin de planificar una solución al problema especificado en el documento de requerimientos. Esta fase es el primer paso para ir del dominio del problema (necesidad o problema de un usuario o cliente), al dominio de la solución (software que satisface la necesidad). Como resultado de las actividades de esta fase, se obtienen diseños que describen la solución a construir, desde distintos puntos de vista: diseño arquitectónico, diseño de alto nivel, diseño detallado entre otros posibles.

- Fase de codificación. El objetivo de ejecutar esta fase, es traducir el diseño del sistema en código.

- Fase de Pruebas. El objetivo de ejecutar esta fase, es detectar defectos en el software: errores en las fases de requerimientos, diseño y programación. Esta fase es el principal punto de control de calidad.

Si bien la ingeniería de software se centra en el proceso de desarrollo de software, existen otros procesos que ocurren a la par de este. Por ejemplo el proceso de administración de proyectos (Project Management Process), consiste de tres fases principales: planeación, monitoreo y control, y terminación del proyecto; El proceso de inspección (Inspection Process) es usado para encontrar defectos en los productos; El proceso de administración de configuraciones de software (Configuration Management Process), se encarga de administrar los cambios de configuración que ocurren durante el desarrollo del proyecto; El proceso de administración de cambio de requerimientos (Requirements Change Management Process) se centra en manejar los cambios de requerimientos; El proceso de administración de procesos (Process Management Process), frecuentemente ejecutado por un grupo de ingeniería de procesos, se centra en mejorar el en proceso de desarrollo en sí, de tal manera que la calidad de productos desarrollados posteriormente sea también mejorada.

Debido a la importancia del proceso de desarrollo se han propuesto diversos modelos a seguir. Un modelo de proceso de software, es la representación abstracta de un proceso de software visto desde una perspectiva particular. Generalmente estos modelos también muestran la evolución del proceso en función del tiempo, y en casos de modelos muy desarrollados, también describen la integración con otros procesos.

Existen modelos de proceso de software desde lo más general, algunas veces llamados paradigmas de proceso, hasta modelos muy específicos para cierto tipo o tamaño de producto, equipo de desarrollo o industria; cada modelo funciona en mayor o menor medida dependiendo del tipo de producto que se quiera desarrollar, las personas que estén involucradas, y las restricciones establecidas para el desarrollo.

Estos modelos son aproximaciones formales para abordar un ciclo de desarrollo de software. La mayoría de los modelos utilizados actualmente corresponden a alguna de las siguientes categorías o combinación de ellas:

Diseño estructurado:

Estas metodologías fueron las que prevalecieron desde finales de los setentas y prácticamente todos los ochentas, y reemplazaron las metodologías ad-hoc que se utilizaron previamente. Estas metodologías adoptaron una aproximación paso-a-paso para moverse entre las diversas fases del ciclo de desarrollo.

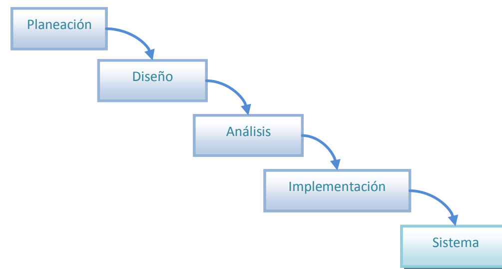


Figura I-IV Desarrollo en cascada (Waterfall).

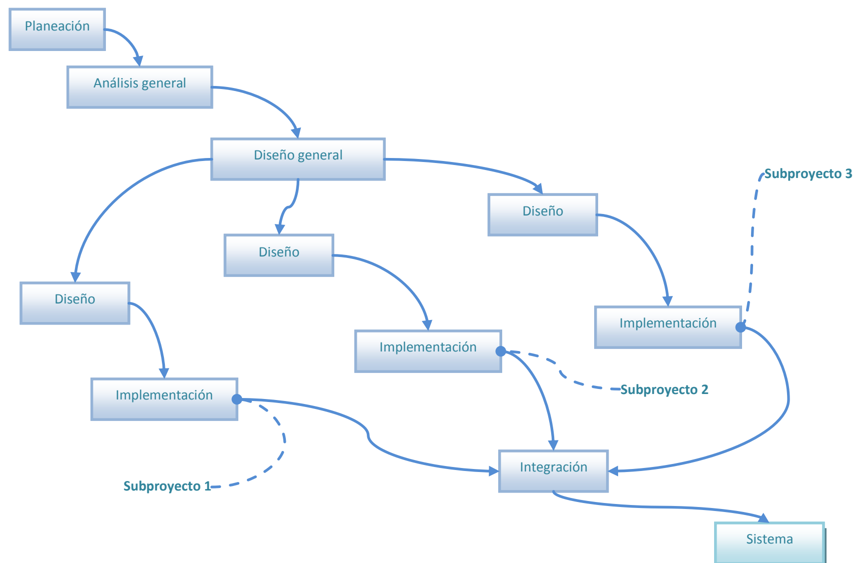


Figura I-V Desarrollo en paralelo.

- *Desarrollo en cascada (Waterfall)*.- es la metodología estructurada original, todavía se utiliza; en esta el equipo de desarrollo procede en secuencia de una fase a otra (Figura I-IV).

- *Desarrollo en paralelo*.- en esta metodología, se desarrolla un análisis y diseño general para el sistema completo, después se divide en sub proyectos que se refinan en diseño y se implementan en paralelo. Cuando todos los sub proyectos están terminados, se hace una integración de las piezas y el sistema se entrega (Figura I-V).

Desarrollo Rápido (Rapid Application Development, RAD):

Estas metodologías son relativamente nuevas, emergieron a principios de los noventas. Su enfoque consiste en desarrollar parte del sistema de forma rápida para ponerlo a disposición de los usuarios. De esta manera, los usuarios pueden entender mejor el software y sugerir revisiones o correcciones que satisfagan más adecuadamente sus necesidades.

La mayoría de las metodologías rápidas recomiendan el uso de técnicas y herramientas especiales para acelerar las fases de análisis, diseño, e implementación. Es con la combinación del ciclo de desarrollo propuesto y el uso de estas técnicas y herramientas que las metodologías rápidas logran aumentar la velocidad con que se desarrolla y la calidad de los sistemas.

- *Desarrollo por fases*.- Esta metodología aborda el desarrollo del sistema en una serie de versiones que se elaboran de forma secuencial. En la fase de análisis se identifica el concepto general del sistema, y después se categorizan los requerimientos en versiones, los requerimientos fundamentales se abordan en la primera versión; entonces se pasa al diseño e implementación, pero sólo de los requerimientos priorizados. Una vez implementada la versión 1, comienzan los trabajos para la versión 2. este proceso continua hasta que el sistema está completo o termina su vida útil. Estas metodologías tienen la ventaja de entregar rápidamente sistemas útiles a los usuarios, pero, también lidian con el problema de que probablemente las primeras versiones presenten a los usuarios sistemas “intencionalmente incompletos” (Figura I-VI).

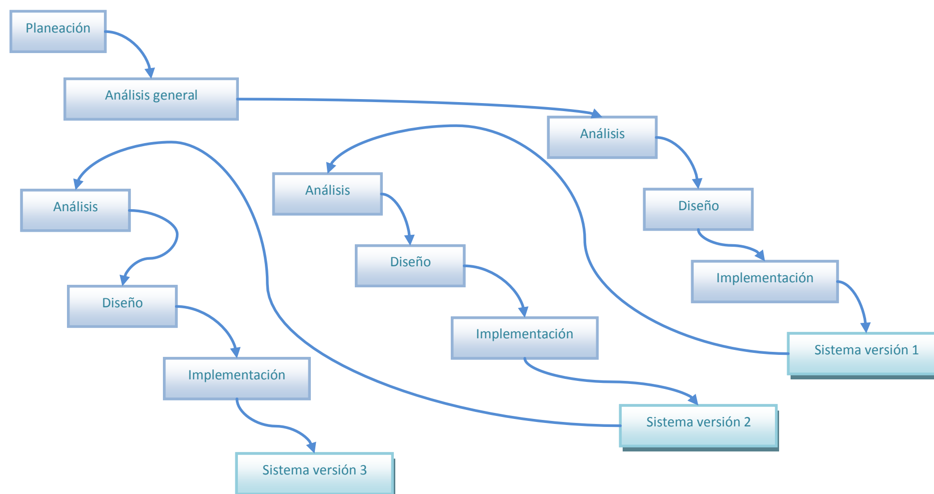


Figura I-VI Desarrollo por fases.

- *Desarrollo de prototipos*.- En esta metodología se realizan las fases de análisis, diseño, e implementación de forma concurrente, en ciclos hasta que el sistema está completo. Se desarrolla un análisis y diseño básico, y de inmediato se comienza con la construcción de un prototipo, implementado rápido e incompleto, que provee una cantidad mínima de características. Este prototipo se les muestra a los clientes o usuarios quienes proveen realimentación que es usada para reanalizar, rediseñar, y reimplementar un segundo prototipo, el cual proveerá nuevas características. Este proceso continúa hasta que el equipo de desarrollo y los usuarios finales acuerdan que el prototipo provee suficiente funcionalidad como para ser instalado y usado (Figura I-VII).

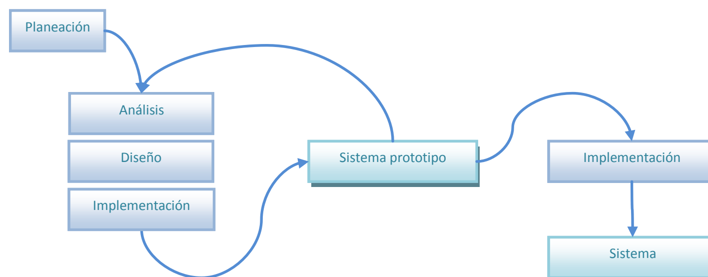


Figura I-VII Desarrollo de prototipos.

Desarrollo ágil:

Este tipo de metodologías surgió a finales de los noventas y actualmente continua en evolución, están centradas en la programación, y pretenden tener pocas reglas fáciles de seguir. La idea es conseguir un ciclo de desarrollo más aerodinámico al eliminar la sobrecarga de modelado, documentación y administración de los desarrollos. En su lugar se pone énfasis en un desarrollo de software simple e iterativo (Figura I-VIII).

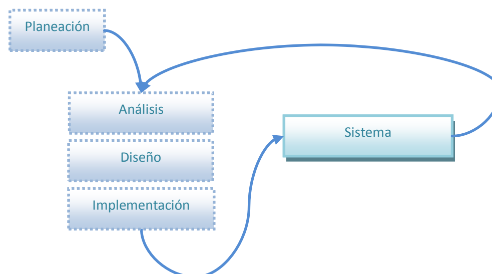


Figura I-VIII Desarrollo ágil.

I.III.II. PERSONAS.

Aunado al proceso que se sigue para el desarrollo, existe un equipo de personas que desempeñan las actividades para la producción de software. Dentro de estos equipos se encuentran individuos con diferentes habilidades y conocimientos útiles en cada fase del proceso de desarrollo utilizado, y/o en los procesos relacionados.

En la evolución del profesional de la producción de software ha sido más común el título de programador o desarrollador, la mayoría de las personas que desarrollan software poseen títulos en ciencias computacionales, ingeniería de sistemas o cualquiera de sus variantes; esto se debe a que tales carreras han estado disponibles durante los últimos treinta años aproximadamente; no así las carreras de ingeniería de software que son

relativamente nuevas. El título profesional de ingeniero de software toma formalidad en junio de 1998 cuando el "Texas Board of Professional Engineers" establece criterios para otorgar licencias de este tipo.

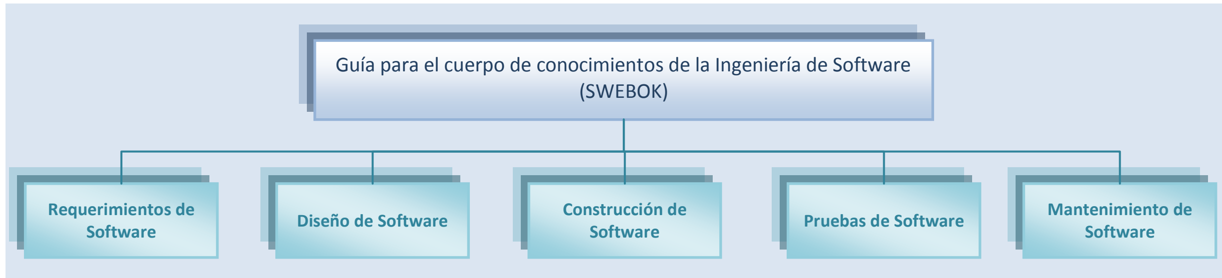


Figura I-IX Áreas del conocimiento de la ingeniería de software (a).[2]

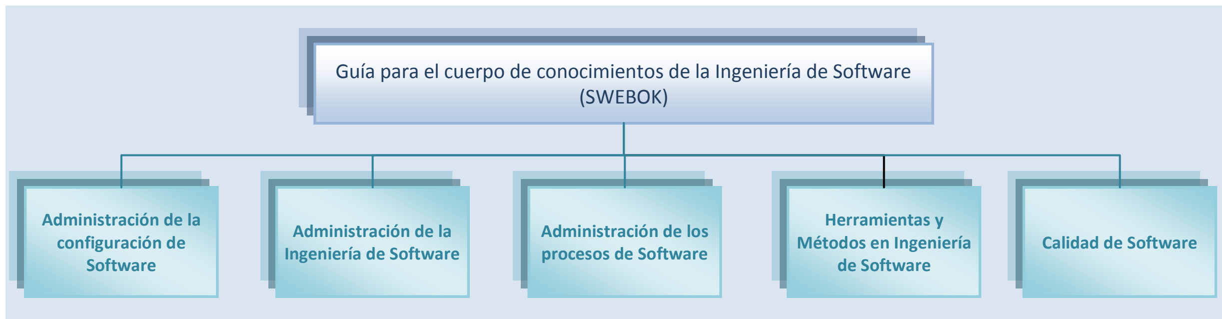


Figura I-X Áreas del conocimiento de la ingeniería de software (b).

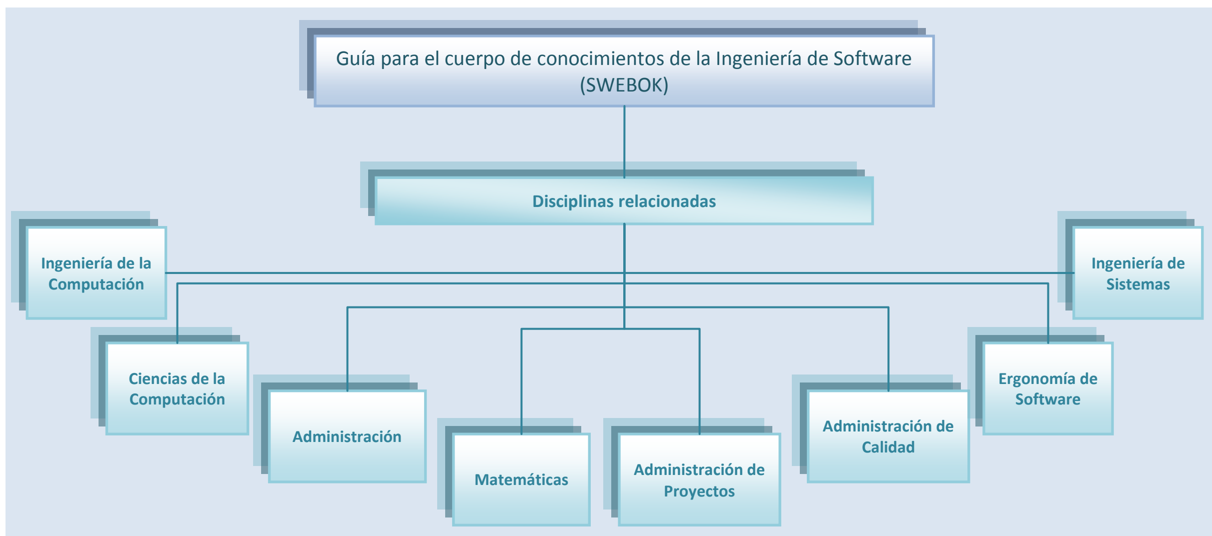


Figura I-XI Áreas de conocimiento de disciplinas relacionadas a la ingeniería de software.

[2] Las primeras áreas de conocimiento del SWEBOK parecen coincidir con el proceso en cascada, sin embargo en SWEBOK no propone ni anima ningún proceso en específico, los nombres se usan por ser de los más aceptados en el campo de la ISW.

I.III.II.I. INGENIERO DE SOFTWARE.

Ingeniero, ra. (Real Academia Española, 2010)
(De ingenio, máquina o artificio).
1. m. y f. Persona que profesa la ingeniería o alguna de sus ramas.
2. m. ant. Hombre que discurre con ingenio las trazas y modos de conseguir o ejecutar algo.

Las definiciones de ingeniero de software se derivan de las variantes de la definición de ingeniería de software:

- "El ingeniero de software, aplica los principios de las ciencias de la computación y el análisis matemático a el diseño, desarrollo, prueba, y evaluación del software y sistemas que hacen que las computadoras trabajen" (U.S. Bureau of Labor Statistics, 2010).
- "Los ingenieros de software son aquellos que contribuyen, mediante la participación directa o enseñanza, al análisis, especificación, diseño, desarrollo, certificación, mantenimiento y pruebas de sistemas de software" (Association for Computing Machinery, 2009).

I.III.II.II. CUERPO DE CONOCIMIENTOS DE LA INGENIERÍA DE SOFTWARE.

Para que la ingeniería de software pudiera ser reconocida como profesión, se requirió de un consenso acerca de los conocimientos mínimos que el profesionista que la ejerce debe poseer. De 1993 a 2000, la IEEE y la ACM cooperaron para promover la profesionalización de la ingeniería de software conformando el Software Engineering Coordinating Committee (SWECC), este inició en 1998 un proyecto para conformar el cuerpo de conocimientos de la ingeniería de software (SWEBOK por sus siglas en inglés). El propósito del SWEBOK es describir que conocimientos son generalmente aceptados como los conocimientos que un ingeniero de software requiere, y proveer acceso temático a estos (IEEE computer society, 2010).

El SWEBOK fue establecido con los siguientes objetivos:

- Promover una visión consistente de la ingeniería de software a nivel mundial.
- Clarificar la posición, y establecer los límites, de la ingeniería de software respecto a otras disciplinas, como las ciencias de la computación o las matemáticas.
- Caracterizar los contenidos de la ingeniería de software.
- Proveer acceso a través de temas específicos al conjunto de conocimientos de la ingeniería de software.
- Proveer bases para el desarrollo curricular y la creación de materiales de certificación y licenciamiento.

I.III.II.II.I. ÁREAS DEL CONOCIMIENTO DE LA INGENIERÍA DE SOFTWARE.

El SWEBOK enuncia once áreas de conocimiento cada una con diferentes sub áreas (Figura I-IX, Figura I-X, Figura I-XI).

"

- Requerimientos de software.- esta área se ocupa de los conocimientos necesarios para la elicitación, análisis, especificación, y validación de los requerimientos de software.
    - *Fundamentos sobre requerimientos de software* (definiciones, requerimientos de proceso y de producto, requerimientos funcionales y no funcionales, propiedades emergentes, requerimientos cuantificables, requerimientos de sistema, requerimientos de software).

   - *Procesos de requerimientos* (modelos de procesos de requerimientos, actores en el proceso de requerimientos, soporte y administración al proceso, calidad y mejora del proceso de requerimientos)
   - *Elicitación de requerimientos* (orígenes de requerimientos, técnicas de elicitación)
   - *Análisis de requerimientos* (clasificación, modelado conceptual, diseño y ubicación de requerimientos, negociación de requerimientos)
   - *Especificación de requerimientos* (documento de definición del sistema, especificación de requerimientos de sistema, especificación de requerimientos de software)
   - *Validación de requerimientos* (revisión, desarrollo de prototipos de validación, validación de modelos, pruebas de aceptación)
   - *Consideraciones prácticas* (naturaleza iterativa de los procesos de requerimientos, administración de cambios, atributos de los requerimientos, seguimiento, métricas)
- Diseño de software.- esta área se ocupa de los conocimientos necesarios para producir la descripción interna del software que sirve de base para su futura construcción.
   - *Fundamentos sobre diseño* (conceptos generales, contextualización del diseño de software, procesos de diseño, técnicas posibilitadoras -abstracción, unión y cohesión, descomposición modular)
   - *Temas clave en diseño de software* (concurrencia, control y manejo de errores, distribución de componentes, manejo de errores y excepciones, tolerancia a fallos, interacción, persistencia de datos)
   - *Estructura y Arquitectura de Software* (Estructuras arquitectónicas y vistas, patrones de diseño, familias de programas y frameworks)
   - *Análisis diseño y evaluación de calidad en software* (atributos de calidad, técnicas de análisis y evaluación, métricas)
   - *Notaciones para el diseño de software* (descripciones estructurales, descripciones de comportamiento)
   - *Estrategias y métodos para el diseño de software* (diseño orientado a funciones, orientado a objetos, centrado en estructura de datos, basado en componentes)
- Construcción de software.- esta área se ocupa de los conocimientos necesarios para la creación detallada de software útil trabajando.
   - *Fundamentos sobre construcción de software* (minimización de la complejidad, anticipación a cambios, construir para verificación, estándares en construcción)
   - *Administración de la construcción* (modelos de construcción, planeación de la construcción, métricas de construcción)
   - *Aspectos prácticos* (diseño de la construcción, lenguajes, codificación, pruebas, reúso, calidad, integración)
- Pruebas de software.- esta área se ocupa de los conocimientos necesarios para la verificación del comportamiento del software construido, la evaluación de su calidad, y su mejora mediante la identificación y corrección de defectos y problemas.
   - *Fundamentos de pruebas de software* (terminología relacionada, temas clave de las pruebas, relación entre pruebas y otras actividades)
   - *Niveles de pruebas* (objetivo de la prueba, pruebas de aceptación/cualificación, pruebas de instalación, pruebas alfa y beta, pruebas de usabilidad, pruebas de recuperación, desarrollo guiado por pruebas)

    - *Técnicas* (basadas en intuición y experiencia, basadas en técnicas de especificación, basadas en técnicas de codificación, basadas en defectos, basadas en uso, basadas en la naturaleza de la aplicación, combinación de técnicas)
    - *Métricas relacionadas a pruebas* (evaluación del programa bajo prueba, evaluación del desempeño de la prueba)
    - *Procesos de pruebas* (aspectos prácticos, actividades)
- Mantenimiento de software.- esta área se ocupa de los conocimientos necesarios para la modificación del software después de su despliegue, para corregir fallas, o mejorar el desempeño, o para adaptar el producto a un ambiente diferente para el que se creó.
    - *Fundamentos sobre mantenimiento de software* (terminología y definiciones, necesidad del mantenimiento, mantenimientos mayores, evolución del software)
    - *Temas clave del mantenimiento de software* (aspectos técnicos, aspectos administrativos, estimación de costos de mantenimiento, métricas de mantenimiento de software)
    - *Proceso de mantenimiento* (procesos de mantenimiento, actividades de mantenimiento)
    - *Técnicas de mantenimiento* (comprensión de programas, reingeniería, ingeniería inversa)
- Administración de la configuración de Software.- esta área se ocupa de los conocimientos necesarios para identificar la configuración de los sistemas en distintos puntos del tiempo, controlar los cambios de configuración y mantener la integridad de la configuración.
    - *Administración de el proceso de configuración* (contexto organizacional, guías y restricciones para el proceso de configuración, planeación para configuración, plan de configuración, supervisión de configuración)
    - *Identificación de la configuración del software* (identificación de elementos a controlar, bibliotecas de software)
    - *Control de la configuración del software* (solicitud, evaluación y aprobación de cambios, implementación de cambios)
    - *Auditoria del estado de la configuración* (información del estado de la configuración, reporte de estado de la configuración)
    - *Auditoría de la configuración del software* (auditoría de la configuración funcional, auditoría de la configuración física, etc.)
    - *Administración de la liberación y entrega del software* (empaquetado, administración del despliegue)
- Administración de la ingeniería de software.- esta área se ocupa de los conocimientos necesarios para la aplicación de actividades de administración (planeación, coordinación, métricas, monitoreo, control) que aseguran que el desarrollo y mantenimiento de software es sistemático, disciplinado, y cuantificable.
    - *Iniciación y definición de ámbito* (determinación y negociación de requerimientos, análisis de factibilidad, procesos para critica y revisión de requerimientos)
    - *Planeación de proyectos de software* (proceso de planeación, determinación de entregables, estimación de costos, esfuerzos y agendas, asignación de recursos, administración de riesgos, administración de la calidad, administración del plan)
    - *Establecimiento de proyectos* (implementación de planes, administración de contratos con proveedores, implementación de métricas de procesos, monitoreo de procesos, control de procesos, reporte)
    - *Reseña y evaluación* (determinar la satisfacción de requerimientos, criticar y evaluar el desempeño)
    - *Cierre de proyectos* (determinar cierre, actividades de cierre)

  - *Métricas en ingeniería de software* (compromiso con el uso de métricas, planificación de el proceso de métricas, desempeño de el proceso de métricas, evaluación de métricas)
- Administración de los procesos de software.- esta área se ocupa de los conocimientos necesarios para la definición, implementación, evaluación, administración, cambio y mejora de los ciclos de vida de procesos de software.
  - *Implementación y cambios de procesos* (infraestructura de procesos, ciclo de administración de procesos de software, modelos para implementación y cambio de procesos)
  - *Definición de procesos* (modelos de ciclo de vida del software, procesos de ciclo de vida de software, notaciones para la definición de procesos, adaptación de procesos, automatización)
  - *Valoración de procesos* (valoración de modelos, valoración de métodos)
  - *Métricas de procesos y productos* (métricas de proceso, calidad de las métricas, modelos de información de software, técnicas para medir procesos)
- Herramientas y Métodos en ingeniería de software.- esta área se ocupa de los conocimientos sobre herramientas y los métodos que intentan soportar sistematicidad en la ingeniería de software.
  - *Herramientas* (para requerimientos, diseño, construcción, pruebas, mantenimiento, SCM, procesos, etc.)
  - *Métodos* (heurísticos, formales, prototipos)
- Calidad de Software.- esta área se ocupa de los conocimientos para evaluar la calidad de los productos de software.
  - *Fundamentos de calidad de software* (ética y cultura, costos y valor de la calidad, modelos y características de calidad, mejora de la calidad)
  - *Proceso de administración de calidad en software* (Aseguramiento de la calidad, verificación y validación, auditorias)
  - *Consideraciones prácticas* (requerimientos de calidad, catalogación de defectos, administración de técnicas de calidad, métricas de calidad)
- Disciplinas relacionadas.- esta área se ocupa del conocimiento acerca de las disciplinas con las que la ingeniería de software comparte conocimientos.
  - *Ingeniería de la computación* (algoritmos y complejidad, arquitectura de computadoras, ingeniería de sistemas de cómputo, circuitos, lógica digital, procesamiento de señales digitales, sistemas distribuidos, electrónica, sistemas incrustados, interacción humano-computadora, administración de información, sistemas inteligentes, redes, sistemas operativos)
  - *Ciencias de la computación* (estructuras discretas, programación, algoritmos, complejidad, Net-Centric computing, lenguajes de programación, graficación, sistemas inteligentes, métodos numéricos)
  - *Administración* (finanzas, mercadotecnia, ventas, administración de negocios, leyes, administración de recursos humanos, economía, políticas y estrategias de negocio)
  - *Matemáticas* (algebra lineal, calculo diferencial e integral, ecuaciones diferenciales, probabilidad, estadística, análisis numérico, matemáticas discretas)
  - *Administración de Proyectos* (integración de proyectos, administración de ámbito de proyectos, administración de tiempos, administración de costos, administración de la calidad, administración de los recursos humanos, administración de la comunicación, administración de riesgos, procuración de recursos)
  - *Administración de Calidad* (Desarrollo, implementación y verificación de sistemas de calidad, planeación, control y aseguramiento de calidad en productos y procesos, mejora de la calidad)

- Ergonomía de Software (naturaleza de la interacción humano computadora, uso y contexto de las computadoras en el trabajo y las organizaciones humanas, áreas de aplicación de las TI, adaptación humano computadora, arquitectura de interfaces de las computadoras)
- *Ingeniería de Sistemas* (procesos de negocios, evaluación de operaciones, arquitectura de sistemas, arquitectura de soluciones, ciclo de vida de costos, análisis costo beneficio, logística, modelado, simulación) ”

La ingeniería de software es una disciplina emergente, esto es especialmente cierto si se compara con otras disciplinas de la ingeniera como la eléctrica o civil. Esto hace que las áreas del conocimiento, sus límites y relaciones con otras disciplinas estén en constante evolución.

I.III.II.III. ROLES DE LAS PERSONAS EN LA INGENIERÍA DE SOFTWARE.

Entre los roles más comunes que toman las personas en el desarrollo de software están: analista de negocios, analista de procesos, analista de sistemas, analista de infraestructura, analista de requerimientos, arquitecto de software, diseñador de interfaces, diseñador de bases de datos, integrador, probador o tester, diseñador de pruebas, analistas de pruebas, programador, administrador de proyectos, entre otros. La participación de estos roles dentro del proceso de producción depende también del modelo de procesos que se esté utilizando. Cada uno de estos roles es una especialización dentro de la ingeniería de software. Un profesionista puede tomar más de un rol en un proceso de producción de software; cada rol obliga a utilizar áreas específicas del cuerpo de conocimientos de la ingeniería de software.

Otra parte importante a mencionar, dentro del tema de las personas relacionadas a la producción de software, es el hecho de que en la actualidad se hace mucho énfasis en incluir permanentemente al cliente, o usuario que solicita el producto; el término utilizado para señalar a estas personas es “stakeholder”: cualquiera que tenga probado interés en el proyecto (Heldman, 2003), y se usa para indicar que como parte del equipo de desarrollo, hay que incluir tanto a quien solicita el producto o proyecto, quien lo paga o el usuario principal, como a todos aquellos involucrados con su uso: usuarios finales, beneficiarios indirectos, futuros usuarios.

I.III.III. TECNOLOGÍAS.

Tecnología. (Real Academia Española, 2010)
(Del gr. τεχνολογία, de τεχνολόγος, de τέχνη, arte, y λόγος, tratado).
1. f. Conjunto de teorías y de técnicas que permiten el aprovechamiento práctico del conocimiento científico.
2. f. Tratado de los términos técnicos.
3. f. Lenguaje propio de una ciencia o de un arte.
4. f. Conjunto de los instrumentos y procedimientos industriales de un determinado sector o producto.

En el contexto de la producción de software se debe entender por tecnología, todo instrumento que sirva para ejecutar alguna actividad, dentro del proceso de desarrollo.

La tecnología más obvia son los lenguajes de programación, estos soportan la etapa de implementación en cualquier proceso; como ejemplos de lenguajes de programación: PHP, C#, JavaScript, Perl, C, Ruby and Ruby on Rails, Java, Python o Visual Basic .Net. Los ambientes integrados de desarrollo (Integrated development environment Figura I-XII), conjuntan herramientas para la creación de software como editores, compiladores, depuradores o diseñadores de interfaces.

Otras tecnologías que han tomado fuerza en los pasados quince años, son los lenguajes de modelado, usados para expresar información, o conocimiento, o sistemas, en una estructura que es definida por un conjunto consistente de reglas, estos lenguajes se clasifican en dos categorías:

- Los lenguajes de modelado gráficos, que usan técnicas basadas en diagramas, utilizan símbolos para representar conceptos y líneas que conectan símbolos para representar relaciones, además de otras notaciones gráficas para representar restricciones. Como ejemplos: UML, SysML, diagramas de flujo, árboles de decisión, EXPRESS, IDEF, redes de Petri, etcétera.
- Lenguajes de modelado textuales, que usan palabras clave estandarizadas acompañadas de parámetros, símbolos y valores para hacer expresiones interpretables. Como ejemplos: teoría de conjuntos, lógica de predicados, funciones λ, notación Z, etcétera.

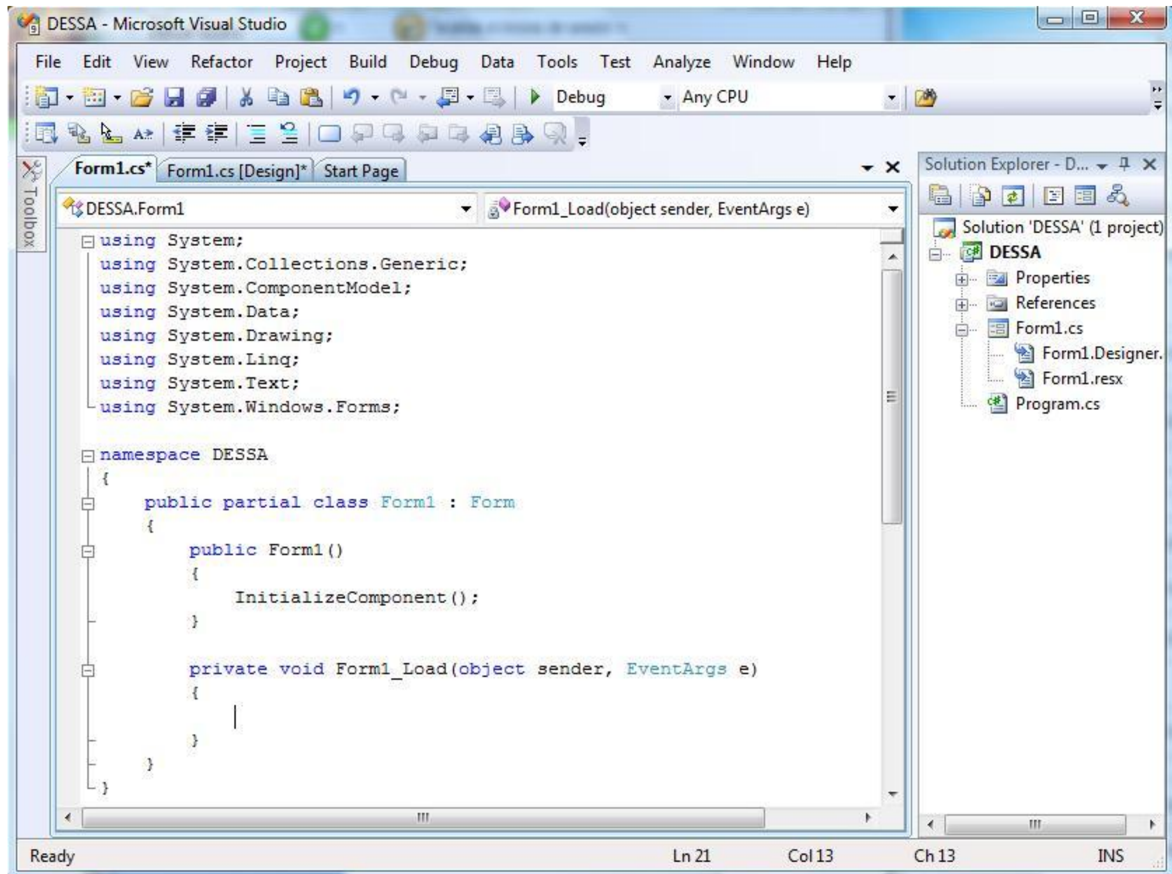


Figura I-XII Visual Studio 2008 IDE para trabajar con las tecnologías .Net (Microsoft, 2010)

Los lenguajes de modelado, soportan las actividades en la fase de análisis para documentar el dominio del problema, y en la fase de diseño para especificar o describir el dominio de la solución (diseñar el software antes de codificarlo).

A mediados de los ochentas se comenzó a utilizar el término CASE (Computer-Aided Software Engineering, ingeniería de software asistida por computadora), para nombrar las herramientas de software que apoyan la producción de software (Case, 1985).

CASE es una categoría de productos de software que pretenden automatizar procesos de producción o partes de este. Las herramientas CASE usadas en la etapa de análisis y al diseño se les conoce como de alto nivel o

"upper CASE", las herramientas CASE usadas en para la implementación y el despliegue son llamadas de bajo nivel o "lower CASE". Actualmente es muy común encontrar CASE integrales o I-CASE (Figura I-XIII) las cuales integran funcionalidad tanto de alto como de bajo nivel.

Otras herramientas que hay que considerar son las herramientas que soportan los "otros procesos" que se relacionan a la producción de software; como ejemplo herramientas para planeación de proyectos, herramientas para la administración de versiones, herramientas para documentación entre otras.

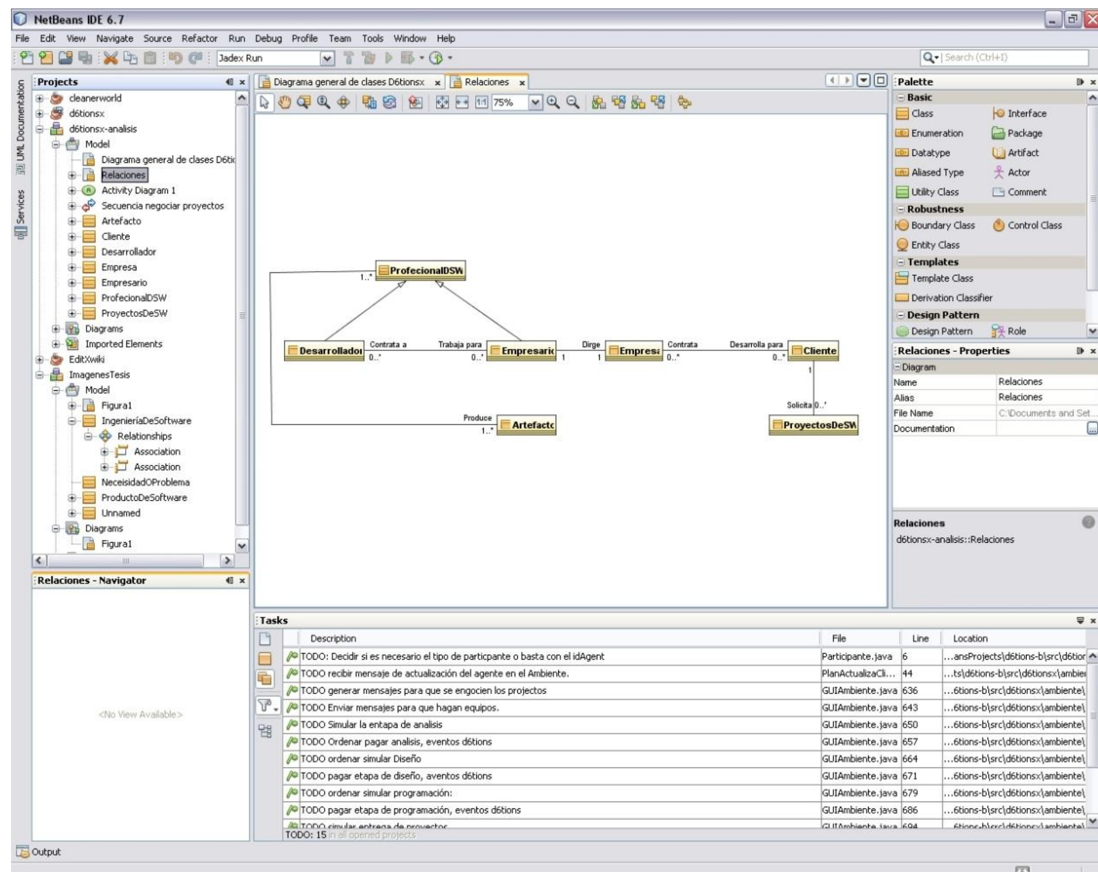


Figura I-XIII NetBeans, IDE para diversas tecnologías, integra herramientas para el modelado, para productividad y para trabajo colaborativo (Oracle Corporation, 2010)

Existen además tecnologías que marcan paradigmas desde lo más general hasta muy específico respecto a las metodologías de implementación de los sistemas de software, como ejemplo las tecnologías orientadas a objetos (programación orientada a objetos, análisis y diseño orientado a objetos), la programación orientada a aspectos, patrones de diseño, frameworks de trabajo específicos para un tipo de aplicación, lenguaje, o incluso empresa como Struts, .Net Framework, PHP Basic Framework. Algunas de estas tecnologías catalogan en la intersección de tecnologías y procesos.

I.IV. LOS MODELOS DE PROCESOS DE SOFTWARE EN LA ACTUALIDAD.

Basados en paradigmas de proceso, los roles que las personas toman, y en ocasiones las herramientas disponibles, se han desarrollado metodologías más detalladas que incorporan tanto el proceso de desarrollo de software como los procesos paralelos relacionados; estas, son creadas como estándares internacionales por agencias gubernamentales o instituciones de investigación, o por firmas de consultoría o empresas para su venta.

Por mencionar ejemplos representativos, el Software Engineering Institute promueve modelos como el Personal Software Process (PSP, Figura I-XIV), el cual provee métodos para mejorar el proceso personal para el desarrollo; otro modelo propuesto por el SEI es el Team Software Process, este en combinación con PSP se propone para ayudar a equipos de desarrollo a producir sistemas de gran escala; Capability Maturity Model Integration es otro modelo propuesto por el SEI, este está enfocado a la mejora de procesos en la producción de productos y servicios (Software Engineering Institute, 2010).

El Rational Unified Process, es un modelo para procesos de desarrollo de software iterativo, creado por Rational Software Corporation, una división de IBM desde 2003 (IBM, 2010); este no es un modelo limitativo, sino que pretende ser un proceso adaptable para las organizaciones que lo adoptan seleccionando los elementos del proceso que son más apropiados a sus necesidades (Figura I-XV).

Las metodologías ágiles (Agile Alliance, 2010) también han desarrollado sus estándares como eXtreme Programming: XP (Wells, 2009), Scrum (Advanced Development Methods Inc., 2010), y el Dynamic Systems Development Method (DSDM Consortium, 2010). Extreme Programming (Figura I-XVI) es el estandarte de estas metodologías, implementa un ambiente simple centrado en el trabajo en equipo y la auto organización, para desarrollar sistemas lo más eficientemente posible; en esta metodología es primordial la comunicación entre clientes y desarrolladores.

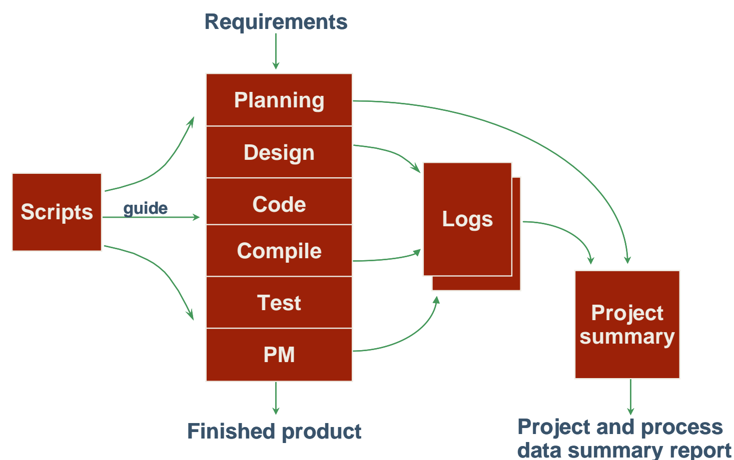


Figura I-XIV PSP Flujo del proceso tomado de (Self-Study PSP Material, 2010)

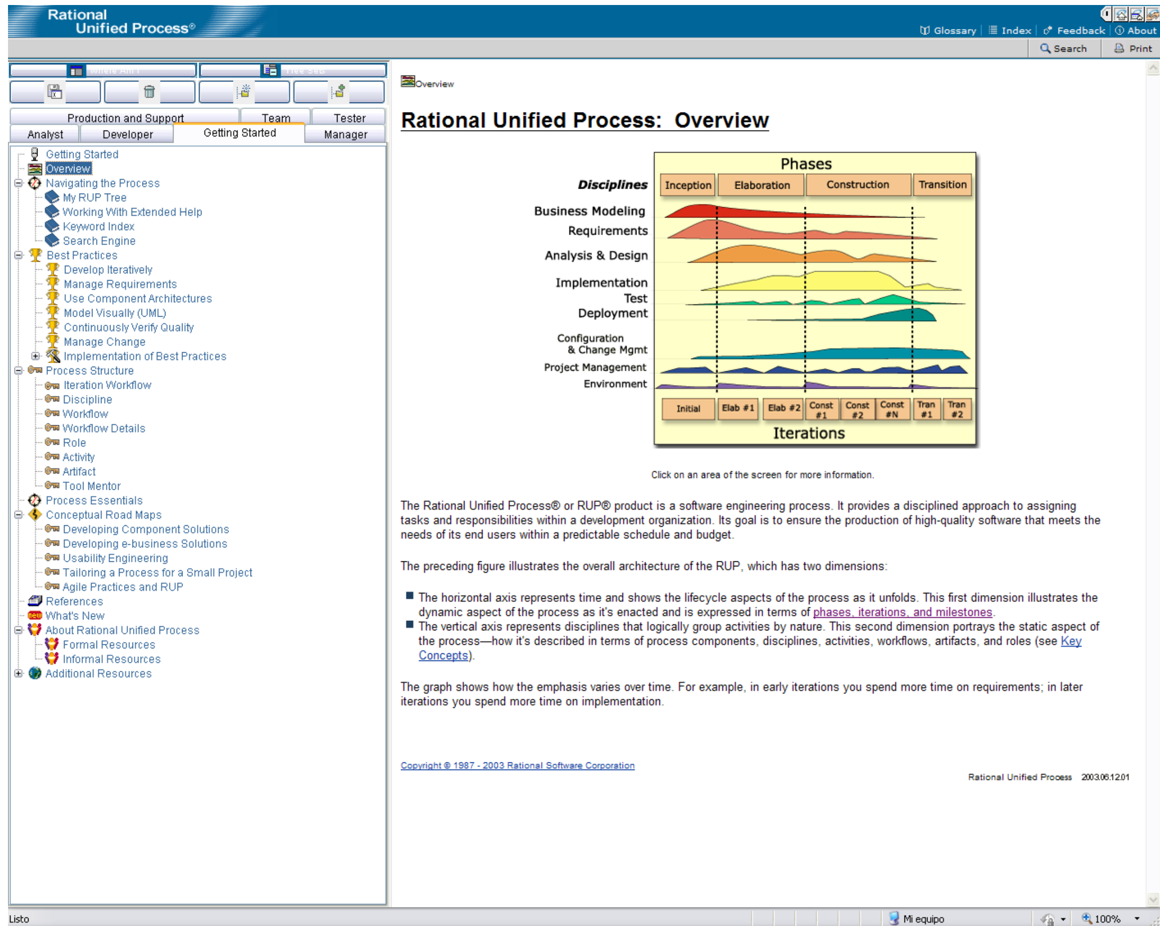


Figura I-XV Herramienta web para el seguimiento de RUP permite personalizar el proceso siguiendo las Fases y flujos de trabajo, tomado de (IBM, 2010).

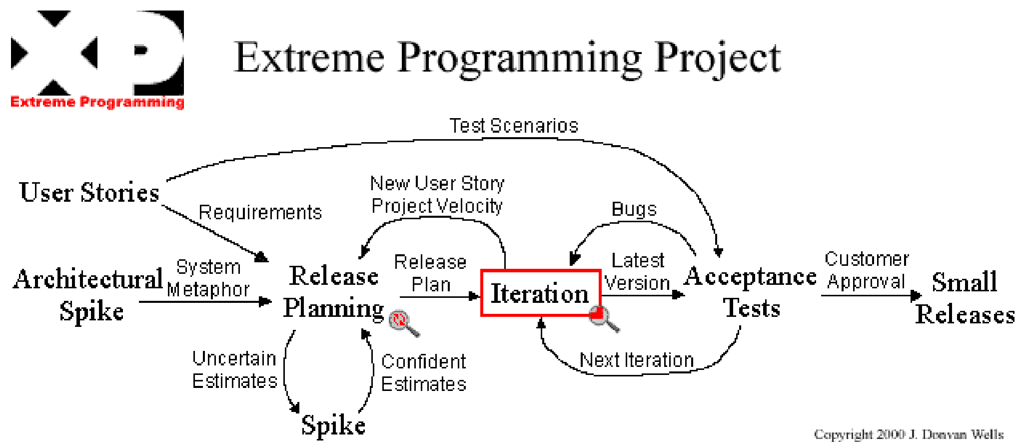


Figura I-XVI Gráfico de Flujo de Extreme Programming tomado de (Wells, Extreme Programming Project, 2000).

II. ENSEÑANZA DE LA INGENIERÍA DE SOFTWARE.

Este tema muestra las aproximaciones que se han seguido en afán de mejorar la enseñanza en ingeniería de software, dentro de estas aproximaciones se presentan propuestas previas para el uso de simulación como herramienta auxiliar.

Históricamente, la ingeniería de software se ha desarrollado como una sub disciplina de las ciencias de la computación, por lo cual sus primeras apariciones en la educación son dentro de cursos en carreras asociadas. Existe bibliografía de varios autores ampliamente usada en estos cursos a nivel internacional, entre los más notables se encuentran (Sommerville, 2006) y (Pressman, 2009), los cuales publicaron las primeras ediciones de sus libros en los ochentas, y siguen siendo usados ampliamente después de varias ediciones. A finales de los noventas aparecen las primeras carreras a nivel licenciatura y posteriormente postgrados en ingeniería de software. La ACM en conjunto con la IEEE publicó las directrices para la creación de planes de estudio a nivel licenciatura en ingeniería de software (ACM/IEEE-CS Task Force on Computing Curricula, 2004), y actualmente patrocinan la primera versión de directrices para planes de estudio a nivel maestría (GSwE2009, 2010).

La aproximación clásica para enseñar ingeniería de software consiste en la presentación de lecturas con las cuales se exponen conceptos y teoría relacionada, además tradicionalmente se usa el desarrollo de un proyecto de clase pequeño que pretende brindar experiencia base a los estudiantes.

La educación en ingeniería de software es difícil de seguir debido a la naturaleza cambiante del campo (Žagar, Bosnić, & Orlić, 2008); a los estudiantes se les requiere que tengan habilidades en arquitectura de sistemas, programación, diseño de sistemas y aplicaciones, etcétera; cada generación de estudiantes se enfrenta con cambios en lo que es considerado tópico actual en el mundo del software; y de cada generación se espera un conjunto de habilidades diferentes a las generaciones anteriores. Junto a las habilidades técnicas, también son requeridas habilidades sociales, habilidades para presentar y adaptar conocimiento, habilidades de trabajo en equipo, habilidades de comunicación, habilidad de auto aprendizaje, etcétera.

II.I. APROXIMACIONES PARA MEJORAR LA ENSEÑANZA EN INGENIERÍA DE SOFTWARE.

Para abordar las dificultades en la enseñanza de ingeniería de software, existen principios básicos que deben de ser atendidos (ACM/IEEE-CS Task Force on Computing Curricula, 2004):

- En afán de fortalecer lo aprendido, muchos conceptos, principios, y temas en ingeniería de software deben ser abordados de forma recurrente, para ayudar a que los estudiantes siempre los tengan en mente; en carreras de ingeniería de software esto es más fácil de direccionar en función de que todo el plan está enfocado a estos conceptos, en carreras donde solamente se ve la ingeniería de software como una materia de uno o dos semestres, el tiempo es una limitante para la práctica por repetición de estos.
- Los estudiantes deben aprender algo más que ingeniería de software. Como se ha dicho la práctica requiere de interrelación con equipos multidisciplinarios, la facilidad de entenderse con otros miembros del equipo dependerá de los conocimientos que se tengan en común. Raramente ocurre que un estudiante se licencie en más de una materia, sin embargo es importante desarrollar la habilidad de adquirir y adaptar nuevos conocimientos en la práctica de la ingeniería de software.
- Es importante entrenar y desarrollar ciertas habilidades personales que van más allá del tema de la ingeniería de software: desarrollar juicio crítico, desarrollar la capacidad de evaluar y cuestionar el

conocimiento recibido, la capacidad de reconocer las limitaciones propias, comunicación efectiva, comportamiento ético y profesional.

- A los estudiantes se les debe inculcar la capacidad y el deseo de aprender, estas actitudes deben conservarlas de manera permanente para ejercer su carrera.
- La ingeniería de software debe de enseñarse como una disciplina que resuelve problemas.
- Principalmente la enseñanza de la ingeniería de software debe hacerse con bases en problemas del mundo real como casos de estudio, elaboración de proyectos, ejercicios prácticos o experiencia del estudiante en trabajos del tema.
- La educación en ingeniería de software siempre debe ir más allá del formato basado en lecturas didácticas, por lo tanto es importante fomentar el uso de nuevas aproximaciones de enseñanza aprendizaje.

Para esto se ha desarrollado diversos trabajos relacionados a la aplicación de estos principios, como ejemplo:

- (Liu, Marsaglia, & Olson, 2002) presentan una aproximación para la formación en ingeniería de software a través de un curso que permite a los alumnos experimentar aspectos como: desarrollo basado en equipos, definición de problemas de clientes reales, administración del desarrollo del proceso, desarrollo iterativo, uso de herramientas y lenguajes de programación, presentaciones de expertos de la industria y presentación formal de productos.
- (Claypool & Claypool, 2005) utilizan el desarrollo de videojuegos como proyecto de clase, para incluir un “factor de diversión”, que atraiga a los estudiantes hacia los cursos de ingeniería de software.
- (Hazzan & Tomayko, 2005) proponen un curso centrado en los aspectos humanos de la ingeniería de software, específicamente aspectos cognitivos tales como la comprensión de programas, y aspectos sociales como el trabajo en equipo, con el objetivo de crear conciencia en los alumnos de la riqueza y complejidad de los dilemas, conflictos y cuestiones éticas que pueden enfrentar al desarrollar un software en la vida real.
- (Jaccheri & Morasca, 2006) señalan la necesidad de integrar a los profesionales de la industria en la formación de nuevos ingenieros de software.
- (Pieterse, Kourie, & Sonnekus, 2006) reportan una investigación para evaluar cómo se interrelacionan alumnos con diferentes personalidades, en afán de mostrar la importancia de la integración de equipos para el éxito en la ejecución de proyectos, integrando estos conceptos a la educación en ingeniería de software.
- (LeJeune, 2006) analiza la factibilidad de usar un modelo de proceso específico como extreme programing en un curso de ingeniería de software. (Boetje, 2006) propone el uso de varias metodologías, desde cascada hasta extreme programming, en un curso de dos semestres para mostrar a los alumnos los conceptos comunes como trabajo en equipo, comunicación y codificación.
- (Hood & Hood, 2006) en orden de generar experiencia, utilizan la construcción de un puente con piezas de LEGO® como la simulación de un proyecto, para enseñar a los estudiantes tópicos sobre la administración de proyectos, como la administración de cambios o la medición y reporte de avances.
- (Cockburn, 2007) presenta una propuesta para readaptar la enseñanza basada en cursos de un semestre para incluir tópicos actualizados de ingeniería de software.
- (Rusu, Rusu, Docimo, Confesor, & Paglione, 2009) aseguran que los estudiantes pueden ser preparados para unirse a la industria a través de formación que incluya cursos teóricos junto con proyectos reales en la industria y colaboración de equipos de más de una academia.

Todos estos esfuerzos se apegan a un objetivo común, mostrar la integración de los tres conceptos fundamentales en la ingeniería de software, enseñando acerca de los procesos, enseñando acerca de las personas y enseñando acerca de las tecnologías.

II.II. LA SIMULACIÓN EN LA ENSEÑANZA DE LA INGENIERÍA DE SOFTWARE.

Simulación.
(Del lat. simulatĭo, -ōnis).
1. f. Acción de simular.
Simular.
(Del lat. simulāre).
1. tr. Representar algo, fingiendo o imitando lo que no es.

La Figura II-I, describe mediante un diagrama de clases UML[3], los elementos que se interrelacionan al impartir una materia en ingeniería de software.

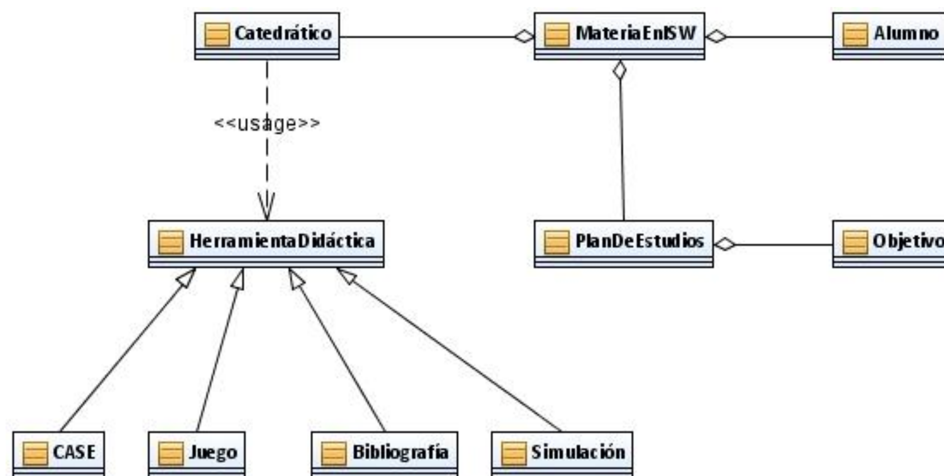


Figura II-I Materia en ingeniería de software.

En la formación académica en ingeniería de software, es necesario enfrentar al estudiante con el tipo de problemas que le permitan aplicar los conocimientos adquiridos. Es altamente deseable, que el alumno desarrolle experiencia en trabajos reales, con vinculación academia-industria, pero como esto no siempre es posible, entonces la simulación de estos trabajos reales debe de ser utilizada como una herramienta auxiliar. Las simulaciones entran dentro de la clase de herramientas didácticas, que un catedrático puede usar para reforzar el aprendizaje de los contenidos de un plan de estudios.

La ingeniería de software es principalmente un fenómeno complejo, ya que en ella intervienen factores individuales como: formación personal, carácter del individuo, conocimientos de la materia; factores de socialización como: trabajo en equipo, comunicación efectiva, organización; y macro factores como: factores

[3] UML es un lenguaje de modelado ampliamente usado, su estudio o descripción salen del alcance de esta tesis, su uso puede ser consultado en (Ambler, 2005), (Pender, 2003) u (Object Management Group, Inc., 2010), entre muchas opciones.

económicos, globalización, propiedad intelectual. Dentro del estudio de diversas ciencias (físicas, químicas, biológicas, matemáticas, etcétera) y en general en el estudio de diversos sistemas, se ha establecido que en ocasiones la única manera eficaz de observar y entender el comportamiento complejo de un sistema, es construir un modelo del fenómeno con el cual simular este comportamiento. Esto da cabida a la aplicación de la simulación en la enseñanza de la ingeniería de software.

En términos generales se puede decir que la simulación es la imitación de alguna cosa real, la acción de simular algo generalmente implica representar cierta característica o comportamiento clave de algún sistema físico o abstracto seleccionado.

Como definición de simulación diremos que: “es el proceso de diseñar un modelo de un sistema real y conducir experimentos con este modelo con el propósito de entender el comportamiento del sistema y/o evaluar varias estrategias para su operación” (Shannon, 1998). Un sistema simulado imita el comportamiento y las respuestas de un sistema real, a eventos que ocurren en el transcurso del tiempo. El término modelo y sistema son componentes claves de la definición de simulación. Por modelo nos referimos a la representación de un grupo de objetos o ideas en alguna forma distinta a de la entidad misma. Por sistema nos referimos a un grupo o colección de elementos interrelacionados los cuales cooperan para lograr algún objetivo.

La simulación es una de las herramientas más poderosas de las que se disponen hoy día, ya que hace posible el estudio, análisis y evaluación de sistemas o procesos complejos que en un ambiente real o completo resultarían difíciles de observar. La principal fortaleza de la simulación es el hecho de que se puede aplicar a sistemas ya existentes como también a sistemas que se desean crear, lo cual la convierte en una metodología práctica para la observación y solución de problemas. Además, la simulación permite a los estudiantes observar o estudiar procesos que pudieran resultar costosos o peligrosos de implementar a un nivel real; también les provee herramientas para que lleven su preparación a través de casos de estudio, lo cual ayuda a relacionar experiencias con conceptos abstractos.

La forma más básica del uso de la simulación en la enseñanza de ingeniería de software es el proyecto pequeño de clase, ya sea de forma individual o en equipos los alumnos se encargan de desarrollar un pequeño software que cumple con las especificaciones establecidas por el profesor, esta aproximación es utilizada desde clases que se enfocan en lenguajes de programación solamente, así como en clases de ingeniería de software.

Con la necesidad de mostrar a los alumnos un panorama completo del ejercicio de la ingeniería de software, se han desarrollado diversas propuestas de cómo aplicar la simulación.

En (Drappa & Ludewig, 2000) se presenta el simulador SESAM (Software Engineering Simulation by Animated Models), usando este simulador un estudiante puede tomar el rol de un administrador de proyectos de software, el proyecto de software simulado se puede concluir en periodos de tiempo de un par de horas. El estudiante controla el simulador usando una interface basada en texto, escribiendo y leyendo comandos, puede contratar o despedir empleados o pedirle que desempeñen tareas como “preparar una especificación”, “revisar un diseño” o “probar un código”, en respuesta recibe mensajes como “he terminado la especificación”, o “durante la inspección he encontrado errores”, el estudiante tiene entonces que reaccionar a estos mensajes ya que es toda la información que obtiene. El simulador mantiene un seguimiento de variables internas como el “grado con el cual la especificación refleja el requerimiento”, cuando la simulación termina, el estudiante puede analizar las variable internas y conocer su calificación.

Otra simulación es presentada en (Baker, 2004), esta no es una simulación por computadora, se trata de un juego de cartas educacional que simula un proceso de ingeniería de software, este juego es llamado “problems

and programmers" (problemas y programadores), está organizado como un juego de competencia en el cual dos jugadores toman el rol de líderes de proyecto en una misma empresa, a ambos se les asigna el mismo proyecto y se les pide completarlo rápidamente; las cartas representa proyectos de desarrollo, conceptos de ingeniería de software, personas siguiendo el proceso y problemas que ocurren al desarrollar el proyecto.

SimJAVA (Shaw & Dermoudy, 2005), es otro simulador del proceso de desarrollo de software, este se presenta con una interface gráfica basada en web, permite a los estudiantes tomar el rol de administrador de proyectos desarrollando un producto de software hipotético, el núcleo del modelo creado para este simulador es la representación de desarrolladores, para estos se modelaron personalidades y habilidades individuales, que influencian la forma en que trabajan en el desarrollo; los proyectos son modelados como una colección de actividades por realizar por los desarrolladores. El usuario de esta simulación utiliza paneles de control en la interfaz web para controlar el trabajo de los desarrolladores y observar el proceso de desarrollo del producto de software.

(Henry & LaFrance, 2006), argumentan que para que los estudiantes entiendan completamente la ingeniería de software, la enseñanza debe considerar los aspectos sociológicos y de comunicación que ocurren en esta, también llamados aspectos socio-técnicos, para esto proponen la inclusión de juegos de rol en puntos del desarrollo de un proyecto clase; por ejemplo, al trabajar con el tema de elicitación de requerimientos, se les pide a los estudiantes que asuman el rol de desarrolladores o de clientes e interactúen en estos roles, después, se les pide discutir sobre la experiencia de representar estos roles.

El simulador SimSE, es un video juego educacional que "provee la experiencia virtual de participar en procesos de ingeniería de software de gran escala, cuasi-realistas" (Navarro, 2010); es para un jugador, el cual toma el rol de administrador de proyecto dirigiendo un equipo de desarrolladores, como el jugador administra el proceso para completar un proyecto de ingeniería de software, el puede contratar y despedir empleados, asignarles tareas, monitorear progreso, entre otras cosas. La interfaz de usuario de SimSE es completamente gráfica, desplegando una oficina virtual en la cual toma lugar el proceso de ingeniería de software. En la interfaz se incluye información acerca de los empleados como su productividad o tarea actual, acerca de los artefactos que se producen como el porcentaje de avance en su construcción, acerca de los proyectos como su presupuesto, etcétera. Los empleados se comunican con el jugador para hacerle saber que trabajo desempeñan o si ocurren eventos inesperados o incluso para renunciar, el jugador usa esta información para tomar decisiones y actuar acorde en la simulación; al terminar la simulación el jugador recibe su calificación de su desempeño.

II.II.I. LA SIMULACIÓN D6TIONS (DECISIONES).

En los cursos de ingeniería de software impartidos en la Facultad de Ingeniería de la Universidad Autónoma de San Luis Potosí, se ha utilizado una simulación original propuesta por el Dr. Héctor Gerardo Pérez González, implementada en forma de juego de roles en cursos a nivel licenciatura y maestría.

El juego se fundamenta en la toma de decisiones de los participantes, el entendimiento de las consecuencias de éstas pretende constituirse como la base de la experiencia de aprendizaje de la ingeniería de software por parte de sus jugadores.

ROLES:

- Coordinador: El Instructor del curso quien presenta las reglas del juego y supervisa el desarrollo del mismo.

- Cliente: El Instructor del curso quien representa durante todo el curso a el cliente y usuario final del sistema a desarrollar.
- Empresario: Una pequeña selección de alumnos que juegan en el rol de empresario.
- Desarrollador: Los alumnos del curso que no juegan el rol de empresario juegan como desarrolladores.

CONCEPTOS:

- Soft: Es la unidad de moneda con el que se hacen las transacciones económicas en el juego. Existe una "paridad soft/punto", en donde un punto equivale a una décima parte de la máxima calificación que se puede obtener en cada evaluación parcial dentro del curso de ingeniería de software. Esta paridad puede variar durante el transcurso del juego. La paridad inicial es 10 softs = 1 punto.
- Lapso de juego: Unidad de tiempo de juego durante el cual se efectúa una sesión del mismo. Este lapso puede ser de unos pocos minutos hasta lo que dure una sesión del curso (60 o 90 minutos).
- Periodo de proceso: Unidad de tiempo durante el cual no cambian algunos parámetros del juego tales como el sueldo para desarrolladores.

OBJETIVO DEL JUEGO:

- Desarrollar un sistema de software de alta calidad.

GANADORES:

- Empresa ganadora: La empresa que al haber desarrollado completamente el sistema de software solicitado haya obtenido la mayor utilidad económica.
- Desarrollador Ganador: El desarrollador que haya participado al menos el 50% del tiempo de juego evaluado, en una empresa y que haya obtenido la mayor utilidad económica.

El juego consta de las seis etapas siguientes:

1. Directions.- En esta etapa se presentan las reglas del juego a los alumnos, se conforman las empresas, se selecciona un proyecto a desarrollar y se elaboran los planes, presupuestos y calendarizaciones por parte de las empresas participantes. La idea central es enfrentar a los participantes a actividades tales como la negociación de presupuestos o calendarizaciones, así como la negociación de sueldos y contratación de personal.
2. Domain of Problem.- En esta etapa se establecen los procesos y modelos de desarrollo de software y se efectúan los trabajos de ingeniería de requerimientos y de análisis del sistema a desarrollar. En esta etapa comienzan a desarrollarse actividades específicas dentro de la ingeniería de software, en esencia se simula la etapa de análisis del sistema a desarrollar, como resultado las empresas entregan al final artefactos propios de esta etapa, ejemplo: la especificación de requerimientos de software (SRS), un cronograma de desarrollo, documentación del ámbito del proyecto y análisis de riesgos, etc.
3. Design.- En esta etapa se sigue el modelo de proceso de desarrollo de software dando énfasis al diseño del mismo. Los entregables para esta etapa son diagramas UML que describen el diseño del sistema.
4. Development.- En esta etapa se sigue el modelo de proceso de desarrollo de software dando énfasis a la tarea de programación (desarrollo) del mismo. Los entregables de esta etapa son los programas que componen el sistema.
5. Documentation.- En esta etapa se sigue el modelo de proceso de desarrollo de software dando énfasis a la tarea de documentación del mismo. Los entregables son los manuales de usuario y manuales técnicos que resumen el desarrollo hecho.

6. Delivery (entrega).- En esta etapa se efectúa la etapa de transición del sistema por parte de la instancia desarrolladora a la instancia que continuará con el mantenimiento del sistema de software. En esta etapa se efectúa la entrega, instalación y demostración del software funcionando.

En el desarrollo del juego existen periodos de pago que se procura coincidan con los periodos de evaluación. Es en estos periodos donde los softs se otorgan como pago a los servicios de desarrollo entre cliente, empresarios y desarrolladores. A la par de los periodos de pago, se realiza una dinámica que de forma aleatoria afecta las condiciones en que está trabajando cada empresa, en esta dinámica se saca una tarjeta que representa un problema y una decisión que tomar para los jugadores (Figura II-II), también se saca una tarjeta para cambiar la "paridad soft/punto" (Figura II-III), lo cual enfrenta a los jugadores con la decisión de conseguir puntos para la evaluación en ese momento o esperar a que la paridad les sea más favorable.

1.- CRISIS ECONOMICA
Sobreviene una crisis económica en todo el País.
Todas las empresas deben ser cautelosas
Para este periodo, sólo se puede gastar el 70% del pago del Cliente.
El monto de sueldos se debe adaptar.
La Empresa no puede adquirir deudas.

2.- Software sin Licencia
Su empresa ha sido acusada de utilizar software sin licencia. Debe ser precavido en gastos para enfrentar una posible multa.
Tome la decisión entre:
1. **Despida a dos desarrolladores y baje sueldos al resto en $100s. c/u**
2. **Pague multa de $300 softs y despida a un desarrollador.**

3.- Espionaje Industrial
Se sospecha que algunos de sus desarrolladores están revelando secretos de la empresa y vendiendo su tecnología
Tome la decisión entre:
1. **Despida a tres desarrolladores y contrate a uno nuevo.**
2. **Despida a un desarrollador y ahorre $500 softs en sueldos.**

4.- Cliente satisfecho hasta el momento
El cliente está sumamente contento con el avance del proyecto.
Tome la decisión entre:
1. **Reciba un bono de $200 softs.**
2. **El adelanto del periodo es de 40%.**

5.- Ajuste por parte del cliente
El cliente no requiere el software con demasiada urgencia.
Tome la decisión entre:
1. **Reciba un bono de 100 softs.**
2. **Acepte una semana de tolerancia para la entrega final.**

6.- Premio de Calidad
La empresa recibe un premio económico por la calidad de sus procesos.
Tome la decisión entre:
1. **Reciba un premio total de $150 softs para la empresa.**
2. **Otorgue el premio de $100 softs a cada uno de sus dos mejores desarrolladores.**

Figura II-II Ejemplos de tarjetas de decisión D6tions.

Figura II-III Ejemplos de tarjetas de paridad soft/punto D6tions.

II.III. TESIS PROPUESTA.

La Figura II-IV presenta un diagrama de clases UML que describe una abstracción básica de los elementos que intervienen en un ambiente de desarrollo o producción de software, este trabajo parte del supuesto de que con

tecnología multiagente se pueden modelar e implementar estos elementos individualmente, y hacerlos interactuar en su conjunto para generar un modelo dinámico que permita observar el ejercicio de la ingeniería de software.

Junto con el uso de tecnología multiagente, este trabajo propone dos adiciones principales a las propuestas presentadas en los trabajos analizados:

- Primero, dado que en las propuestas de simulación basadas en computadora (Drappa & Ludewig, 2000), (Shaw & Dermoudy, 2005) y (Navarro, 2010) sólo puede intervenir un usuario en un rol único, se considera importante que como en las propuestas de (Henry & LaFrance, 2006) el participante pueda asumir más de un rol en la simulación;
- Segundo, en los proyectos de clase usados en (Henry & LaFrance, 2006) y en la simulación d6tions, los productos o artefactos de software son construidos por los alumnos, y reflejan los conocimientos y habilidades de estos en distintos temas como manejo de tecnologías o de conceptos de ingeniería de software, así como el seguimiento de un proceso, en las simulaciones basadas en computadora, no hay evaluación o experimentación con otros conocimientos o habilidades, salvo con los que se relacionan al seguimiento de un proceso. Es importante incluir en la simulación por computadora la idea de que los artefactos producidos reflejan habilidad y conocimientos en diversas áreas de la ingeniería de software (análisis, diseño, programación, control de calidad, administración de proyectos), que tiene el participante.

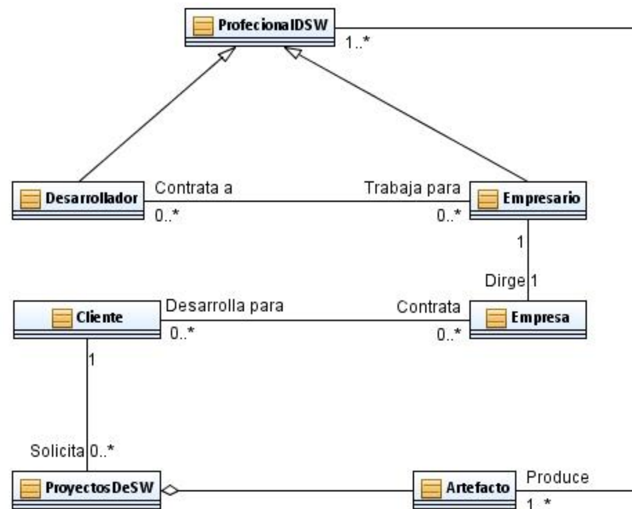


Figura II-IV Clases que intervienen en la producción de software.

III. SISTEMAS MULTIAGENTE.

Este tema presenta las definiciones de agente y sistema multiagente, teoría de agentes BDI, así como la tecnología multiagente JADEX, utilizada en la implementación del simulador propuesto.

III.I. AGENTE.

Agente. (Real Academia Española, 2010)
(Del lat. agens, -entis, part. act. de agĕre, hacer).
1. adj. Que obra o tiene virtud de obrar.
2. adj. Gram. Dicho de una palabra o de una expresión: Que designa a la persona, animal o cosa que realiza la acción del verbo. U. m. c. s. m.
3. m. Persona o cosa que produce un efecto.
4. m. Persona que obra con poder de otra.
5. com. Persona que tiene a su cargo una agencia para gestionar asuntos ajenos o prestar determinados servicios.
6. com. En algunos cuerpos de seguridad, individuo sin graduación.

Se consideran agentes a sistemas que están situados en algún ambiente, lo pueden percibir, y tienen un repertorio de posibles acciones que desempeñan, en orden de modificar el ambiente para alcanzar algún estado deseado.

Un agente está compuesto de ciertos elementos básicos. Estos incluyen uno a mas sensores que son usados para percibir el ambiente, uno a mas efectores o actuadores que manipulan el ambiente, y un sistema de control que provee mapeo entre sensores y efectores, y provee comportamiento inteligente o racional.

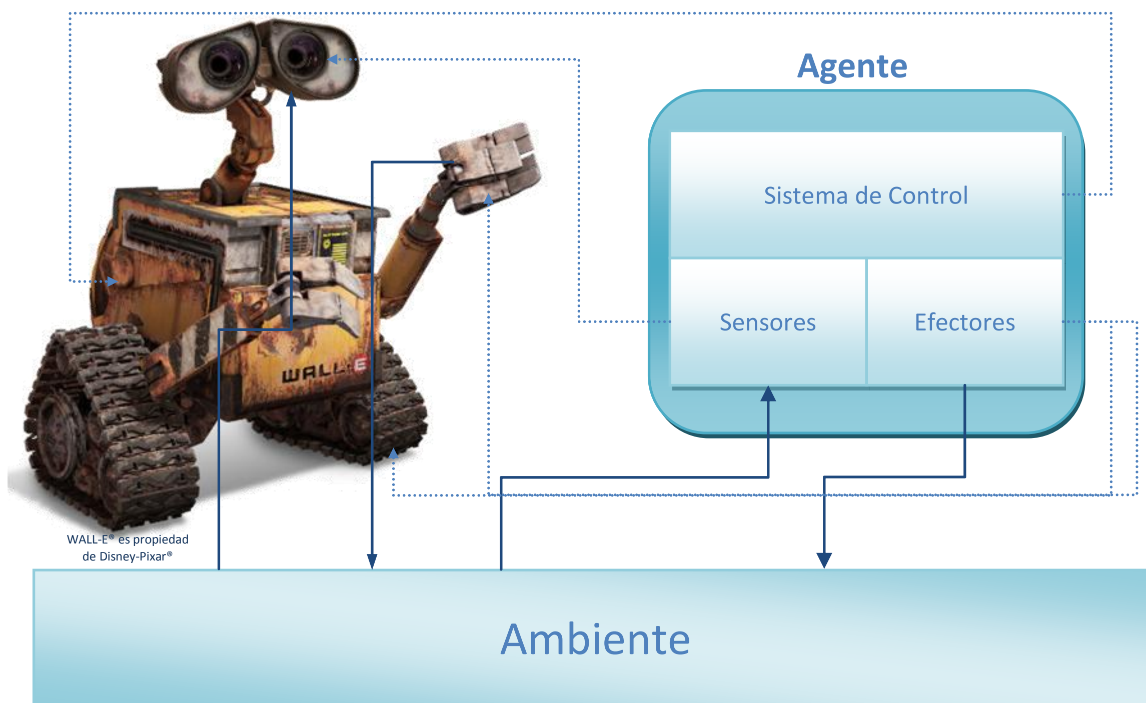


Figura III-I Anatomía básica de un agente.

Esta descripción puede ser aplicada a los humanos como primer ejemplo, el cuerpo humano es abundante en sensores que cubren una variedad de dominios, visión, olfato, audición, gusto, tacto, equilibrio, y otros. El cuerpo humano, también cuenta con ciertos sistemas efectores, incluyendo los dedos, las extremidades u otros sistemas motrices, gestos por ejemplo. El sistema de control incluye el cerebro y el sistema nervioso central.

La descripción también puede ser aplicada a otros tipos de agentes, ya sean virtuales físicos. Un robot puede ser considerado un agente físico, ya que puede tener una variedad de sensores, que incluyen cámaras, detectores infrarrojos, transductores ultrasónicos, micrófonos, o sensores de presión, temperatura, y movimiento. Sus efectores incluyen motores para mover ruedas, cintas, o extremidades, y bocinas para vocalizar sonidos, otros tipos más específicos de efectores pueden ser una aspiradora, un aspersor de agua, o incluso un arma.

III.II. AGENTES COMPUTACIONALES.

No existe una definición única para "agente" en el contexto de agentes virtuales, de software, o computacionales, sin embargo la mayoría de los autores concuerda en que se trata esencialmente de un componente de software o programa especial, que de forma autónoma provee una funcionalidad, trabajando para algún cliente/usuario. A partir de esta sección utilizaremos el término "agente" para referirnos a agentes virtuales o computacionales, específicamente agentes de software de forma general.

Como definición diremos que:

- Un agente es un sistema computacional que es situado en algún ambiente, y es capaz de actuar de forma autónoma en este ambiente para alcanzar objetivos para los que es diseñado.

III.II.I. PROPIEDADES DE UN AGENTE DESDE EL PUNTO DE VISTA DE LA INTELIGENCIA ARTIFICIAL

Lo que hace que un programa pueda ser considerado como agente, es que implemente una o más propiedades que exhiban o aparenten "inteligencia".

Propiedad	Descripción
Racional	Capacidad para actuar en forma racional o inteligente
Autónomo	Capacidad de actuar de forma independiente, sin estar sujeto a control externo
Persistente	Capacidad de actuar continuamente
Comunicativo	Capacidad de proveer o intercambiar información u órdenes con otros
Cooperativo	Capacidad de trabajar con otros para alcanzar objetivos
Móvil	Capacidad de moverse, generalmente relacionado a viajar dentro de una red
Adaptable	Capacidad de aprender y adaptarse

Tabla III-I Propiedades de un agente.

La propiedad de racionalidad significa que el agente ejecuta la acción correcta en el momento correcto, dado un resultado conocido. Esto depende de las acciones que están a disposición del agente, si estas pueden alcanzar el mejor resultado, y de cómo es evaluado el actuar del agente.

Autonomía significa que al agente le es posible moverse en su ambiente sin la guía de una entidad externa a él, como un operador humano. Por lo tanto el agente autónomo puede perseguir sus metas en su ambiente, mantenerse a sí mismo, o resolver problemas.

Persistencia implica la existencia del agente, en su ambiente y en el tiempo, de manera continua.

La capacidad de comunicación, permite a los agentes intercambiar información con otros agentes, o comunicarse con usuarios.

La capacidad de comunicación y cooperación están estrechamente relacionadas. La propiedad de cooperación, implica que un agente puede trabajar con otros agentes colectivamente para resolver problemas en un ambiente. Con el fin de cooperar, los agentes deben de tener la habilidad de comunicarse de alguna forma. Algunos autores señalan una propiedad contraria a cooperar, “decepcionar” (“deception”, engañar, defraudar, mentir, tomar el pelo) se refiere a que, en lugar de comunicar para resolver un problema, un agente comunica “desinformación” para engañar a otro agente.

La movilidad de un agente es comúnmente definida como la habilidad de migrar entre sistemas sobre una red. Esto puede hacerlo de forma autónoma, usando algún framework que soporte esta funcionalidad. Un ejemplo de esto son los virus, los cuales usan SMTP o HTTP para moverse entre sistemas.

Probablemente la más destacada de las propiedades de un agente es la habilidad de aprender y adaptarse al ambiente. Desde la perspectiva del agente, aprender significa crear una relación de lo que ocurre entre sensores y efectores, que demuestre comportamiento inteligente, o comportamiento que satisface un conjunto de restricciones dadas.

III.II.II. AMBIENTE.

Ambiente. (Real Academia Española, 2010)
(Del lat. ambĭens, -entis, que rodea o cerca).
1. adj. Dicho de un fluido: Que rodea un cuerpo.
2. m. Aire o atmósfera.
3. m. Condiciones o circunstancias físicas, sociales, económicas, etc., de un lugar, de una reunión, de una colectividad o de una época.

Todo agente existe y actúa dentro de un ambiente, para agentes virtuales su ambiente puede ser internet, el escenario o paisaje virtual en un video juego, o el espacio que rodea el ambiente de un problema determinado.

La gama de ambientes que pueden ser generados en el contexto de sistemas basados en agentes o inteligencia artificial pudiera ser difícil de delimitar. (Norvig & Russell, 2003), sugieren diversas dimensiones entre las cuales se pueden catalogar los ambientes, estas dimensiones son útiles al momento de determinar el diseño del agente o la técnica para su implementación.

Completamente observable vs Parcialmente observable.

El ambiente es completamente observable, si los sensores del agente le proveen acceso al estado completo del ambiente en cada punto del tiempo, es decir, si los sensores detectan todos los aspectos que son relevantes para elegir una acción para los efectores. Los ambientes completamente observables son convenientes por que el agente no necesita mantener ningún estado interno para seguir la pista del estado del ambiente. Un ambiente puede ser parcialmente observable debido a ruido o inexactitud en los sensores, o debido a que parte de la información del estado del ambiente no es accesible a los sensores.

Determinísticos vs Estocásticos.

El ambiente es determinístico si el siguiente estado del ambiente es completamente determinado por el estado actual y la acción ejecutada; de otra forma es estocástico. En principio, un agente no tiene que preocuparse de incertidumbre en ambientes completamente observables y determinísticos. Si el ambiente es parcialmente observable, entonces podría parecer estocástico, sobre todo si el ambiente es muy complejo, lo que hace difícil

rastrear todos los aspectos no observables. Por lo tanto es recomendable pensar en si el ambiente es determinístico o estocástico desde el punto de vista del agente. Si el ambiente es determinístico excepto por las acciones de otros agentes en el, entonces se dice que es un ambiente estratégico.

De episodios vs Secuencial.

El ambiente es de episodios si la experiencia (o el actuar), del agente está dividido en episodios atómicos. Cada episodio consiste en el agente percibiendo y entonces ejecutando una acción. Crucialmente el siguiente episodio no depende de las acciones efectuadas en episodios previos. A diferencia, en los ambientes secuenciales, las decisiones actuales pueden afectar las decisiones futuras.

Estático vs Dinámico.

Un ambiente es dinámico si su estado puede cambiar mientras el agente está deliberando, de otra forma es un ambiente estático. Los ambientes estáticos son fáciles de abordar porque el agente no necesita permanecer observando el ambiente mientras decide por una acción, ni necesita preocuparse por el paso del tiempo. Los ambientes dinámicos, por otra parte, continuamente preguntan al agente que acción va a efectuar, el tiempo que le toma decidir, es tomado como tiempo en el que no hace nada. Si el ambiente en si no cambia con el paso del tiempo, pero el tiempo afecta la evaluación del desempeño de agente, entonces se dice que el ambiente es semidinámico.

Discreto vs Continuo.

La distinción entre discreto y continuo se aplica al estado del ambiente, a la forma en que el tiempo es manejado, y a las percepciones y acciones del agente. Un ambiente discreto es aquel en el que existe un número finito de estados y un conjunto definido de percepciones y acciones, un ambiente continuo se relaciona a la continuidad de las acciones y percepciones a través del tiempo, lo cual genera un estado único-continuo.

Agente simple vs Multiagente.

Esta distinción es obviamente sobre la cantidad de agentes en el ambiente uno o más de uno. Sin embargo presenta algunos aspectos a considerar desde el punto de vista de abstracción. Desde el punto de vista de un agente A, que tiene que tratar con otro agente B, el primero podría tratar al segundo simplemente como un objeto con comportamiento sofisticado dentro del ambiente que comparten. Sin embargo pudiera existir una diferencia clave en este trato, cuando el comportamiento del agente B, afecta la funcionalidad para la cual fue diseñado el agente A, lo que obliga a A considerar a B como un agente creando así un sistema multiagente.

III.III. SISTEMAS MULTIAGENTE.

Un sistema multiagente es un sistema compuesto de agentes interactuando en cooperación o competencia, con el fin de alcanzar objetivos comunes o individuales.

Actualmente los sistemas multiagente son utilizados en diversos campos de investigación y son considerados como una forma interesante y conveniente de entender, modelar, diseñar e implementar diferentes tipos de sistemas, (Chalamish, Sarne, & Kraus, 2007), (Khan, Makkena, McGeary, Decker, Gillis, & Schmidt, 2003). Los sistemas multiagente representan también una alternativa al modelado basado en técnicas matemáticas, al permitirnos representar y simular sistemas reales o virtuales que pueden ser descompuestos en individuos y sus interacciones.

Desde el punto de vista de ingeniería de software, una de las características más importantes de los sistemas multiagente es que, ese conjunto final de agentes generalmente no se establece en el diseño del sistema, donde sólo es especificado un conjunto inicial, sino en tiempo de ejecución. Por esto, en la práctica, los sistemas multiagente están basados en arquitecturas abiertas que permiten a los agentes unirse y abandonar el sistema en forma dinámica, demostrando su autonomía, y comportamiento proactivo que no es completamente predecible a priori. En estos ambientes computacionales heterogéneos, abiertos y distribuidos donde los agentes deben de interactuar y traspasar diferentes plataformas para acceder a recursos, la interoperabilidad es un factor clave, para asegurar esta interoperabilidad, diversas organizaciones han desarrollado estándares que se enfocan en diferentes aspectos de la implementación de sistemas multiagente. Como ejemplos: DARPA - Knowledge Sharing Effort, OMG - Mobile Agent System Interoperability Facility y FIPA - Foundation for Intelligent Physical Agents. Las tecnologías para el desarrollo del simulador propuesto por esta tesis, se alinean con los estándares FIPA por lo que a continuación se presentan brevemente.

III.III.I. FIPA (FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS)

FIPA desarrolla estándares que principalmente describen: el manejo de agentes, la comunicación entre agentes y la arquitectura de sistemas de agentes. A la fecha se han emitido 25 especificaciones (Tabla III-II), que pueden ser consultadas en (IEEE Foundation for Intelligent Physical Agents, 2010).

Identifier	Title
SC00001	FIPA Abstract Architecture Specification
SC00008	FIPA SL Content Language Specification
SI00014	FIPA Nomadic Application Support Specification
SC00023	FIPA Agent Management Specification
SC00026	FIPA Request Interaction Protocol Specification
SC00027	FIPA Query Interaction Protocol Specification
SC00028	FIPA Request When Interaction Protocol Specification
SC00029	FIPA Contract Net Interaction Protocol Specification
SC00030	FIPA Iterated Contract Net Interaction Protocol Specification
SC00033	FIPA Brokering Interaction Protocol Specification
SC00034	FIPA Recruiting Interaction Protocol Specification
SC00035	FIPA Subscribe Interaction Protocol Specification
SC00036	FIPA Propose Interaction Protocol Specification
SC00037	FIPA Communicative Act Library Specification
SC00061	FIPA ACL Message Structure Specification
SC00067	FIPA Agent Message Transport Service Specification
SC00069	FIPA ACL Message Representation in Bit-Efficient Specification
SC00070	FIPA ACL Message Representation in String Specification
SC00071	FIPA ACL Message Representation in XML Specification
SC00075	FIPA Agent Message Transport Protocol for IIOP Specification
SC00084	FIPA Agent Message Transport Protocol for HTTP Specification
SC00085	FIPA Agent Message Transport Envelope Representation in XML Specification
SC00088	FIPA Agent Message Transport Envelope Representation in Bit Efficient Specification
SI00091	FIPA Device Ontology Specification
SC00094	FIPA Quality of Service Specification

Tabla III-II estándares emitidos por la FIPA.

El diseño e implementación interna de agentes inteligentes y la infraestructura física para ejecutarlos, no son normados por FIPA, esta sólo desarrolla estándares que aseguran la interoperabilidad de los sistemas de agente.

III.III.I.I. MANEJO DE AGENTES (FIPA).

El modelo de referencia para el manejo de agentes, provee un framework normativo dentro del cual los agentes FIPA existen y operan. Este establece el modelo de referencia lógico para la creación, registro, ubicación, comunicación, migración y destrucción de agentes. Las entidades contenidas en el modelo de referencia son conjuntos de capacidades lógicas (servicios) y no implican ninguna configuración física. El modelo de referencia de manejo de agentes consiste de los componentes que muestra la Figura III-II.

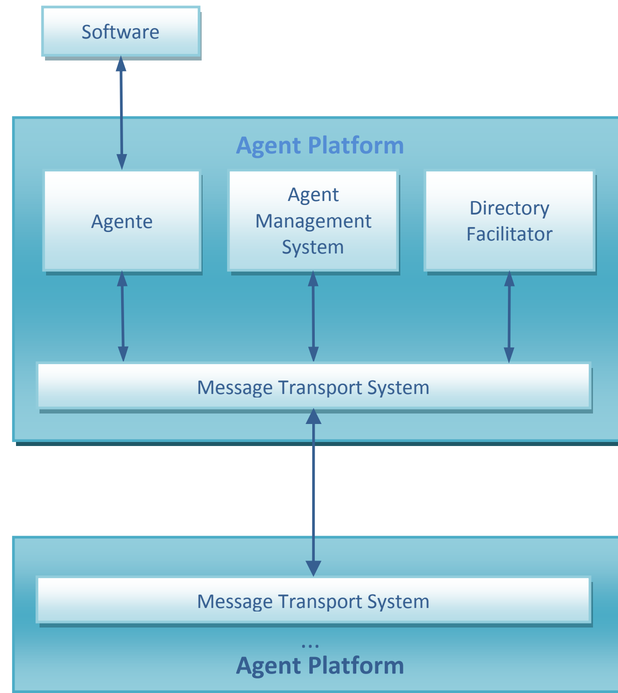


Figura III-II Modelo de referencia FIPA para el manejo de agentes.

La *Agent Platform* (*AP* - Plataforma de Agentes) provee la infraestructura física en la cual los agentes son desplegados. La AP consiste de la maquina (o maquinas), el sistema operativo, el software de soporte para agentes, los componentes de manejo FIPA (el Agent Management System, el Directory Facilitator, y el Message Transport Service) y los agentes mismos.

El diseño interno de la AP es un tema de implementación para quien la desarrolla y no está sujeta a estandarización, FIPA sólo se ocupa de cómo la comunicación es transportada entre agentes dentro de la plataforma, y agentes fuera de la plataforma. Los agentes son libres de intercambiar mensajes directamente o por cualquier medio que dispongan.

El *Agent Management System* (*AMS* - Sistema de Manejo de Agente) es un componente obligado y solamente debe existir uno en cualquier AP. Cada agente debe de registrarse para obtener un Agent Identifier (AID - identificador de agente), el AMS mantiene un directorio de AIDs el cual contiene direcciones (entre otros elementos) para los agentes registrados en la AP. El AMS ofrece el servicio de páginas blancas (sección blanca en un directorio telefónico) para otros agentes.

El *Directory Facilitator* (*DF* - Facilitador de Directorio) es un componente opcional de la AP, cuando está presente debe implementarse como un servicio de directorio. El DF provee servicio de páginas amarillas (como

la sección amarilla en un directorio telefónico) para otros agentes. Los agentes pueden registrar los servicios que ofrecen en el DF o consultarlo para encontrar servicios ofrecidos por otros agentes. Una AP puede tener múltiples DF.

El *Message Transport Service* (*MTS* - Servicio de Transporte de Mensajes) es el método de comunicación entre agentes predeterminado. Es un elemento obligado en cualquier AP FIPA.

Los *Agentes* son procesos computacionales (que analizamos al principio de este tema) que implementan la funcionalidad autónoma y comunicativa. Los agentes son los actores fundamentales en una AP, deben tener al menos un propietario (un usuario humano o una organización), y soportar al menos una noción de identidad. Esta noción de identidad es el Agent Identifier (AID) registrado en el AMS.

En la descripción del modelo en la Figura III-II, *Software* se refiere al todo lo no-agente, colecciones de instrucciones accesibles a través de un agente, de ahí su relación con la plataforma.

III.III.I.II. COMUNICACIÓN ENTRE AGENTES (FIPA).

Los agentes son una forma de procesos de código distribuidos, por lo tanto cumplen con la noción clásica de modelo de computación distribuida, compuesta de dos partes; componentes y conectores. Los componentes son consumidores, productores y mediadores de intercambio de mensajes de comunicación vía los conectores.

Los mensajes son la forma fundamental de comunicación entre agentes. La estructura de un mensaje es un conjunto de valores clave escritos en FIPA-ACL (FIPA Agent Communication Language - Lenguaje de comunicación de agentes). El contenido de un mensaje es expresado en un lenguaje de contenido (como FIPA-SL o FIPA-KIF), y estos contenidos pueden ser fundamentados en ontologías de referencia. Los mensajes pueden contener otros mensajes dentro (de forma recursiva) y deben contener parámetros clave como el AID del receptor y el AID del transmisor. Previo a su transmisión cada mensaje se codifica en un sobre de transporte (transport envelope) para el protocolo particular a utilizar.

Estructura de mensaje FIPA-ACL.

Parámetro	Descripción
`performative`	Tipo de acto de comunicación del mensaje
`Sender`	Identidad del agente que envía el mensaje
`Receiver`	Identidad del agente que se intenta reciba el mensaje
`reply-to`	Agente al cual se deben dirigir los mensajes subsecuentes dentro de un hilo de conversación.
`Content`	Contenido del mensaje
`Language`	Lenguaje en el cual está expresado el parámetro content en el mensaje
`Encoding`	Especifica el formato de codificación del mensaje
`Ontology`	Referencia a una ontología que da significado a los símbolos contenidos en el parámetro content del mensaje
`Protocol`	Protocolo de interacción usado para estructurar una conversación
`conversation-id`	Identificación única para un hilo de conversación
`reply-with`	Una expresión que debe ser usada por un agente que responde para identificar el mensaje
`in-reply-to`	Referencia a una acción anterior para la cual el mensaje es respuesta
`reply-by`	Un tiempo/fecha que indica cuando debe ser respondido un mensaje

Tabla III-III Parámetros de un mensaje FIPA-ACL.

Un mensaje FIPA-ACL contiene uno o más parámetros de mensaje. Los parámetros necesarios para la comunicación efectiva entre agentes puede variar de acuerdo a la situación; el único parámetro obligatorio en cualquier mensaje FIPA-ACL es el `performative`, aunque es común que la mayoría de los mensajes contengan también los parámetros `sender`, `reciever`, y `content`. La Tabla III-III muestra los parámetros

que componen un mensaje FIPA-ACL. Para la codificación de los mensajes FIPA define tres formatos: String (EBNF notation), XML. Y Bit-Efficient.

Actos comunicativos.

FIPA-ACL define la comunicación en términos de una función o acción, llamado acto comunicativo (CA-communicative act), realizado por el acto de la comunicación. En general los actos comunicativos están basados en la teoría de los actos del habla (Searle, 1969), la cual define las funciones de simples acciones específicas al hablar. Estas funciones están especificadas en FIPA CA Library specification (Tabla III-IV); por ejemplo, incluye funciones interrogativas las cuales requieren por información, imperativas las cuales solicitan la ejecución de una acción, referenciales las cuales comparten información relativa al ambiente, paralingüísticas las cuales relacionan un mensaje a otros mensajes, y expresivas las cuales expresan actitudes, intenciones o creencias. Un mensaje puede desempeñar varias funciones al mismo tiempo.

Acto comunicativo FIPA	Descripción.
Accept Proposal	La acción de aceptar la propuesta de otro agente para ejecutar una tarea.
Agree	La acción de aceptar desempeñar una tarea, posiblemente en el futuro.
Cancel	El emisor informa al receptor que desea que cancele la ejecución de una tarea que le solicito previamente.
Call for Proposal	La acción de solicitar propuestas a otros agentes para desempeñar una tarea.
Confirm	El emisor, confirma al receptor que una proposición en cierta (el receptor asumía que la proposición era cierta y necesitaba confirmación).
Disconfirm	El emisor no confirma al receptor que una proposición es cierta (el receptor asumía que la proposición era cierta y necesitaba confirmación).
Failure	La acción de avisarle a otro agente que falló el intento de ejecutar una tarea solicitada previamente.
Inform	El emisor informa al receptor que una proposición dada es verdadera.
Inform If	El emisor informa al receptor si una proposición es cierta o falsa.
Inform Ref	El emisor informa al receptor sobre un objeto referido en una expresión.
Not Understood	El emisor 'i' informa al receptor 'j', que percibe que 'j' ha desempeñado alguna acción, pero 'i' no entendió lo que 'j' acaba de hacer. El caso común 'i' no entiende el mensaje que 'j' acaba de enviarle.
Propagate	El emisor quiere que el receptor propague el mismo mensaje a otros agentes que cumplen con una descripción dada.
Propose	La acción de enviar una propuesta para ejecutar cierta acción, y proponer ciertas condiciones para su ejecución.
Proxy	El transmisor quiere que el receptor seleccione a otros agentes dados en una descripción y les reenvié un mensaje incrustado dentro del mensaje principal.
Query If	La acción de preguntar a otro agente si una proposición es cierta.
Query Ref	La acción de preguntar a otro agente por el objeto referido por una expresión.
Refuse	La acción de rehusar ejecutar una tarea, y explicar porque se rehúsa.
Reject Proposal	La acción de rechazar la propuesta hecha por otro agente para ejecutar una tarea.
Request	El emisor requiere al receptor para que ejecute una tarea.
Request When	El emisor requiere al receptor para que ejecute una tarea si una proposición dada en cierta.
Request Whenever	El emisor requiere al receptor para que ejecute una tarea tan pronto como una proposición dada se vuelva cierta, y repita esta tarea cada que dicha proposición se vuelva cierta otra vez.
Subscribe	El emisor requiere al receptor para que lo notifique persistentemente cada que el valor de una referencia dada cambie.

Tabla III-IV Actos comunicativos FIPA

```
(request
        :sender (agent-identifier :name alice@mydomain.com)
        :receiver (agent-identifier :name bob@yourdomain.com)
        :ontology travel-assistant
        :language FIPA-SL
        :protocol fipa-request
        :content""((action(agent-identifier :name bob@yourdomain.com)(book-hotel
                :arrival 15/10/2006:departure 05/07/2002 ... )))""
)
```

Figura III-III Ejemplo de un mensaje FIPA-ACL con el acto comunicativo "request".

Protocolos de interacción.

Basados en los actos comunicativos, FIPA ha definido un conjunto de protocolos de interacción, cada uno consiste en una secuencia de actos para coordinar acciones multimensaje. Los estándares FIPA definen nueve protocolos de interacción. Como ejemplos:

FIPA Request Interaction Protocol

- El FIPA Request Interaction Protocol permite a un agente, el iniciador, requerir a otro, el participante, que ejecute una tarea. El participante procesa el requerimiento y hace una decisión de cuando aceptar o rehusar ejecutar la tarea.
  Si las condiciones indican que se debe notificar si se acepta o no el requerimiento (el parámetro `notification necessary` en el mensaje, es `true`), entonces el agente participante debe comunicar si acepta o rehúsa ejecutar la tarea. Una vez que el requerimiento ha sido atendido, el agente participante debe comunicar ya sea:
    - Un mensaje de falla (`fialure`), si algo falló al intentar completar la tarea,
    - Un mensaje que informa que fue hecho (`inform-done`), si completo la tarea exitosamente, o
    - Un mensaje que informa el resultado (`inform-result`), para indicar: que completo la tarea exitosamente, y el resultado de lo hecho.

  Cualquier interacción dentro de este protocolo es identificada de forma única por el valor del parámetro `conversation-id`, asignado por el agente iniciador, los agentes involucrados en la interacción deben etiquetar todos sus mensajes con este valor.
  En cualquier punto del protocolo, el receptor de una comunicación puede informar al transmisor que no entiende lo que se le está comunicando, enviando un mensaje `not-understood`; la comunicación de un mensaje `not-understood` termina toda interacción dentro del protocolo, lo cual implica que cualquier compromiso hecho durante la interacción es anulado.

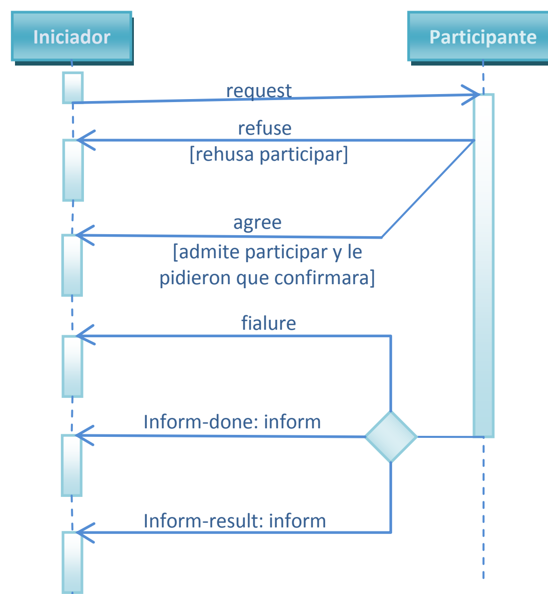


Figura III-IV FIPA - Request interaction protocol

Adicionalmente, en cualquier punto de la ejecución del protocolo, el iniciador puede cancelar la interacción al iniciar el meta protocolo para cancelar: `cancel`. El parámetro `conversation-id` de la interacción `cancel` es el mismo que el utilizado en la interacción que el iniciador pretende cancelar.

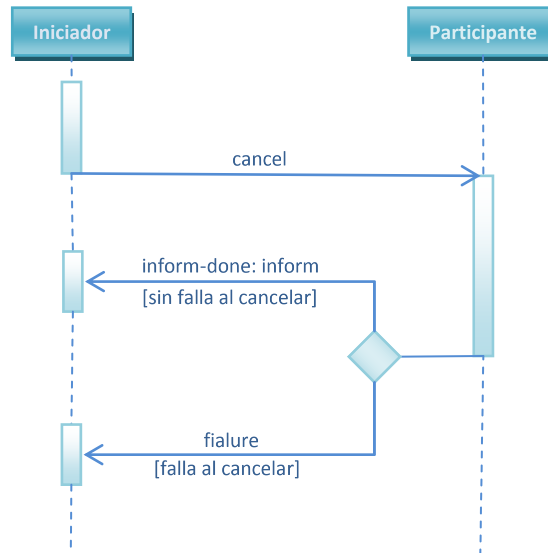


Figura III-V FIPA - Cancel Meta Protocol

FIPA Contract Net Interaction Protocol

- El FIPA Contract Net Interaction Protocol describe el caso en que un agente, el iniciador, desea que una tarea sea realizada por uno a mas agentes, el/los participantes; y además desea optimizar la función que caracteriza esta tarea. Tal función característica es expresada comúnmente en costos, pero también puede ser en tiempos, o en distribución equitativa de tareas.
  Para una tarea dada, cualquier número de participantes pueden responder con una propuesta; el resto puede rehusar participar. Las negociaciones continúan con los participantes que emitieron propuestas.
  El agente iniciador solicita *m* propuestas de los otros agentes usando un `call for proposal` el cual especifica la tarea y las condiciones que el iniciador necesita conocer para seleccionar a quien o quienes la ejecutaran.
  Los participantes que reciben el `call for proposals` son vistos como potenciales contratados y se habilitan para generar *n* respuestas; de estas, *j* son propuestas para desempeñar la tarea. Las propuestas de los participantes incluyen las condiciones establecidas para ejecutar la tarea (requeridas por el iniciador), estas pueden ser el precio, el tiempo, etcétera. Alternativamente, los $i=n-j$ participantes pueden rehusar hacer una propuestas.
  Este protocolo de interacción requiere que el iniciador conozca cuando ha recibido todas las repuestas a sus mensajes. Si un participante falla al responder, ya sea con una propuesta o rehusándose a participar, el iniciador podría, potencialmente, quedarse esperando indefinidamente; para prevenir esto, el mensaje `call for proposals` incluye un tiempo límite en el cual las respuestas deben ser recibidas por el iniciador. Las propuestas que se reciben después de este tiempo, son rechazadas automáticamente. El tiempo límite es especificado en el parámetro `reply-by` en el mensaje.

Una vez que el tiempo límite se supera, el iniciador evalúa las *j* propuestas recibidas, y selecciona los agentes que desempeñaran la tarea; uno, varios o ningún agente pueden ser elegidos. Los *l* agentes de las propuestas seleccionadas recibirán un `accept-proposal` (propuesta aceptada) y los *k* restantes recibirán un `reject-proposal` (propuesta rechazada). Las propuestas están ligadas a los participantes, por lo tanto una vez que el iniciador acepta una propuesta, el participante adquiere el compromiso de ejecutar la tarea. Una vez que el participante ha completado la tarea, envía un mensaje al iniciador como forma de un `inform-done` o si se requiere más explicación `inform-result`. Si el participante falla en el intento de completar la tarea, un mensaje `fialure` es enviado.

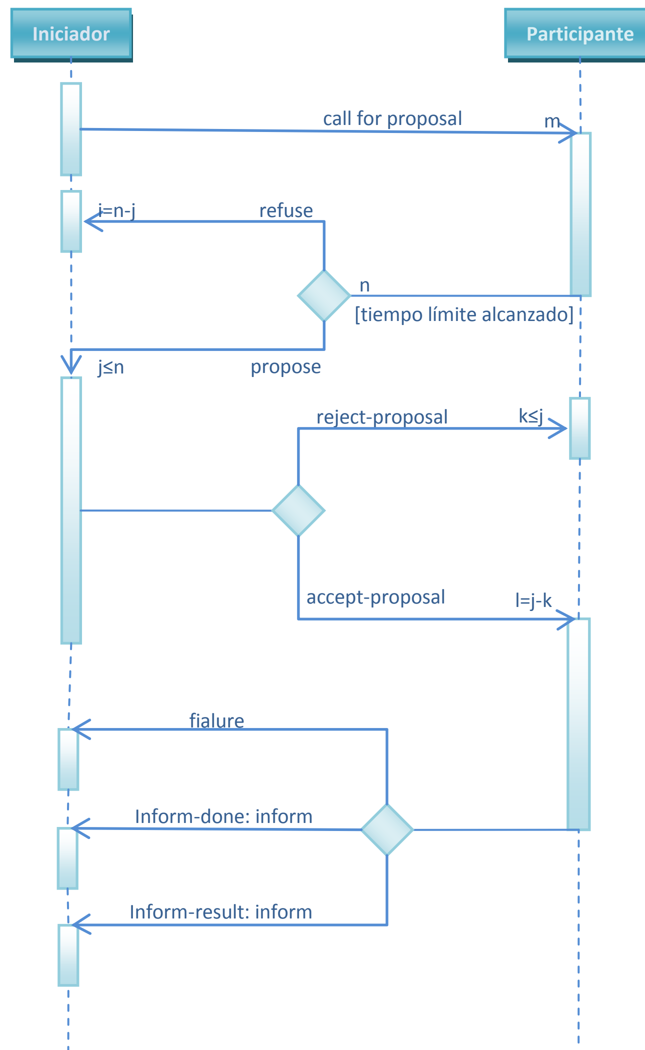


Figura III-VI FIPA Contract Net Interaction Protocol[4]

Los otros protocolos son: Query Interaction Protocol, Request When Interaction Protocol, Iterated Contract Net Interaction Protocol, Brokering Interaction Protocol, Recruiting Interaction Protocol, Subscribe Interaction Protocol, Propose Interaction Protocol; cada uno define la interacción entre agentes en distintos escenarios.

[4] Los diagramas están basados en una extensión de UML propuesta en (Odell, Van Dyke Parunak, & Bauer, 2001).

III.IV. TECNOLOGÍA UTILIZADA

III.IV.I. AGENTES BDI JADEX

JADEX es una plataforma para desarrollar agentes racionales BDI con XML y Java. Actualmente Jadex Agents forma parte de Jadex Software Projects, desarrollados en el Departamento de Informática, de la Facultad de Matemáticas, Informática y Ciencias Naturales de la Universidad de Hamburgo (Pokahr, 2010).

Los agentes JADEX extienden las prácticas de la programación orientada a objetos. La programación orientada a agentes representa un nivel de abstracción nuevo y agrega el concepto explicito de actores autónomos a el mundo de objetos pasivos. A nivel de implementación, los agentes representan componentes de software activos con capacidades de razonamiento individual; es decir, los agentes pueden exhibir comportamiento reactivo en respuesta a eventos externos, así como también comportamiento proactivo motivado por los objetivos propios del agente.

III.IV.I.I. ARQUITECTURA BDI.

Las arquitecturas de agente, son descripciones formales de los elementos que componen un sistema y la forma en que se interrelacionan. Estas arquitecturas pueden ser aplicadas en diferentes niveles a los sistemas de agentes.

La arquitectura BDI (Belief-Desire-Intention, Creencia-Deseo-Intención), sigue las teorías del razonamiento humano tal como la define (Bratman, 1987). Una creencia representa cómo ve el mundo un agente, lo que él cree o percibe del estado del ambiente en que existe. Un deseo es el objetivo que decide la motivación del agente, lo que quiere lograr o alcanzar, el agente podría tener varios deseos, los cuales debieran ser consistentes. Por último, las intenciones especifican que el agente utiliza sus creencias y sus deseos para elegir una o más acciones a ejecutar para alcanzar su objetivo.

La arquitectura BDI define la base de cualquier agente deliberativo. Este tipo de agente almacena una representación del estado del ambiente, mantiene un conjunto de objetivos y finalmente algún elemento intencional, que mapea creencias y deseos, para proveer una o más acciones que modifiquen el estado del ambiente según las necesidades del agente.

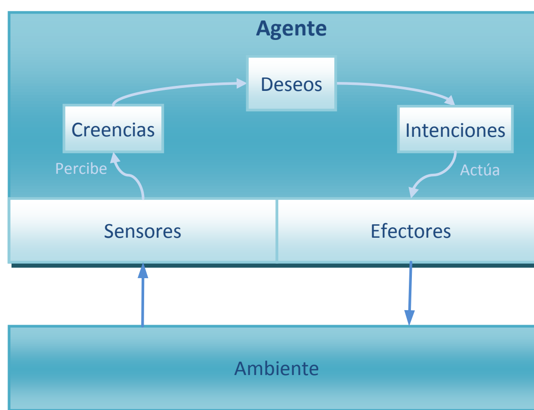


Figura III-VII Arquitectura BDI.

Un agente BDI es un tipo particular de agente de software con rasgos de racionalidad, provisto con actitudes mentales guiadas por sus creencias, deseos e intenciones (McCarthy, 1979).

Creencias (Beliefs): las creencias representan el estado informacional del agente, es decir lo que sabe o conoce acerca del ambiente o mundo que lo rodea, incluyendo a otros agentes y a sí mismo. Las creencias pueden también incluir reglas de inferencia, lo cual le permitiría al agente generar nuevas creencias. El uso del término creencia en lugar de conocimiento implica que lo que un agente cree no necesariamente es cierto, y la información que este da por hecho puede cambiar en el futuro. Las creencias son almacenadas en un repositorio llamado base de creencias o conjunto de creencias (belief set).

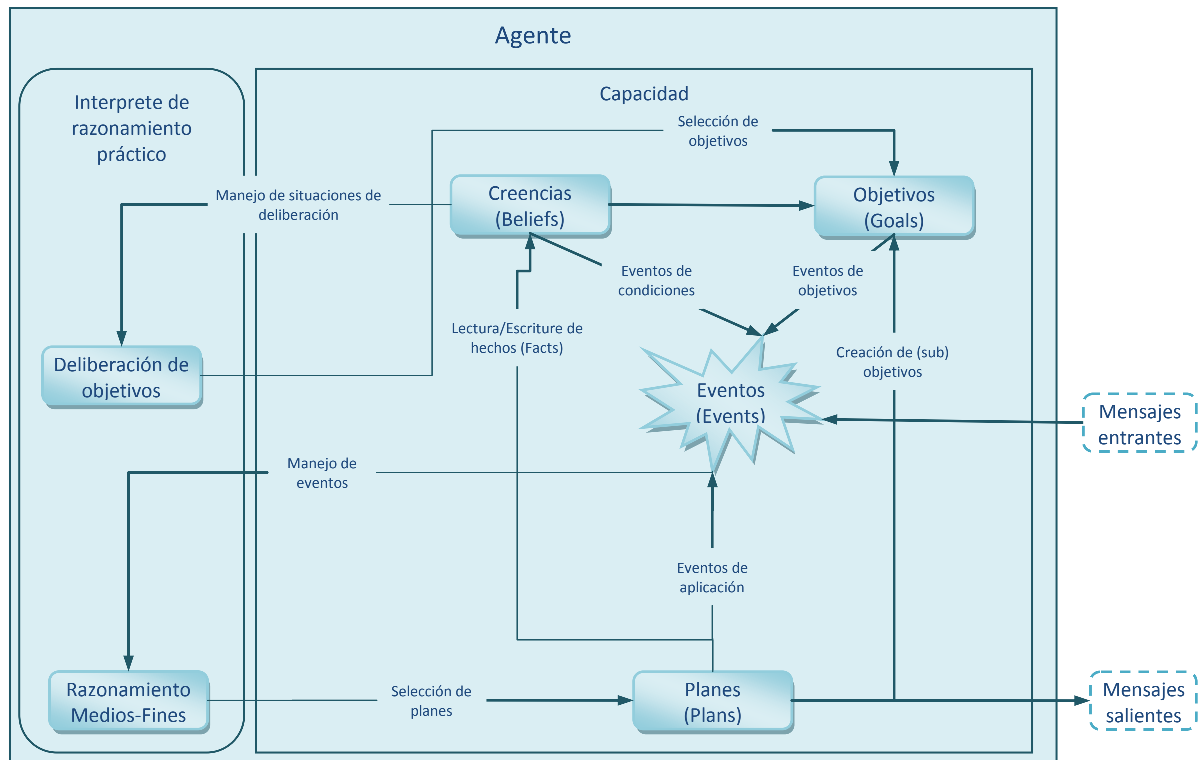


Figura III-VIII Arquitectura abstracta de un agente JADEX

Deseos (Desires): los deseos representan el estado motivacional del agente. Estos representan objetivos o situaciones que al agente le gustaría alcanzar o lograr, por ejemplo: conseguir trabajo, encontrar el mejor precio o volverse rico. Un objetivo (Goal) es un deseo que ha sido adoptado y el agente lo persigue de forma activa. El uso del término objetivo agrega una restricción para que las actividades que el agente emprenda sean consistentes con sus objetivos.

Intenciones (Intentions): las intenciones representan el estado deliberativo de el agente, lo que el agente elige hacer. Las intenciones son deseos para los cuales el agente tiene un compromiso. En sistemas implementados, esto significa que el agente comienza a ejecutar un plan. Los planes son secuencias de acciones que un agente puede desempeñar para lograr una o más de sus intenciones.

Eventos (Events): los eventos son los disparadores para que el agente emprenda actividades. Los eventos, pueden actualizar creencias, activar planes o modificar objetivos. Los eventos además pueden generase dentro del agente al cambiar su estado interno, o generarse fuera del agente y ser captados por sus sensores.

JADEX facilita el uso del modelo BDI en programación, al introducir creencias (beliefs), objetivos (goals) y planes (plans) como objetos que pueden ser creados y manipulados dentro del agente.

El razonamiento de los agentes JADEX es un proceso que interrelaciona dos componentes: por un lado, el agente reacciona a los mensajes entrantes, eventos internos y objetivos para seleccionar y ejecutar planes (razonamiento medios-fines); por otro lado, el agente delibera continuamente acerca de sus objetivos actuales, para decidir cuál es el subconjunto de acciones que debe adoptar, coherentes con sus objetivos. Las creencias, objetivos y planes que definen el comportamiento del agente, son definidos por el programador.

Como lo ilustra la Figura III-VIII, las creencias influencian la deliberación de objetivos y el proceso de razonamiento medios-fines; los planes pueden cambiar las creencias actuales al ser ejecutados o crear nuevos objetivos; el cambio en las creencias puede causar eventos internos los cuales pueden generar la adopción de nuevos objetivos y la ejecución de nuevos planes.

JADEX provee un Framework completo para desarrollo (API), así como herramientas para la operación y administración de su plataforma de agentes, que cumplen con las especificaciones FIPA (IEEE Foundation for Intelligent Physical Agents, 2010).

Para desarrollar sistemas con JADEX se trabaja con dos tipos de archivos: archivos de definición de agente en XML (ADF: Agent Definition Files) y clases Java para la implementación de planes. Los ADF pueden ser vistos como una clase que sirve para instanciar agentes; para ejecutar un agente, primero el ADF es cargado, y se inicializa el agente con las creencias, los objetivos y los planes que el ADF especifica

```xml
<agent  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
        xmlns='http://jadex.sourceforge.net/jadex'
        xsi:schemaLocation='http://jadex.sourceforge.net/jadex
http://jadex.sourceforge.net/jadex-0.96.xsd'
        name="Ambiente" package="d6tionsx.ambiente">

    <imports>
          …
    </imports>
<!--Capacidades -->
    <capabilities>
          …
    </capabilities>
 <!--Creencias-->
    <beliefs>
        …
    </beliefs>
<!--Objetivos-->
    <goals>
          …
    </goals>
 <!--Planes-->
    <plans>
        …
    </plans>
 <!--Expresiones-->
    <expressions>
          …
    </expressions>
<!—Configuraciones-->
    <configurations>
          …
    </configurations>

</agent>
```

Figura III-IX Ejemplo del ADF del agente Ambiente (Ambiente.agent.xml)

III.IV.I.II. DECLARACIÓN DE AGENTES JADEX

Todo ADF es un documento XML, y sigue el modelo especificado por el esquema JADEX-0.96.xsd, lo cual nos permite verificar que los ADFs no sólo sean documentos XML bien formados, sino que también sean ADFs validos. El nombre del agente es especificado en el atributo `name` de la etiqueta `<agent>`, y debe coincidir con el nombre que se le da al archivo, además todo ADF debe ser nombrado con la extensión `.agent.xml`. La Figura III-IX muestra la estructura general de un ADF.

III.IV.I.II.I. IMPORTS.

La etiqueta import es usada para especificar cuáles clases y paquetes pueden ser usados por expresiones dentro de un ADF, Figura III-X. Una sentencia import en JADEX tiene la misma sintaxis que un `import` en Java, y la misma funcionalidad que permite importar tanto clases como paquetes completos para su uso.

```
<agent  …
        name="Ambiente"
package="d6tionsx.ambiente">

    <imports>
        <import>jadex.runtime.*</import>
        <import>jadex.util.*</import>
        <import>jadex.planlib.*</import>

<import>jadex.adapter.fipa.*</import>
        <import>d6tionsx.*</import>
        <import>d6tions.Etapa.*</import>

<import>d6tionsx.ambiente.gui.*</import>
    </imports>
         …
</agent>
```

Figura III-X Imports.

III.IV.I.II.II. CAPABILITIES (CAPACIDADES).

```
<!--
   La df capability tiene todos los planes, creencias y objetivos para las actividades
   relacionadas con la funcionalidad del Directory Facilitator (DF - Facilitador de
   Directorio) del estándar FIPA.
-->
<capability xmlns="http://jadex.sourceforge.net/jadex"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://jadex.sourceforge.net/jadex
http://jadex.sourceforge.net/jadex-0.96.xsd"
        abstract="true"
        package="jadex.planlib"
        name="DF">

        <imports>
                <import>jadex.adapter.fipa.*</import>
        </imports>

        <beliefs>… </beliefs>

        <goals>… </goals>
        …
</capability>
```

Figura III-XI Implementación de capability (Df capability de JADEX).

En el contexto de JADEX, "capability" es un módulo que encapsula funcionalidad para usarse dentro de un agente. Las "capabilities" permiten empaquetar un subconjunto de creencias, planes y objetivos dentro de un módulo-agente. Se puede decir que funcionan de forma análoga a una clase y una subclase, un agente que utiliza una "capability" hereda la funcionalidad que esta contiene.

Al igual que los agentes, las capacidades se definen en un documento XML, sólo que la etiqueta raíz es `<capability>` en lugar de `<agent>`, los archivos XML de una capability se nombran con la extensión `.capability.xml`, Figura III-XI.

Los agentes y las capacidades pueden implementar cualquier número de sub capacidades, las cuales son referidas en la etiqueta `<capabilities>`. Para referenciar una capacidad, se provee un nombre local y la ubicación de la definición de la capacidad en los atributos `name` y `file` de la sub etiqueta `<capability>` (Figura III-XII).

III.IV.I.II.II.I. CAPACIDADES PREDEFINIDAS.

La versión 0.96 de JADEX contiene varias capacidades predefinidas para su uso. Las más importantes por su relación con los estándares FIPA son:

- Capacidad de manejo básico de agentes.- permite a los agentes usar las funcionalidades del AMS (Agent Management System), básicamente para administrar el ciclo de vida de otros agentes y para interactuar con la plataforma. Esta capacidad es usada concretamente para: crear agentes, iniciar agentes, destruir agentes, suspender agentes, restablecer agentes, buscar agentes y detener la ejecución de la plataforma de agentes.
- Capacidad para el facilitador de directorio. Permite a los agentes registrar sus servicios y buscar los servicios ofrecidos por otros agentes en el DF (Director Facilitator).

Capacidad de Protocolos. Permite a los agentes el uso de protocolos de interacción predefinidos por FIPA, espeficificamente: FIPA Request Interaction Protocol (RP), FIPA Contract Net Interaction Protocol (CNP), FIPA Iterated Contract Net Protocol (ICNP), FIPA English Auction Interaction Protocol (EA), FIPA Dutch Auction Interaction Protocol (DA), Abnormal Termination of Protocols. La capacidad contiene la funcionalidad tanto del lado iniciador como del participante de la interacción.

```
<agent  …
        name="Ambiente" package="d6tionsx.ambiente">

          …
    <capabilities>
        <!--Capacidades-->
        <capability  name="dfcap" file="jadex.planlib.DF"/>
        <capability name="procap" file="jadex.planlib.Protocols"/>
    </capabilities>

          …
</agent>
```

Figura III-XII Capabilities

III.IV.I.II.III. BELIEFS (CREENCIAS).

Las creencias (beliefs) representan el conocimiento del agente acerca de su ambiente y de sí mismo, estas son almacenadas en una base de creencias (belief base) que es un contenedor para los hechos conocidos por el

agente, y pueden ser referenciados en expresiones en el ADF como también pueden ser accedidos y modificados desde los planes. En JADEX las creencias pueden ser cualquier tipo de objeto Java.

Para definir una creencia de valor único se utiliza la etiqueta <belief>, para definir una creencia multi-valor se utiliza la etiqueta <beliefset> dentro de la etiqueta <beliefs> del ADF. Para cada <belief> o <beliefset> se debe de especificar un atributo name usado para referenciar el hecho contenido y un atributo class usado para especificar la clase del objeto que se guardará como un hecho (fact) de esa creencia. Cada hecho en una creencia o conjunto de creencias se denota con la etiqueta <fact> (Figura III-XIII).

```
<agent  …
        name="Ambiente" package="d6tionsx.ambiente">
          …
    <beliefs>

        <!--Relacionado con la Capability de Protocolos FIPA(procap)-->
        <beliefref name="rp_filter" class="IFilter">
            <concrete ref="procap.rp_filter"/>
        </beliefref>
        <!-- Etapa de la Simulación -->
        <belief class="int" name="estadoSimulacion">
            <fact>Etapa.INICIADA</fact>
        </belief>
          …
        <!-- Participantes en la Simulación -->
        <beliefset name="participantes" class="Participante"></beliefset>
        <beliefset name="desarrolladores" class="Desarrollador"></beliefset>
        <beliefset class="Empresario" name="empresarios"></beliefset>
        <beliefset class="Cliente" name="clientes"></beliefset>

        <!--Interfáz gráfica-->
        <belief class="GUIAmbiente" name="GUI">
            <fact>new GUIAmbiente($agent.getExternalAccess())</fact>
        </belief>

    </beliefs>
          …
</agent>
```

Figura III-XIII Beliefs.

III.IV.I.II.IV. GOALS (OBJETIVOS).

Los objetivos (goals) constituyen la naturaleza motivacional de un agente guían su comportamiento, estos son declarados dentro de la etiqueta <goals> en el ADF.

En tiempo de ejecución, un agente puede tener cualquier número de objetivos, así como sub objetivos. Los objetivos de nivel superior son creados cuando el agente nace (configurado en el estado inicial del agente en el ADF) o pueden ser adoptados después al generarse ciertas condiciones, mientras que los sub objetivos sólo pueden ser originados dentro de un plan ejecutándose. Independientemente de cómo se cree un objetivo, el agente tratará automáticamente de seleccionar los planes adecuados para alcanzarlo.

Uno de los aspectos del comportamiento racional es que un agente puede perseguir múltiples objetivos de forma paralela, JADEX provee un framework para decidir cómo los objetivos interactúan y cómo un agente puede decidir autónomamente qué objetivo perseguir. Este proceso es llamado deliberación de objetivos (goal deliberation), y es soportado por el ciclo de vida del objetivo (goal lifecycle). Para distinguir entre objetivos

simplemente adoptados y objetivos activamente perseguidos, el ciclo de vida del objetivo tiene los estados opción, activo y suspendido (Figura III-XIV). Cuando se adopta un objetivo, este se vuelve una opción que se agrega a los deseos del agente. El mecanismo de deliberación de objetivos se responsabiliza de administrar las transiciones de todos los objetivos adoptados. Adicionalmente algunos objetivos pueden ser válidos sólo en contextos específicos determinados por las creencias del agente; cuando el contexto de un objetivo no es válido, este puede ser suspendido hasta que el contexto sea valido nuevamente.

Al momento de declarar los objetivos dentro del ADF, se pueden configurar propiedades que influencian como el agente los maneja. En JADEX para cada objetivo se declara un nombre que lo identifica en el atributo `name`, a su vez se pueden declarar parámetros del objetivo dentro del ADF. La declaración de parámetros sigue la misma filosofía que la declaración de `<beliefs>`; se puede distinguir entre parámetros de valor único y parámetros multivalor. Los parámetros pueden ser de entrada (`in`), salida (`out`) o bidireccionales (`inout`) lo que es especificado en el atributo `direction` de la etiqueta `<parameter>`. Los parámetros de entrada, son establecidos antes de que el objetivo sea creado, los parámetros de salida son establecidos por el plan que procesa el objetivo.

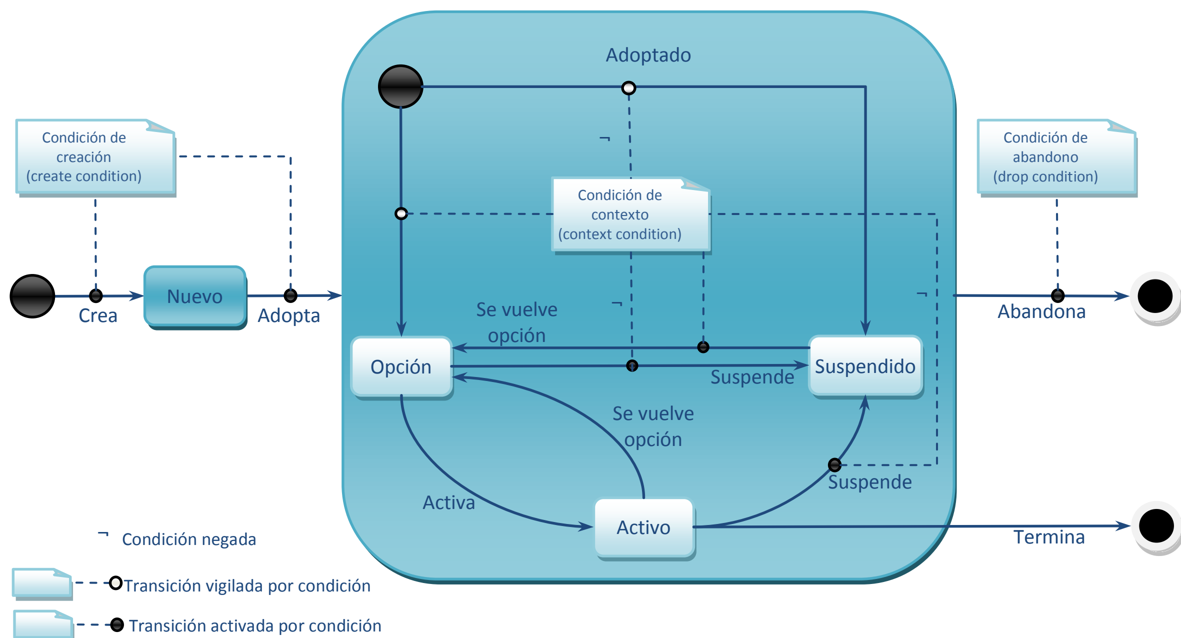


Figura III-XIV Ciclo de vida del objetivo.

La etiqueta `<creationcondition>` es utilizada para describir la situación en la cual un nuevo objetivo, de los declarados en el ADF, será automáticamente instanciado. Para un objetivo creado, se pueden especificar condiciones en las cuales este será suspendido o desechado usando las etiquetas `<contextcondition>` y `<dropcondition>` respectivamente; al suspender un objetivo, todos los planes ejecutándose son terminados, al cancelar la suspensión nuevos planes son instanciados y ejecutados; al desechar un objetivo este es removido y no puede ser reactivado.

La Tabla III-V muestra algunos de los atributos que se usan en las etiquetas de declaración `<goal>` para personalizar la forma en que los objetivos son tratados por el agente.

Atributo	Valores Posibles	Valor Default	Función
Retry	{true, false}	True	Indica que el objetivo debe ser reevaluado o rehecho hasta que sea alcanzado, o no haya más medios disponibles para perseguirlo.
Retrydelay	positive long value	0	Indica un tiempo de espera en milisegundos para reevaluar el objetivo.
Exclude	{"when_tried", "when_succeeded", "when_failed", "never"}	"when_tried"	Es usado en conjunto con Retry e indica que, al reevaluar un objetivo, los planes sólo deben llamarse cuando no han sido ejecutados para este objetivo.
Posttoall	{true, false}	False	Habilita el procesamiento paralelo de un objetivo, al notificarlo a todos los planes aplicables a la vez. El primer plan que completa el objetivo, "gana" y los otros planes son terminados. Si todos los planes son terminados sin alcanzar el objetivo, este se considera como fallido (failed).
randomselection	{true, false}	False	Se utiliza para elegir de forma aleatoria entre varios planes aplicables para un objetivo. La selección aleatoria es aplicada solamente a los planes que tienen el mismo rango y prioridad.
Recur	{true, false}	False	Se utiliza para indicar que el objetivo se debe de estar activando recursivamente.
Recurdelay	positive long value	0	Indica un tiempo de retardo en milisegundos para que un objetivo sea activado recursivamente.

Tabla III-V Atributos de la etiqueta <goal>.

JADEX 0.96 soporta cuatro diferentes tipos de objetivos:

- *Perform goal* (objetivo desempeñar): específica que algunas actividades deben ser desempeñadas, por lo tanto, el resultado de perseguir este tipo de objetivos depende solamente del hecho de si las actividades fueron desempeñadas. Se declara dentro de la etiqueta `<performgoal>`.
- *Achieve goal* (objetivo alcanzar): puede ser visto como un objetivo en el sentido clásico, ya que representa una situación deseada que necesita ser alcanzada. Se declara dentro de la etiqueta `<achievegoal>`. Para estos objetivos, se agrega una `<targetcondition>` y una `<fialurecondition>`. En `<targetcondition>` se puede especificar en qué casos un objetivo puede ser considerado como alcanzado, `<fialurecondition>` es útil para describir los casos de falla o cuando el objetivo no es alcanzado. Si no se especifica un `<targetcondition>` ni `<fialurecondition>`, el resultado de la ejecución de planes es usado para decidir si el objetivo se alcanzo. A diferencia de un perform goal, un achieve goal sin `<targetcondition>` es completado cuando el primer plan se completa sin error, mientras que el perform goal continúa hasta que todos los planes disponibles sean ejecutados.
- *Query goal* (objetivo consultar): es usado para consultar información acerca de un tema específico. Se declara dentro de la etiqueta `<querygoal>`. De forma parecida al achieve goal, query goal exhibe una `<tagetcondition>` implícita al requerir que todos sus parámetros de salida (out parameters) tengan un valor diferente de nulo (`null`) en caso de parametersets estos deben tener al menos un valor. Por lo tanto un query goal es exitoso automáticamente cuando todos sus parámetros contienen un valor. El agente unicamente ejecutará planes cuando la información requerida no está disponible.
- *Mantain goal* (objetivo mantener): tiene como propósito observar algún estado deseado dentro del ambiente y activamente restablecerlo cuando este es violado. Se declara dentro de la etiqueta `<maintaingoal>`. Este tipo de objetivo agrega una `<mantaincondition>` obligatoria, donde se especifica el estado a observar.

JADEX 0.96 incluye una estrategia para deliberación de objetivos llamada "Easy Deliberation", está diseñada para permitir a los desarrolladores especificar relaciones entre objetivos, utilizando "goal cardinalities" las

cuales restringen el número de objetivos de un mismo tipo que pueden estar activos al mismo tiempo y en "goal inhibitions" las cuales prohíben explícitamente perseguir otros objetivos en paralelo.

La configuración de deliberación de objetivos se incluye en la especificación de cada objetivo dentro del ADF usando la etiqueta `<deliberation>`, esta etiqueta incluye un atributo `cardinality` para manejar la cantidad de objetivos que se pueden adoptar. La etiqueta `<inhibits>` incluye un atributo `ref` para especificar qué objetivos se inhibirán si un objetivo es creado.

```
<agent  …
        name="Empresario" package="d6tionsx.empresario">
          …
    <goals>
          …

        <querygoalref name="cnp_evaluate_proposals">
            <concrete ref="procap.cnp_evaluate_proposals"/>
        </querygoalref>
          …

        <performgoal name="conocer_desarrolladores" retrydelay="1000">
            <creationcondition>$beliefbase.cantidad_desarrolladores_conocidos==0 &amp;&amp;
$beliefbase.cantidad_subproyectos!=0</creationcondition>
            <dropcondition>$beliefbase.cantidad_desarrolladores_conocidos!=0</dropcondition>
            <deliberation cardinality="1">
                <inhibits ref="subcontratar_subproyectos"  inhibit="when_active"/>
            </deliberation>
        </performgoal>

        <achievegoal name="producir_software">

        </achievegoal>

        <achievegoal name="subcontratar_subproyectos">
            <creationcondition >$beliefbase.cantidad_subproyectos!=0</creationcondition>
            <dropcondition>$beliefbase.cantidad_subproyectos==0</dropcondition>
            <deliberation cardinality="1"></deliberation>
        </achievegoal>

    </goals>
          …
</agent>
```

Figura III-XV Goals.

III.IV.I.II.V. PLANS (PLANES).

Los planes (plans) son los medios con que el agente cuenta para actuar en el ambiente, estos componen la colección de acciones que el agente puede desempeñar. Los planes son seleccionados en respuesta a eventos ya sea de mensaje, de objetivo, de condiciones, o de aplicación (Figura III-VIII).

En JADEX los planes consisten de dos partes: un encabezado de plan, o plan head, declarado en el ADF, y su correspondiente cuerpo de plan, o plan body implementado como una clase de Java; el plan head define las circunstancias en las cuales el plan body es instanciado y ejecutado. En el plan head se establece la relación con el plan body, por lo tanto los plan body pueden ser reutilizados en diferentes plan heads.

Dentro de la etiqueta `<plans>` de un ADF, se puede declarar cualquier cantidad de plan heads identificadas por la etiqueta `<plan>`. Para cada plan se deben declarar su nombre en el atributo `name` y el atributo `priority` que describe la preferencia de ejecución que tiene en comparación con otros planes.

Para cada plan se debe declarar un plan body dentro de la etiqueta `<body>`, en esta se provee una expresión Java para la creación de un nuevo plan, en la mayoría de los casos es la llamada a un constructor como: `new MyPlan()`.

La etiqueta `<trigger>` es utilizada para indicar en qué casos es aplicable la creación de una nueva instancia de un plan. Esta sub etiqueta especifica para cual mensaje o evento es utilizable el plan como respuesta. Un plan puede ser ejecutado en la presencia de un objetivo, etiqueta `<goal>`, o al momento en que un objetivo es terminado, etiqueta `<goalfinished>`, en respuesta a un evento interno, etiqueta `<internalevent>` o de mensaje, etiqueta `<messageevent>`, al cumplirse una condición, etiqueta `<condition>`, o al cambiar las creencias del agente, etiquetas: `<beliefchange>`, `<beliefsetchange>`, `<facadded>`, `<factremoved>`. Otro mecanismo de control para la selección de planes es la etiqueta `<matchexpression>`, cuando es incluida en el `<trigger>`, el plan solamente es seleccionado cuando la expresión es cierta.

De forma similar a los objetivos, los planes pueden tener parámetros, los cuales pueden almacenar valores requeridos durante la ejecución del plan.

```
<plans>
       …
        <plan name="muestra_GUI">
            <parameter name="action" class="Object">
                <goalmapping ref="rp_execute_request.action"/>
            </parameter>
            <parameter name="result" class="Object" direction="out">
                <goalmapping ref="rp_execute_request.result"/>
            </parameter>
            <body class="PlanMuestraGUI" />
            <trigger>
                <goal ref="rp_execute_request">
                    <match>$goal.getParameter("action").getValue() instanceof Boolean</match>
                </goal>
            </trigger>
        </plan>
       …
        <plan name="cnp_make_proposal_plan">
            <parameter name="cfp" class="Object">
                <goalmapping ref="cnp_make_proposal.cfp"/>
            </parameter>
            <parameter name="initiator" class="AgentIdentifier">
                <goalmapping ref="cnp_make_proposal.initiator"/>
            </parameter>
            <parameter name="proposal" class="Object" direction="out">
                <goalmapping ref="cnp_make_proposal.proposal"/>
            </parameter>
            <parameter name="proposal_info" class="Object" direction="out" optional ="true">
                <goalmapping ref="cnp_make_proposal.proposal_info"/>
            </parameter>
            <body>new PlanHacerPropuestaContrato()</body>
            <trigger>
                <goal ref="cnp_make_proposal"/>
            </trigger>
        </plan>
         …
    </plans>
```

Figura III-XVI Plans (Plans Heads).

Un plan body representa una parte de la funcionalidad del agente y encapsula un conjunto de acciones. En JADEX, los planes son escritos en Java, por lo tanto pueden acceder a cualquier biblioteca de clases Java. En implementación los planes son clases que heredan de `jadex.runtime.Plan`, que a su vez heredan de

`jadex.runtime.AbstractPlan`, para las cuales es obligatorio proveer el método `body()`, donde se ubica el código principal a ejecutar en el plan.

Dentro de un plan, se tiene acceso a la base de creencias del agente usando el método `getBeliefbase()`. La base de creencias provee los métodos `getBelief()` y `getBeliefSet()` que se utilizan para recuperar creencias por el nombre asignado en la declaración dentro del ADF. El contenido de una creencia puede ser recuperado con el método `getFact()`, para un conjunto de creencias se utiliza el método `getFacts()` que regresa un arreglo. El contenido de una creencia se modifica utilizando el método `setFact()`, el cual sobrescribe el valor previo de la creencia.

```
package d6tionsx.desarrollador;

import …;

public class PlanCierraTrato extends Plan {

    IGui gui = ((IGui) getBeliefbase().getBelief("GUI").getFact());

    @Override
    public void body() {

        SubProyecto pa = (SubProyecto) getBeliefbase().getBelief("trabajo_asignado").getFact();
        SubProyecto p = (SubProyecto) (getParameter("proposal").getValue());
        if (pa == null) {

         getBeliefbase().getBelief("sueldo_actual").setFact(p.getCostoEnSofts());

        getBeliefbase().getBelief("empresa_empleadora").setFact(p.getEmpresarioQueLoDesarrolla().getNombre
        ());
            //Notifica Cerrar el trato
        gui.addLog("Desarrollando: " + p.getNombreProyecto() + ", Para: " +
        p.getEmpresarioQueLoDesarrolla().getNombre());
            getBeliefbase().getBelief("trabajo_asignado").setFact(p);
            getParameter("result").setValue(p);
        } else {
        gui.addLog("Avandonando desarrollo: " + p.getNombreProyecto() + ", " +
        p.getEmpresarioQueLoDesarrolla().getNombre());
        }
    }

    @Override
    public void failed() {
        gui.addLogS(getAgentName() + ": " + getException().toString());
    }

    @Override
    public void passed() {
        gui.addLogS("Plan CierraTrato EJECUTADO!!!!: " + getAgentName());
    }
}
```

Figura III-XVII Implementación de planes (Plan Bodies).

Un plan se considera exitoso si el método `body()` concluye sin generar ninguna excepción. Dependiendo del resultado de la ejecución del método `body()`, si tiene éxito (termina sin ninguna excepción), falla (ocurre una excepción en la ejecución), o es abortado (cuando el objetivo que persigue el plan es alcanzado o desechado antes de que el plan body termine de ejecutarse), existen los métodos `passed()`, `failed()` y `aborted()` que sirven para controlar las acciones finales deseadas en cada caso.

III.IV.I.II.VI. CARACTERÍSTICAS GENERALES DE JADEX

Las secciones anteriores presentan el uso general de JADEX para la implementación de agentes BDI. Sin embargo, el Framework completo provee más utilidades para la definición e implementación de agentes.

Dentro de los ADFs se pueden proveer las siguientes especificaciones para ampliar la definición del agente:

- Eventos.- JADEX soporta dos tipos de eventos a nivel aplicación, los cuales pueden ser definidos en el ADF. Los eventos internos, etiqueta `<internalevent>`, son utilizados para denotar un evento ocurrido dentro del agente, los eventos de mensaje, etiqueta `<messageevent>`, representan un evento en la comunicación entre dos o más agentes.

```
<agent  …
        name="Ambiente" package="d6tionsx.ambiente">
        …
    <expressions>
        <expression name="find_empresario" class="Empresario[]">
          select Empresario $e from $beliefbase.empresarios where
          $e.getNombre().equals($nomemp)
           <parameter name="$nomemp" class="String"/>
        </expression>
        <expression name="find_desarrollador" class="Desarrollador[]">
          select Desarrollador $d from $beliefbase.desarrolladores where
          $d.getNombre().equals($nomdes)
            <parameter name="$nomdes" class="String"/>
        </expression>
        …
    </expressions>
    …
</agent>
```

Figura III-XVIII Expresiones.

```
<agent  …
        name="Ambiente" package="d6tionsx.ambiente">
        …
    <configurations>
        <configuration name="default">
            <beliefs>
                <initialbelief ref="rp_filter">
                    <fact>IFilter.ALWAYS</fact>
                </initialbelief>
            </beliefs>
            <goals>
                <initialgoal ref="df_keep_registered">
                    <parameter ref="description">
                   <value>SFipa.createAgentDescription(null,
                   SFipa.createServiceDescription("Ambiente", "Vista Simulador D6tions",
                   "d6tions"))</value>
                    </parameter>
                    <parameter ref="leasetime">
                        <value>12000</value>
                    </parameter>
                </initialgoal>
            </goals>
        </configuration>
    </configurations>
    …
</agent>
```

Figura III-XIX Configuraciones.

- Expresiones ADF.-Las expresiones son utilizadas en los ADF para evaluación dinámica, por ejemplo, consultar el estado de las creencias del agente.
- Configuraciones.- Las configuraciones representan tanto estados iniciales como estados finales de los agentes. Elementos de inicialización pueden ser declarados los cuales son creados cuando el agente es iniciado. Esto es, elementos como objetivos, o planes son creados inmediatamente cuando el agente nace (Figura III-XIX). Por otro lado, los elementos de finalización pueden ser usados para declarar objetivos o planes que se crearán cuando un agente vaya a ser terminado.

El framework también provee funcionalidad para crear o eliminar elementos (creencias, objetivos, planes) en tiempo de ejecución, así como interfaces para interactuar con otros componentes de software que no son necesariamente agentes (elementos de interacción externa). Los detalles del uso completo del API de JADEX 0.96 pueden ser consultados en (Braubach & Pokahr, Jadex User Guide. Release 0.96, 2007) y (Braubach & Pokahr, Jadex Tutorial. Release 0.96, 2007).

IV. SIMULADOR DE AMBIENTES DE DESARROLLO DE SOFTWARE.

Este tema presenta el simulador implementado para esta tesis, que es un prototipo inicial implementado con tecnología JADEX. Se tomó como base la simulación d6tions presentada en el tema II de este documento para modelar el sistema multiagente que se describe a continuación.

IV.I. SIMULADOR D6TIONSX.

El simulador d6tionsx extiende la simulación d6tions a un ambiente virtual. Cada elemento propuesto en la simulación d6tions es modelado dentro del sistema multiagente d6tionsx. La Figura IV-I agrega el estereotipo `<<agente>>` a las clases del modelo que se pueden considerar como actores dentro de este, las clases restantes son simples objetos que los actores producirán o modificaran al evolucionar la simulación.

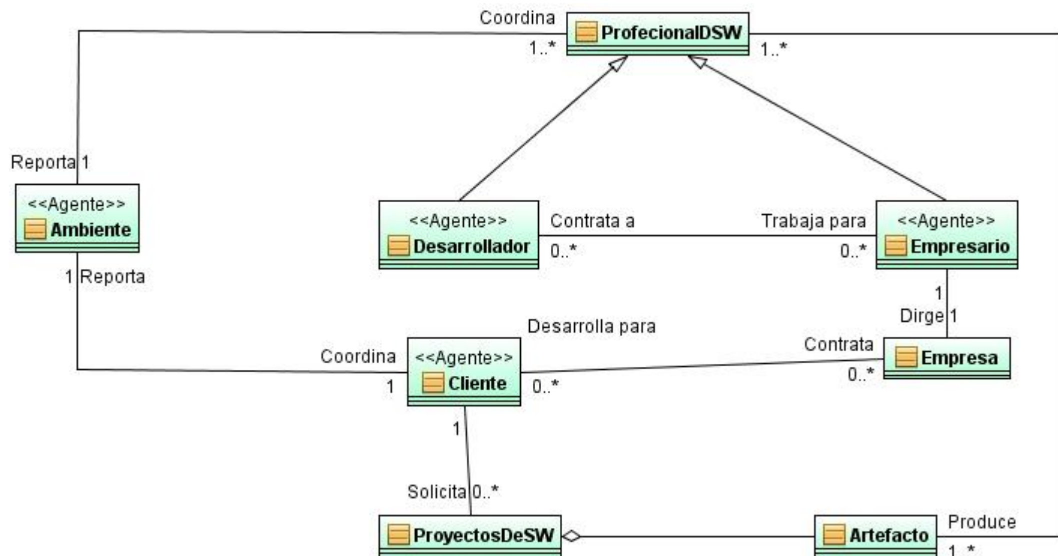


Figura IV-I Agentes implementados en el simulador d6tionsx

La simulación d6tionsx sigue la línea de actividades que muestra la Figura IV-II:

1. para comenzar los agentes que quieran participar en la simulación se reportan con un agente central (agente ambiente) quien conocerá a los participantes y los roles que desempeñan,
2. el agente central coordina el inicio de la siguiente actividad que es asignar proyectos, dándole la señal a agente cliente para que inicie negociaciones con los empresarios
3. cuando el empresario tiene un proyecto por desarrollar, negocia con los desarrolladores para que trabajen para el
4. los desarrolladores contratados comienzan a producir los artefactos de software en varias etapas para integrar el proyecto solicitado
5. el empresario integra el proyecto y se lo envía al cliente
6. el cliente evalúa el proyecto que le fue desarrollado

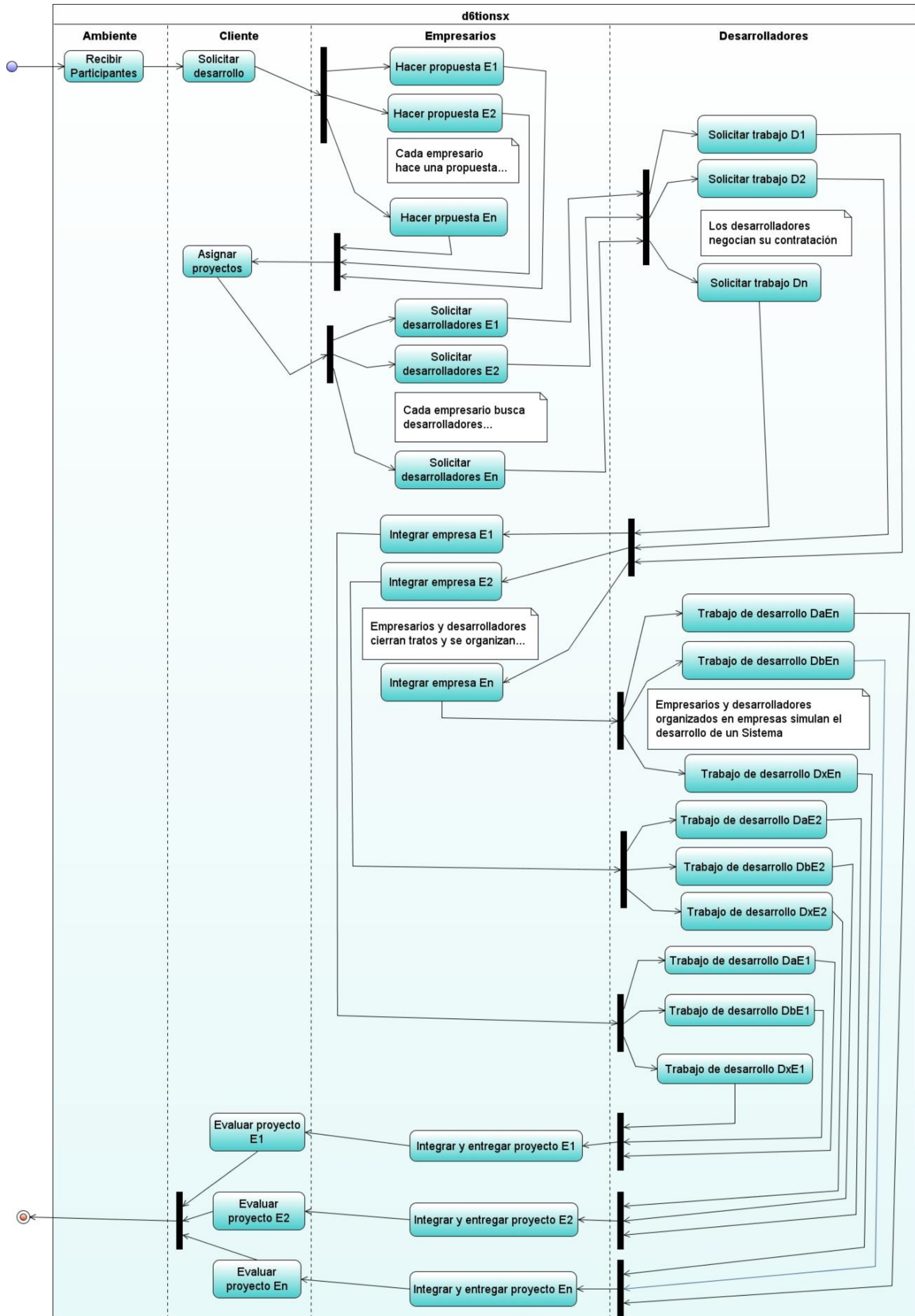


Figura IV-II Flujo general de actividades en la simulación d6tionsx.

IV.I.I. AGENTES IMPLEMENTADOS.

IV.I.I.I. AGENTE AMBIENTE.

Este agente se encarga de centralizar la información de la simulación, la cual comparte con los participantes que la requieran, conoce a los participantes y coordina el inicio de la etapa de negociación de proyectos donde los clientes buscan empresarios para que desarrollen un proyecto.

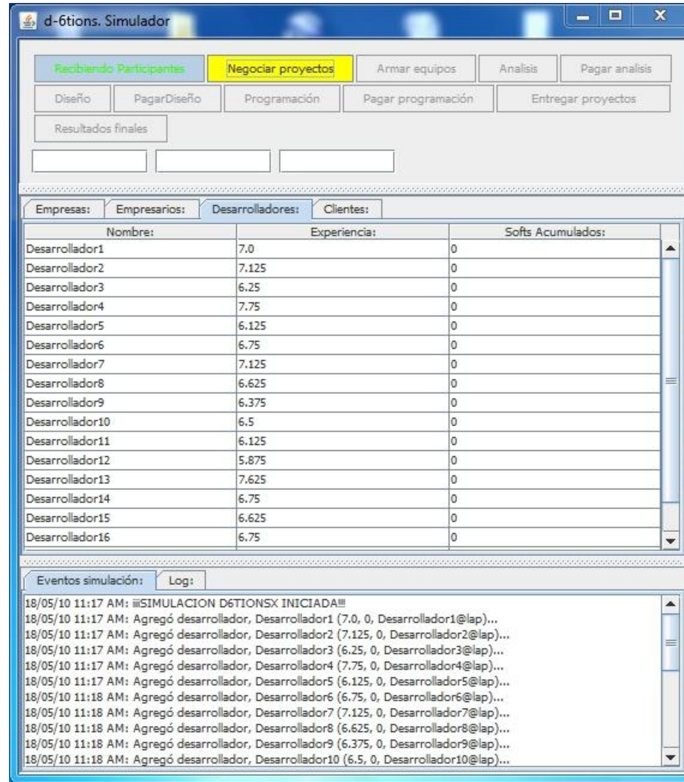


Figura IV-III Interfaz gráfica del agente ambiente.

IV.I.I.II. AGENTE CLIENTE.

Representa a un cliente en la simulación, solicita a los empresarios que le desarrollen un proyecto de software, este proyecto se lo asigna a todos los empresarios que participan en la simulación.

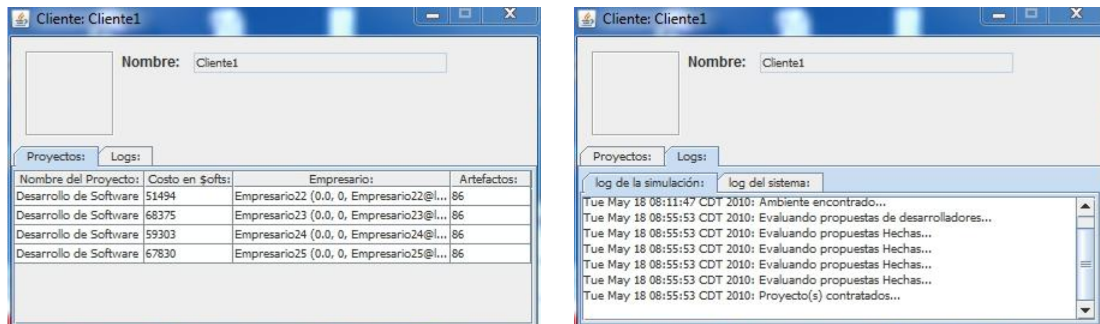


Figura IV-IV Interfaz gráfica del agente cliente

IV.I.I.III. AGENTE EMPRESARIO.

Representa a un empresario en la simulación, atiende la solicitud de desarrollo del agente cliente, una vez que el agente cliente cierra el trato para la producción de un proyecto, el empresario negocia con los desarrolladores existentes para que lo ayuden a producir el software, de no conseguir ayuda de desarrolladores él se encarga de desarrollar.

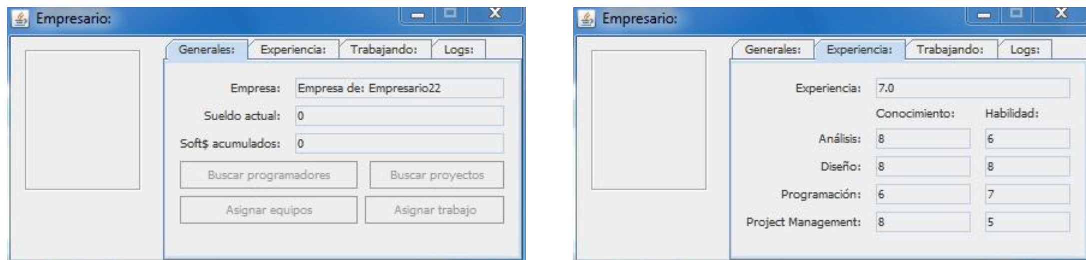


Figura IV-V Interfaz gráfica del agente empresario.

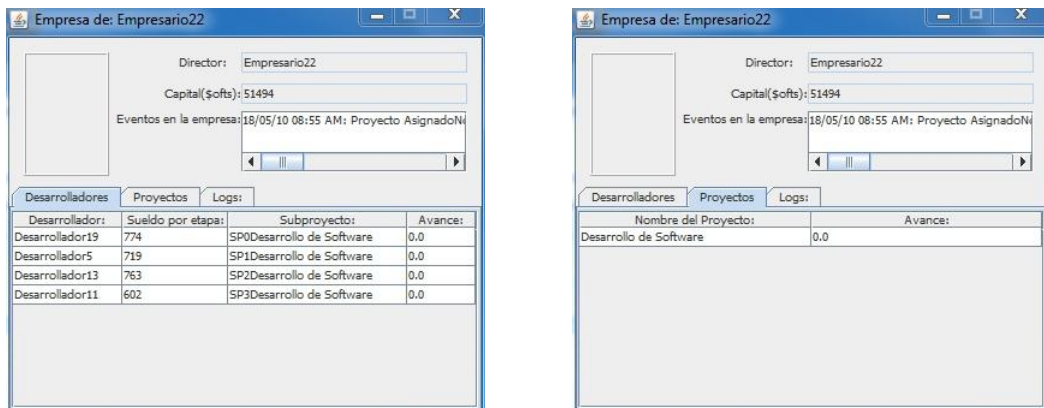


Figura IV-VI Interfaz gráfica de una empresa.

El agente empresario conoce como queda compuesta su empresa después de negociar con los desarrolladores, esto lo presenta por medio de la interfaz gráfica que muestra la Figura IV-VI

IV.I.I.IV. AGENTE DESARROLLADOR.

Representa a un desarrollador en la simulación, atiende solicitudes de los empresarios para ayudar en el desarrollo de software. Al ser contratado, simula la producción de artefactos que integran un sub proyecto que le es asignado por el empresario que lo contrata.

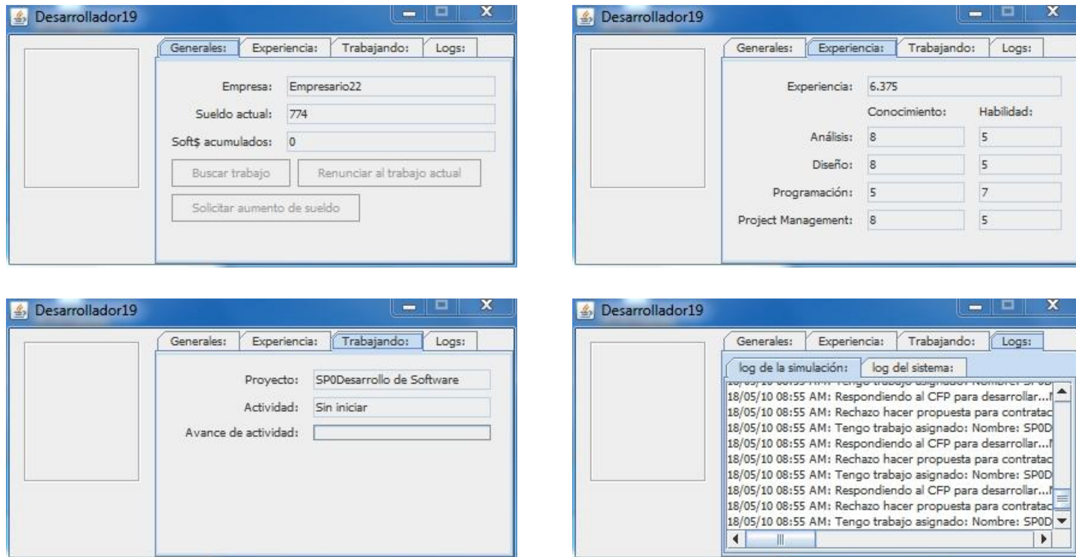


Figura IV-VII Interfaz gráfica del agente desarrollador.

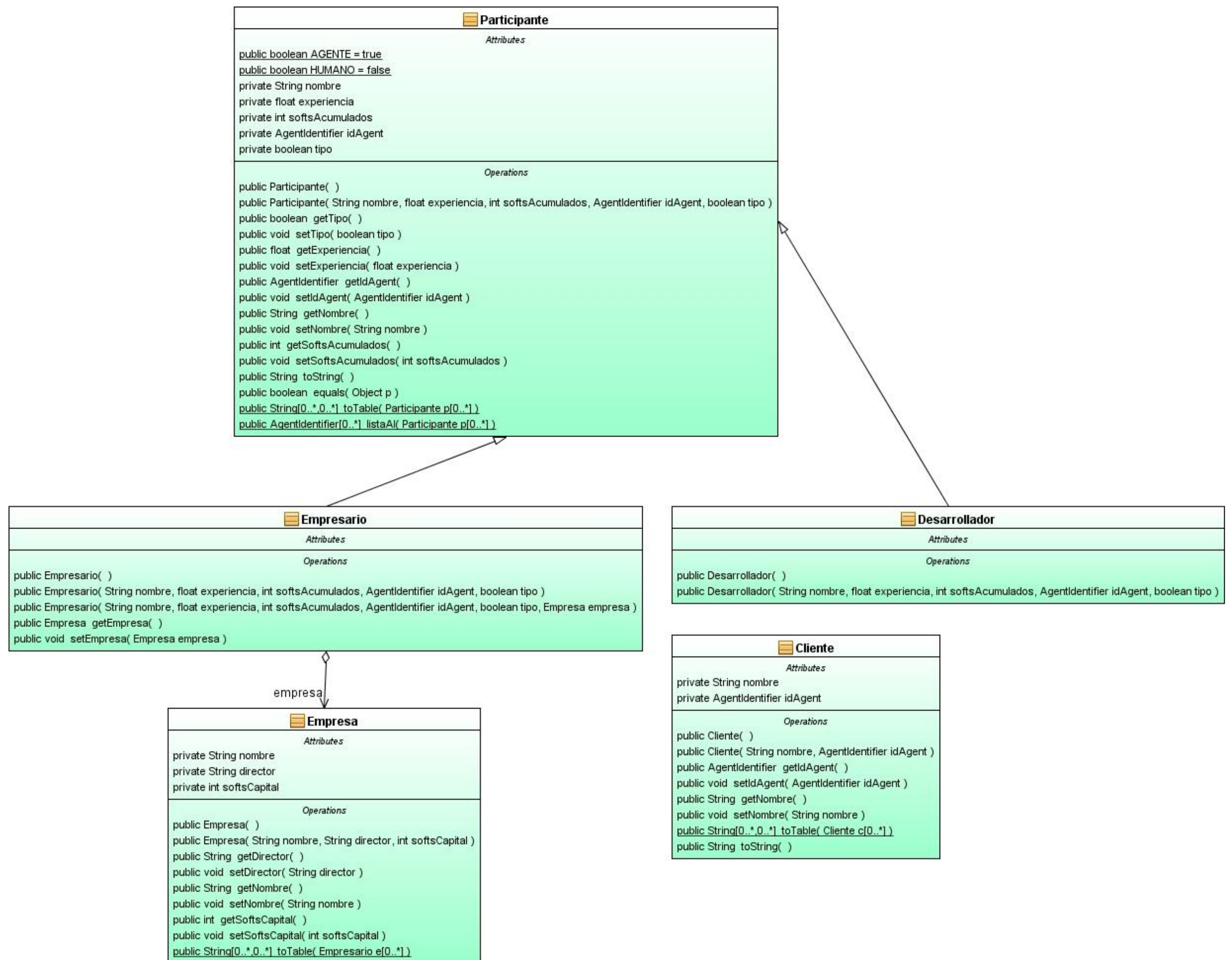


Figura IV-VIII Clases Auxiliares para manejo de información en los agentes d6tionsx (a).

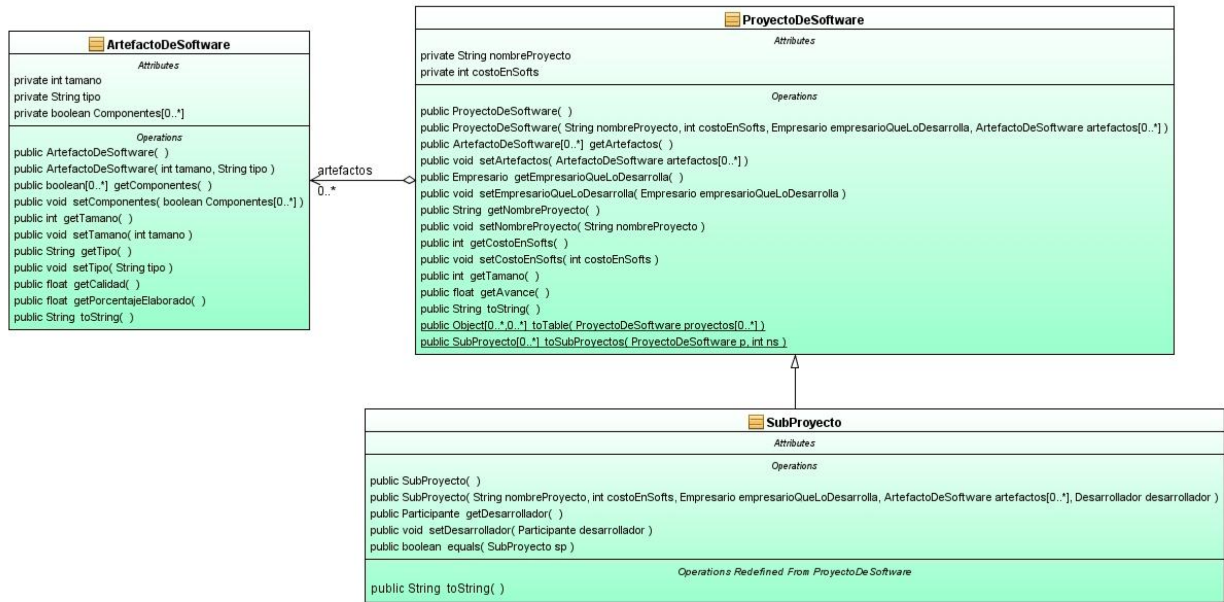


Figura IV-IX Clases Auxiliares para manejo de información en los agentes d6tionsx (b).

IV.I.I.V. CLASES AUXILIARES.

La información que manejan los agentes son objetos java. La Figura IV-VIII y Figura IV-IX muestran las clases implementadas para su uso como información en creencias y mensajes entre agentes.

IV.I.I.VI. AGENTES EMPRESARIO JUGADOR Y DESARROLLADOR JUGADOR.

Estos dos tipos de agentes son agentes intermediarios para que usuarios de la simulación participen como actores en el rol de empresario o desarrollador.

En el prototipo implementado prácticamente son los mismos que sus equivalentes no jugadores, la única diferencia es que se diseñaron de tal manera que los artefactos producidos por jugadores reflejan los resultados de la dinámica de preguntas y respuestas diseñada para evaluar a los jugadores.

Las preguntas están clasificadas como preguntas de conocimiento y preguntas de habilidad, estas incluyen preguntas de conocimientos generales del área de ingeniería de software, así como retos de habilidad matemática, lógica, de análisis etc., que representan habilidades deseadas en cualquier profesional de la ingeniería de software.

Al momento de simular la producción de un artefacto de software, se le hacen diez preguntas al jugador, cada respuesta representa una parte del artefacto, de tal manera que la calidad del artefacto producido en la simulación es la calificación obtenida por el jugador con base en el conocimiento y habilidad mostrados.

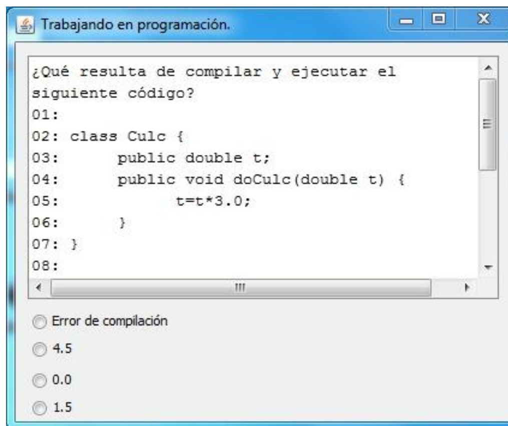


Figura IV-X Evaluación para los participantes.

IV.II. PLATAFORMA DE EJECUCIÓN DE AGENTES.

JADEX 0.96 además del API de desarrollo provee herramientas para el manejo y monitoreo de agentes en tiempo de ejecución. La plataforma de agentes JADEX es responsable de la ejecución de los agentes y provee las herramientas necesarias para la administración de estos, por ejemplo: iniciar o detener agentes y registrar o buscar servicios. Esta plataforma se incluye en el API de JADEX en la clase `jadex.adapter.standalone.Platform`.

El paquete de distribución `d6tionsx.jar` al ser ejecutado inicia la plataforma de agentes y el JADEX Control Center (JCC, Figura IV-XI) de forma automática. El JCC representa el punto de acceso principal a todas las herramientas necesarias para la ejecución y monitoreo de agentes en tiempo de ejecución.

En el JCC la primera herramienta disponible es el JADEX Starter ( ) desde donde se pueden cargar, inicar y detener agentes, consta principalmente de tres paneles:

- Model Tree Panel.- Muestra los agentes incluidos en el modelo de agentes cargado para ejecución (panel superior izquierdo Figura IV-XI).
- Runing Agents Panel.- Muestra los agentes que se encuentran en ejecución en la plataforma (panel inferior izquierdo Figura IV-XI).
- Model Panel.- Muestra los detalles del modelo del agente actualmente seleccionado el el Model Tree (panel derecho Figura IV-XI).

Para iniciar la ejecución de un agente, se selecciona su modelo en el Model Tree Panel, esto despliega los detalles del modelo del agente en el Model Panel, y se hace clic en el boton Start del Model Panel.

Cada uno de los agentes inmediatamente al iniciarse se registra en el DF de la plataforma. Sólo el registro del agente ambiente es necesario en esta versión del simulador, sin embargo todos los agentes se registran en el DF para uso futuro.

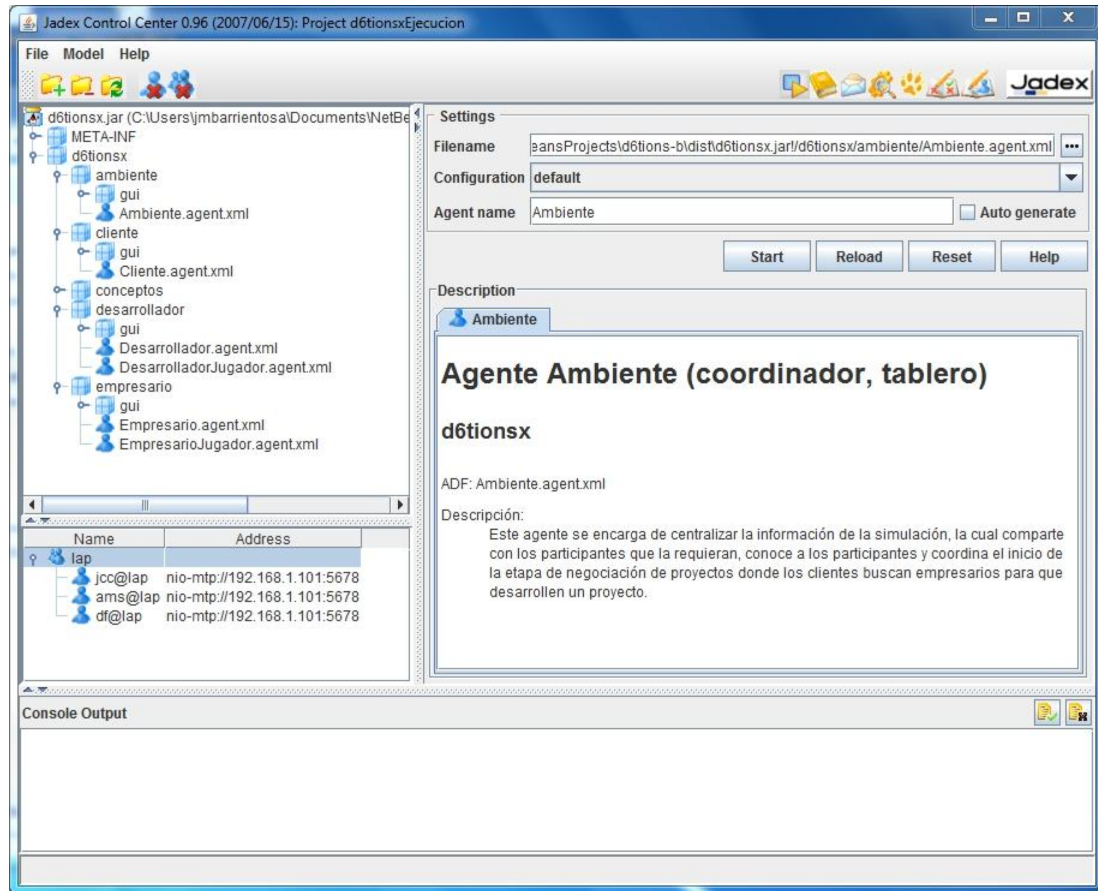
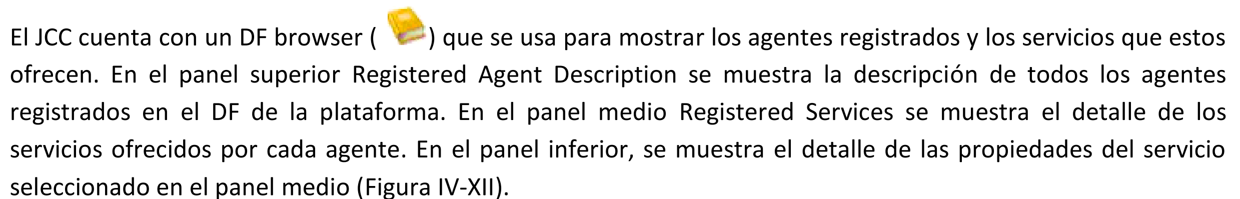


Figura IV-XI JADEX Control Center.



El JCC cuenta con un DF browser ( ) que se usa para mostrar los agentes registrados y los servicios que estos ofrecen. En el panel superior Registered Agent Description se muestra la descripción de todos los agentes registrados en el DF de la plataforma. En el panel medio Registered Services se muestra el detalle de los servicios ofrecidos por cada agente. En el panel inferior, se muestra el detalle de las propiedades del servicio seleccionado en el panel medio (Figura IV-XII).

IV.III. EJECUCIÓN DE LA SIMULACIÓN.

IV.III.I. INICIO DE AGENTES.

El agente ambiente, quien coordina la simulación, debe ser iniciado primero, al iniciarse queda en estado de espera para el registro de los siguiente participantes. Si bien se puede iniciar cualquier cantidad de agentes ambiente, unicamente el que se registre primero en el DF de la plataforma JADEX será considerado por los otros agentes participantes como coordinador de la simulación.

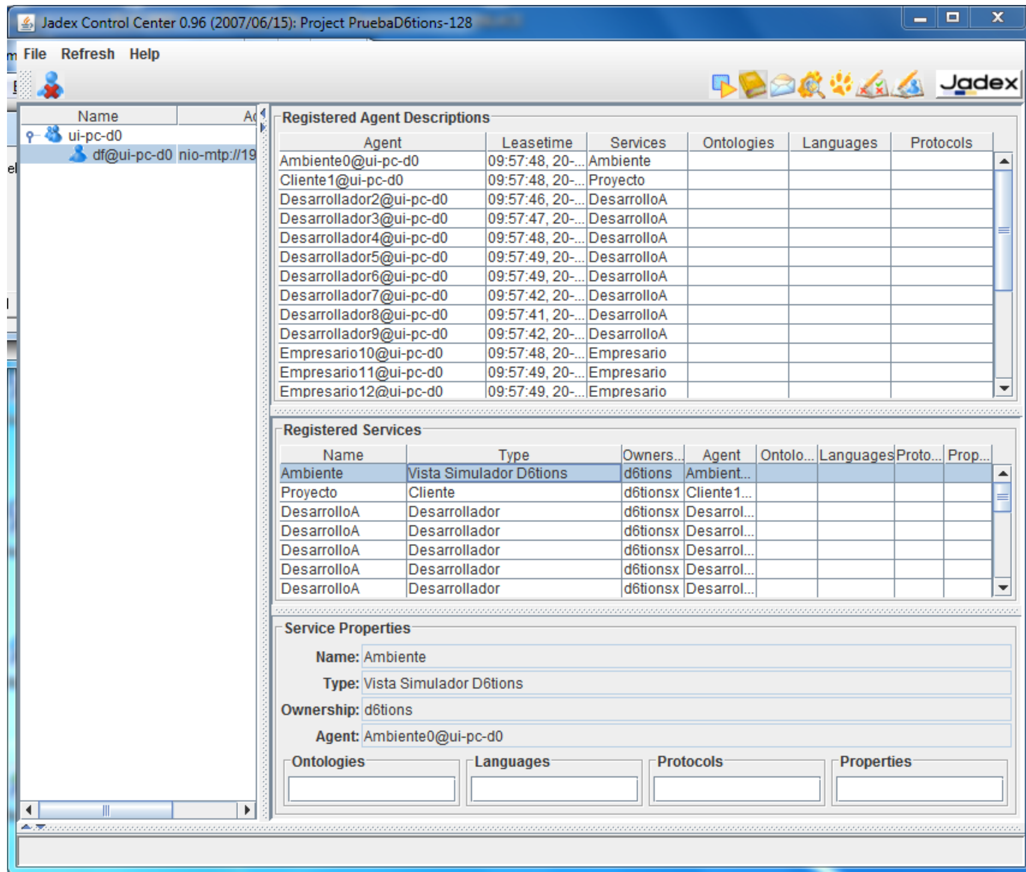


Figura IV-XII DF Browser.

No existe impedimento para iniciar cualquier tipo de agente participante antes que el agente ambiente, sin embargo, el primer objetivo que persiguen los agentes participantes es encontrar al agente ambiente en la plataforma y reportarse con él, de no encontrar al agente ambiente, los agentes participantes tendrán latente este objetivo ejecutando un plan para encontrarlo hasta que el agente ambiente se inicie (Figura IV-XIII).

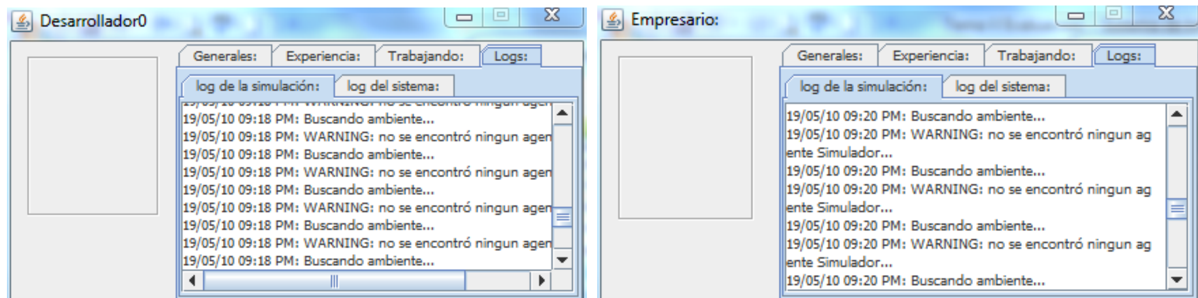


Figura IV-XIII Agentes desarrollador y empresario buscando el agente ambiente.

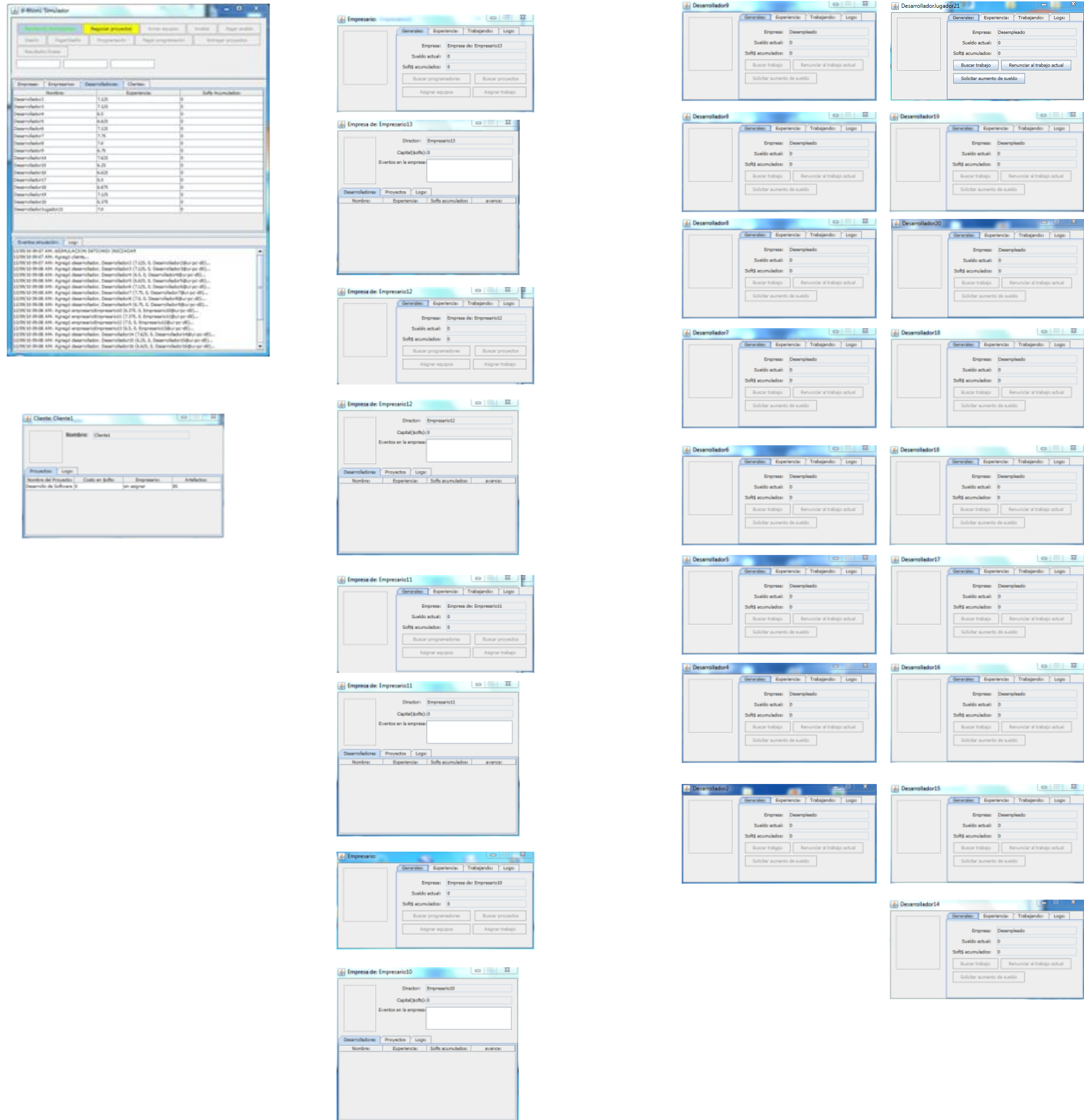


Figura IV-XIV Agentes d6tionsx iniciados.

La Figura IV-XIV muestra una simulación donde se han iniciado: un agente ambiente, un agente cliente, cuatro agentes empresarios, dieciséis agentes desarrolladores y un agente desarrollador-jugador. El cliente, los empresarios y los desarrolladores inmediatamente al iniciarse se reportan con el agente ambiente para que los conozca. Una vez que los agentes están registrados como participantes de la simulación con el agente ambiente, al no tener otro objetivo que perseguir, esperan a que la interacción para la simulación de la producción de software se inicie.

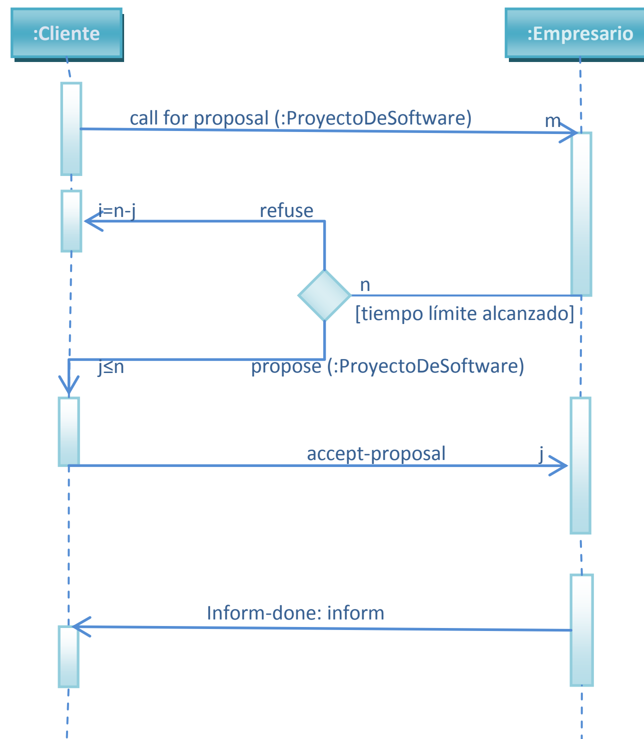


Figura IV-XV Uso del "Contract Net Protocol" para simular la contratación entre cliente y empresario.

IV.III.II. SIMULACIÓN DE LA ASIGNACIÓN DE PROYECTOS.

Para iniciar la asignación de proyectos, el agente ambiente requiere al agente cliente para que les asigne un proyecto a los agentes empresario. Esto se activa desde la interfaz del agente ambiente, al hacer clic en el botón negociar proyectos; el agente ambiente inicia un `request protocol` donde ordena al agente cliente inicie la negociación para asignar los proyectos de software.

El agente cliente atendiendo este requerimiento, inicia un `contract net protocol` para asignar a cada empresario un proyecto, en la ejecución de este protocolo los empresarios hacen su propuesta (`proposal`) donde indican el precio por desarrollar el software, el agente cliente acepta todas las propuestas hechas de manera tal que todos los agentes empresario simulan el desarrollo del mismo proyecto de software, si el agente empresario termina la interacción con un `inform`, significa que el trato para el desarrollo quedo cerrado entre el cliente y el empresario. La Figura IV-XV muestra el uso del `contract net protocol` para el desarrollo de esta interacción entre cliente y empresarios.

La Figura IV-XVI muestra a el agente cliente después de negociar con cuatro agentes empresarios la asignación de proyectos, la información del agente cliente concuerda con la información de los agentes empresario respecto al proyecto asignado y a el costo acordado en la negociación. Todos los agentes empresario simulan el desarrollo del mismo proyecto de software pera cada uno asigna su precio por desarrollar.

En este momento de la simulación, el agente cliente conoce a que empresarios les asigna proyectos, y los empresarios conocen que proyectos les fueron asignados.

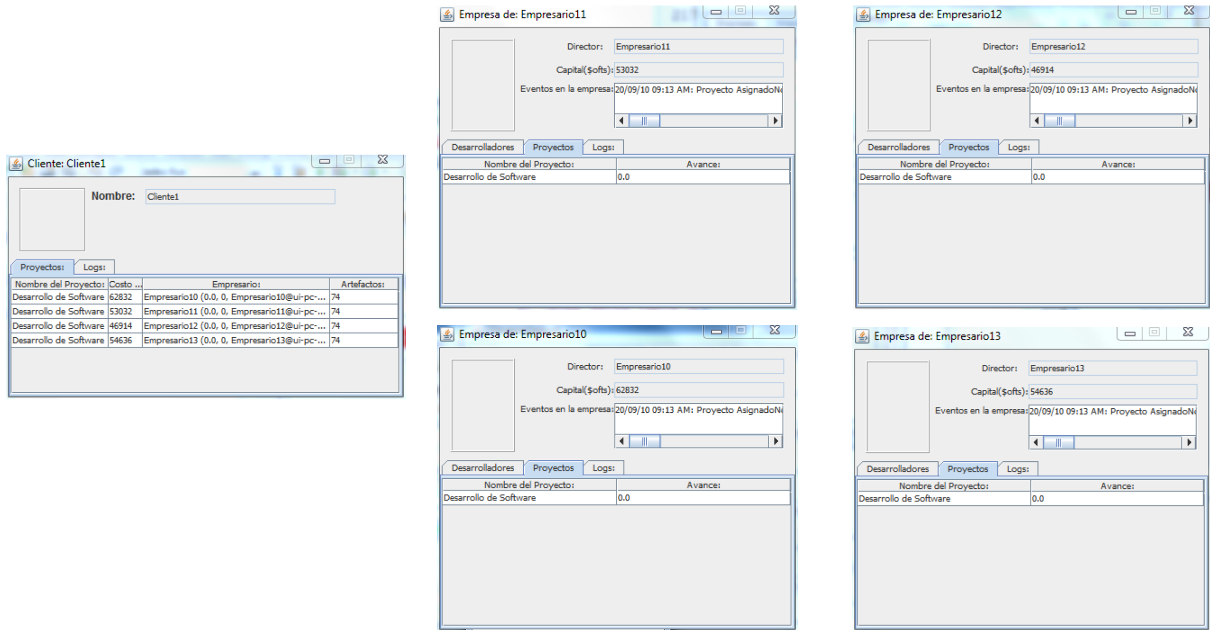


Figura IV-XVI Resultado de la negociación de proyectos en la simulación.

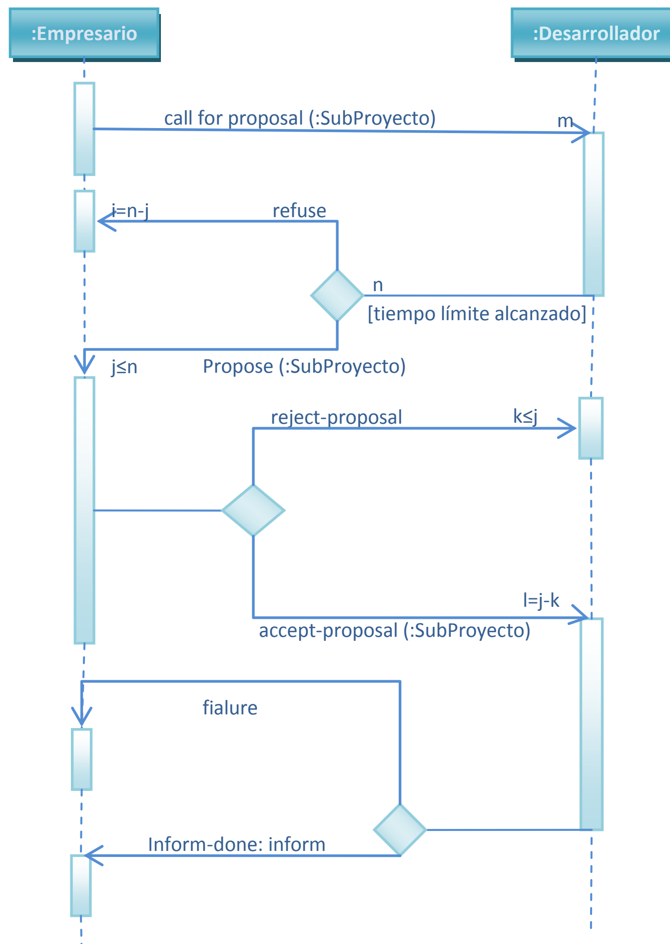


Figura IV-XVII Uso del "Contract Net Protocol" para simular la contrataciones de desarrolladores por empresarios.

IV.III.III. SIMULACIÓN DE LA CONTRATACIÓN DE DESARROLLADORES.

Cuando los agentes empresarios cierran el trato para desarrollar un proyecto, cada agente empresario activa los objetivos para contratar desarrolladores. Para esto primero requieren al agente ambiente para que les informe la lista de desarrolladores registrados en la simulación, esto se hace mediante el uso de un `request protocol` donde el resultado (`result`) del requerimiento del agente empresario al agente ambiente es dicha lista. En este momento los agentes empresario conocen a todos los agentes desarrollador.

Antes de negociar con los agentes desarrollador, los agentes empresarios dividen el desarrollo solicitado por el agente cliente en cuatro sub proyectos, estos sub proyectos son los que les asignarán a los desarrolladores que se unan a su empresa.

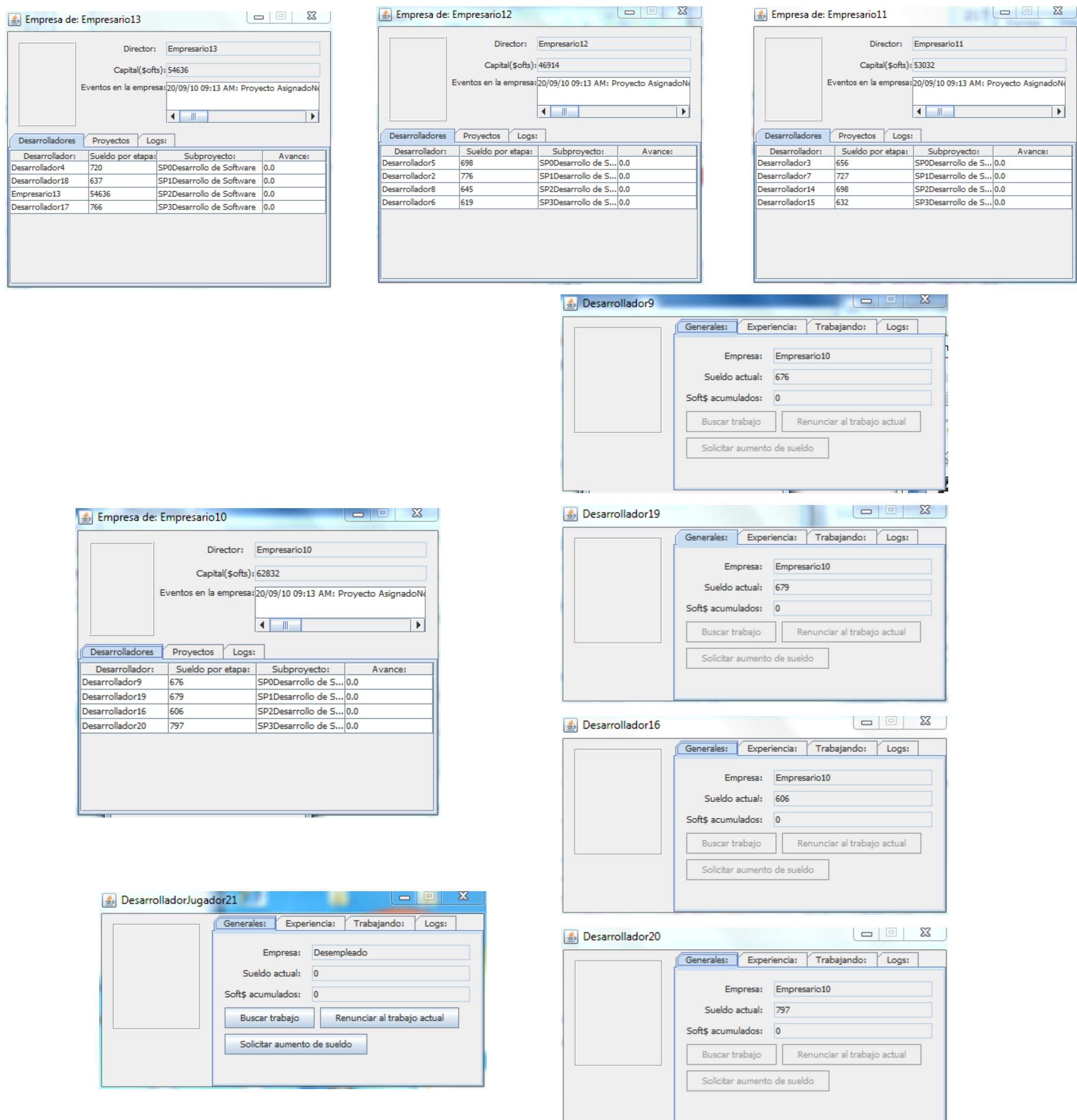


Figura IV-XVIII Resultado de la negociación para contratar desarrolladores.

Para cada sub proyecto a asignar, se inicia un `contract net protocol` en el que los agentes empresarios solicitan a los agentes desarrollador propuestas para ser contratados (Figura IV-XVII). Cada agente desarrollador puede o no hacer una propuesta, esto depende de si ya fue contratado por otro empresario o si está participando en otra negociación. El agente empresario recibe las propuestas hechas y selecciona una aleatoriamente, comunicándole al agente desarrollador que la hizo, que su propuesta fue aceptada (`accept-proposal`). Si el agente desarrollador seleccionado concluye la interacción con un `inform`, cierra el trato para desarrollar el sub proyecto con el agente empresario, esto equivale a quedar contratado en la empresa; si termina la interacción con un `fialure` el trato no es cerrado esto equivale a no ser contratado.

Si un empresario no consigue desarrollador para un sub proyecto, el mismo se encargara de desarrollarlo. En esta negociación está presente la competencia de empresarios por conseguir desarrolladores.

La Figura IV-XVIII muestra el resultado después de la simulación de la contratación en el escenario planteado en la Figura IV-XIV. En este ejemplo, la empresa del agente Empresario10 quedó formado por los agentes Desarrollador9, Desarrollador19, Desarrollador16 y Desarrollador20. Los agentes desarrollador que no consiguieron cerrar un trato quedan como desempleados, como el caso del agente DesarrolladorJugador21 en este ejemplo.

Al término de esta interacción, los empresarios conocen que desarrolladores se unieron a su empresa y los desarrolladores conocen la empresa para la cual trabajan.

IV.III.IV. SIMULACIÓN DE LA PRODUCCIÓN DE SOFTWARE.

En este punto de la simulación cada agente tiene trabajo asignado, a partir de aquí cada agente desarrollador simula la producción del sub proyecto que se le asignó.

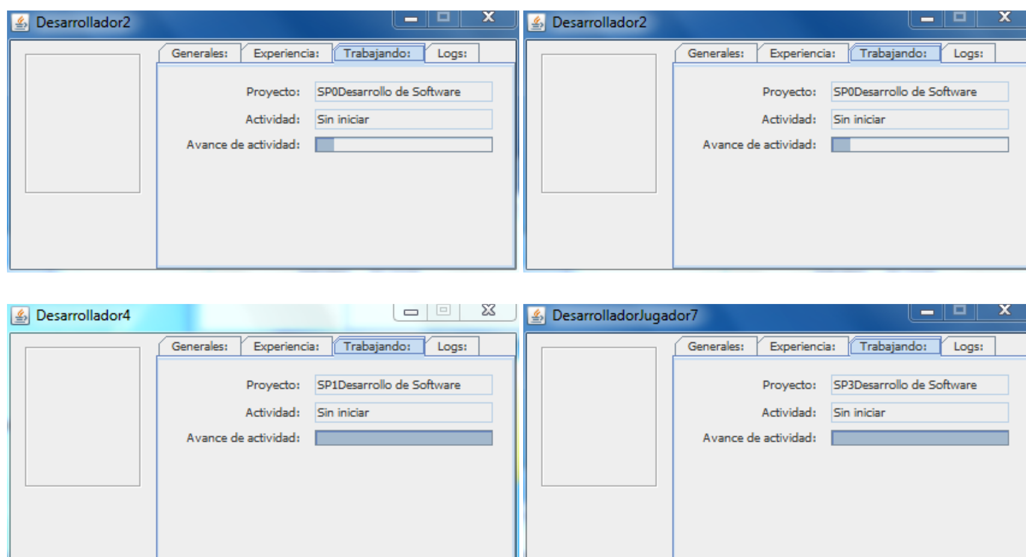


Figura IV-XIX Simulación de avance en el desarrollo.

Los agentes autónomos simulan la producción al crear los objetos de las clases auxiliares que representan el trabajo que hacen, esto sólo lo reflejan con una barra de avance en su interfaz gráfica (Figura IV-XIX).

Por otra parte los agentes de interfaz que representan participantes humanos, comienzan a mostrar preguntas relacionadas con ingeniería de software para cada artefacto a producir (Figura IV-XX), el responder estas preguntas equivale a avanzar en el proceso de desarrollo.

Al concluir la producción del sub proyecto cada participante requiere su pago que se descuenta del capital de la empresa que lo contrató. El empresario utiliza los sub proyectos para integrar el desarrollo de software que le fue asignado por el cliente.

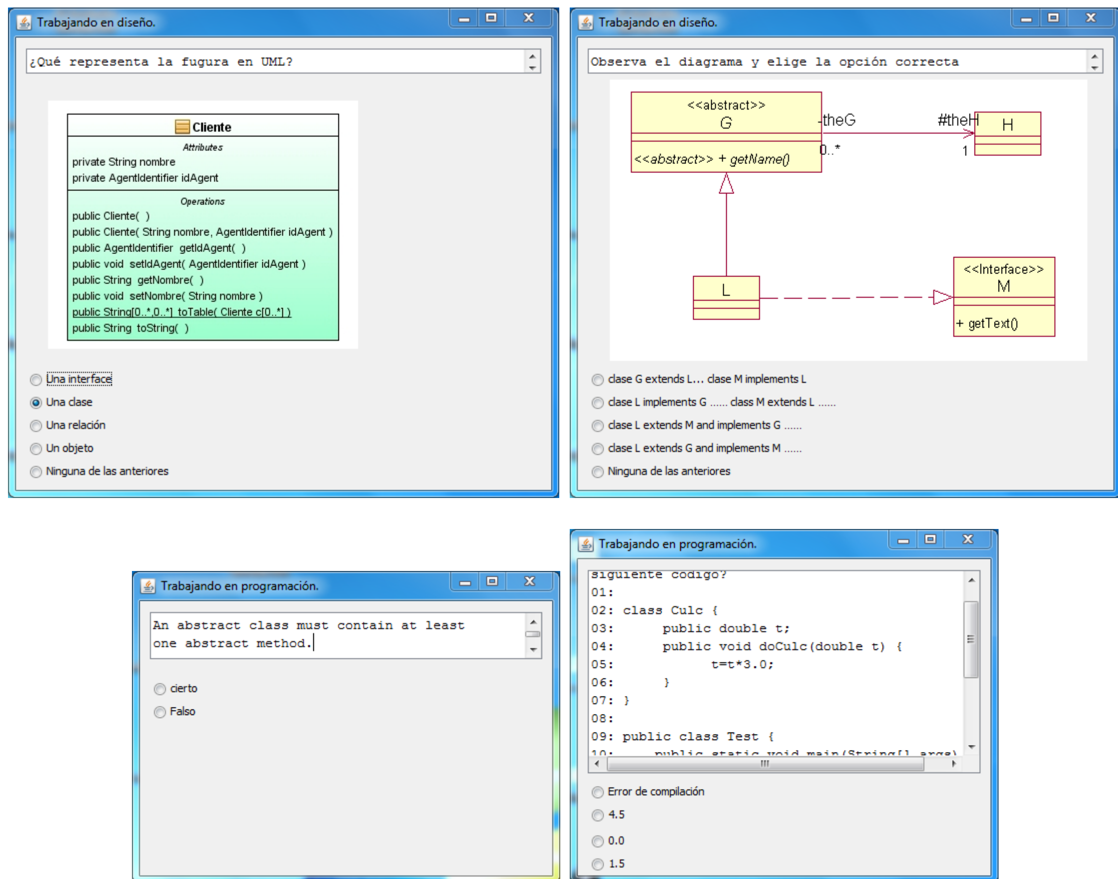


Figura IV-XX Simulación de trabajo de desarrollo para participantes jugadores.

IV.III.V. MONITOREO DEL ESTADO INTERNO DE LOS AGENTES.

Para el monitoreo individual de cada agente JCC incluye la herramienta Introspector ( ) con la cual se puede observar y manipular el estado mental de los agentes. El estado mental de los agentes incluye las creencias, los objetivos adoptados y los planes ejecutándose en un momento dado.

La Figura IV-XXI muestra el monitoreo de las creencias de un agente desarrollador en una simulación, con esta herramienta se pueden observar los valores que el agente almacena en su base de creencias, la Figura IV-XXII muestra el monitoreo de los objetivos que el agente ha adoptado, finalmente la Figura IV-XXIII muestra el monitoreo de los planes que el agente ha ejecutado.

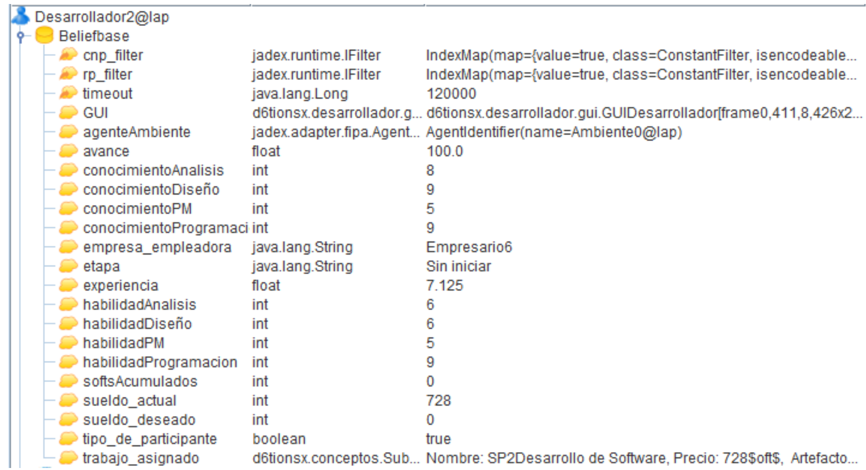


Figura IV-XXI Monitoreo de Creencias con la herramienta Introspector del JCC.

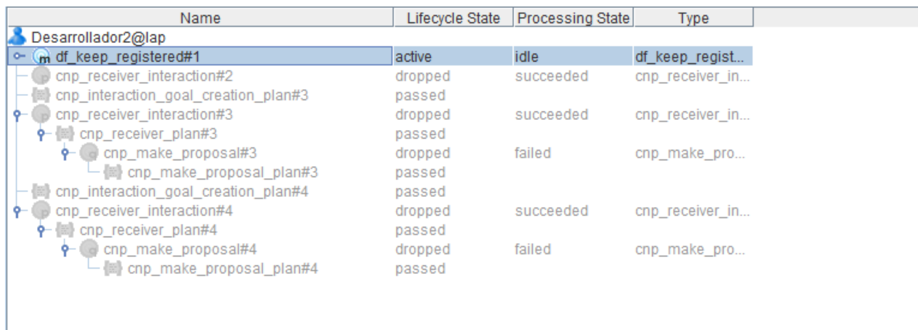


Figura IV-XXII Monitoreo de Objetivos con la herramienta Introspector del JCC.

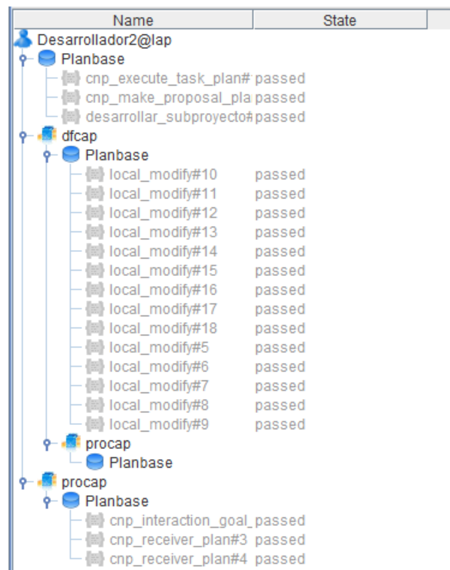
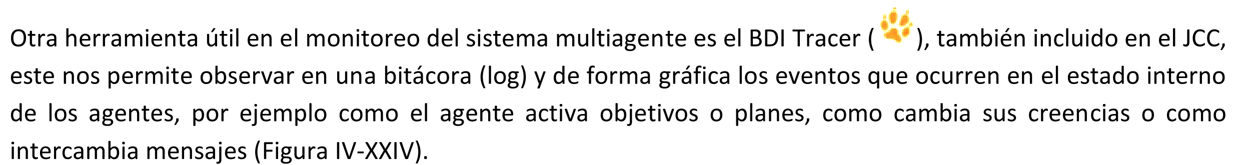


Figura IV-XXIII Monitoreo de Planes con la herramienta Introspector del JCC.

IV.III.VI. MONITOREO DE EVENTOS EN EL SISTEMA.



Otra herramienta útil en el monitoreo del sistema multiagente es el BDI Tracer ( ), también incluido en el JCC, este nos permite observar en una bitácora (log) y de forma gráfica los eventos que ocurren en el estado interno de los agentes, por ejemplo como el agente activa objetivos o planes, como cambia sus creencias o como intercambia mensajes (Figura IV-XXIV).

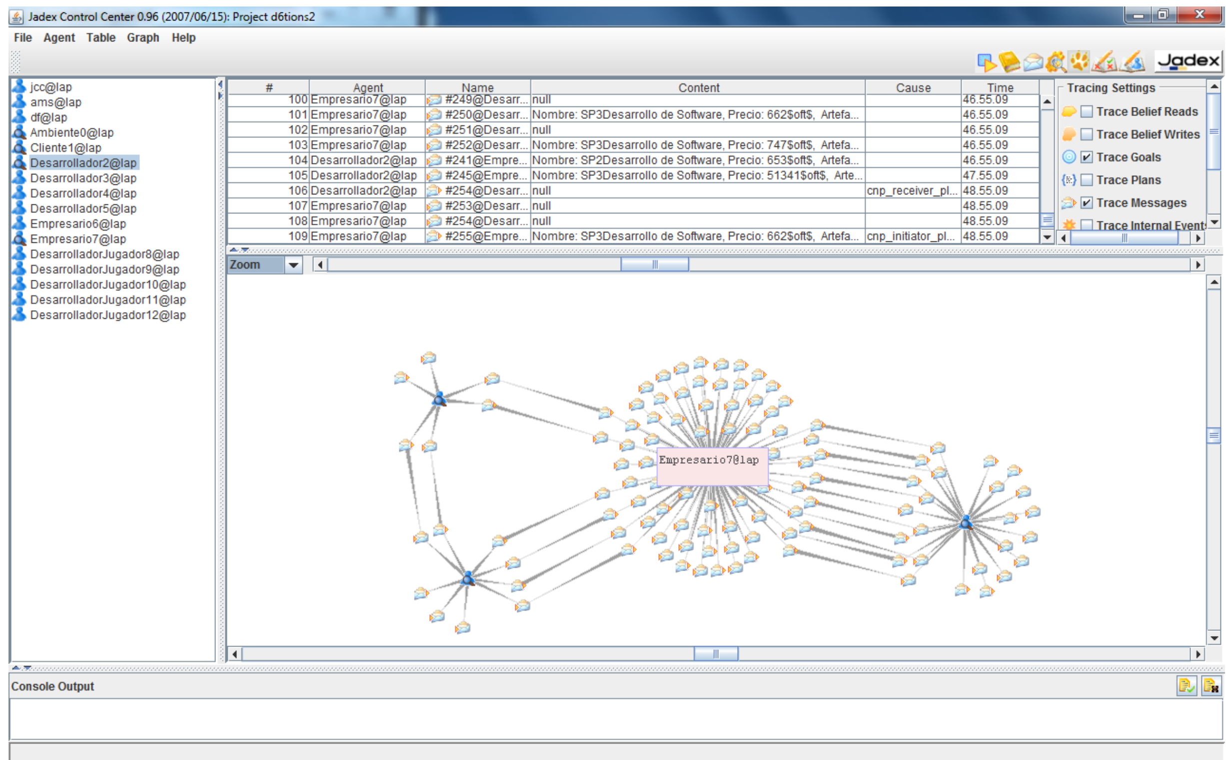


Figura IV-XXIV BDI Tracer.

El rastro de eventos se puede visualizar en forma de bitácora en la Trace Table, que muestra información como el identificador (id) del evento, el nombre del agente, el nombre del evento, el contenido y el tiempo en que ocurre, la Figura IV-XXV muestra un rastreo de eventos de la ejecución de un agente desarrollador.

Otra opción para visualizar esta información es el Trace Exploration Graph, que muestra en forma gráfica estos eventos, la Figura IV-XXVI muestra el rastreo gráfico de eventos de la ejecución de un agente desarrollador. Con el Trace Exploration Graph también se puede observar la interacción entre agentes, La Figura IV-XXVII muestra el gráfico del intercambio de mensajes entre varios agentes al ejecutar una simulación.

#	Agent	Name	Content	Cause	Time
19	DesarrolladorJugador6@...	cnp_make_proposal#2	RQueryGoal(name=cnp_make_proposal#...	cnp_receiver_plan#2	29.01.10
20	DesarrolladorJugador6@...	cnp_make_proposal_plan#2	RPlan(name=cnp_make_proposal_plan#2)	cnp_make_propos...	29.01.10
21	DesarrolladorJugador6@...	GUI	17	cnp_make_propos...	29.01.10
22	DesarrolladorJugador6@...	trabajo_asignado	null	cnp_make_propos...	29.01.10
23	DesarrolladorJugador6@...	GUI	17	cnp_make_propos...	29.01.10
24	DesarrolladorJugador6@...	trabajo_asignado	null	cnp_make_propos...	29.01.10
25	DesarrolladorJugador6@...	#58@DesarrolladorJugador6	Nombre: SP0Desarrollo de Software, Prec...	cnp_receiver_plan#1	29.01.10
26	DesarrolladorJugador6@...	#59@DesarrolladorJugador6	Nombre: SP0Desarrollo de Software, Prec...	cnp_receiver_plan#2	29.01.10
27	DesarrolladorJugador6@...	local_modify#7	RPlan(name=local_modify#7)	df_keep_registered...	29.01.10
28	DesarrolladorJugador6@...	#61@Empresario10	Nombre: SP0Desarrollo de Software, Prec...		29.01.10
29	DesarrolladorJugador6@...	cnp_execute_task#1	RAchieveGoal(name=cnp_execute_task#1)	cnp_receiver_plan#1	29.01.10
30	DesarrolladorJugador6@...	cnp_execute_task_plan#1	RPlan(name=cnp_execute_task_plan#1)	cnp_execute_task#1	29.01.10
31	DesarrolladorJugador6@...	GUI	17	cnp_execute_task_...	29.01.10
32	DesarrolladorJugador6@...	trabajo_asignado	null	cnp_execute_task_...	29.01.10
33	DesarrolladorJugador6@...	trabajo_asignado	Nombre: SP0Desarrollo de Software, Prec...	cnp_execute_task_...	29.01.10
34	DesarrolladorJugador6@...	desarrollar_trabajo_asignado#1	RPerformGoal(name=desarrollar_trabajo...		29.01.10
35	DesarrolladorJugador6@...	desarrollar_subproyecto#1	RPlan(name=desarrollar_subproyecto#1)	desarrollar_trabajo...	29.01.10
36	DesarrolladorJugador6@...	GUI	17	desarrollar_subpro...	29.01.10
37	DesarrolladorJugador6@...	trabajo_asignado	Nombre: SP0Desarrollo de Software, Prec...	desarrollar_subpro...	29.01.10
38	DesarrolladorJugador6@...	#70@DesarrolladorJugador6	Nombre: SP0Desarrollo de Software, Prec...	cnp_receiver_plan#1	30.01.10
39	DesarrolladorJugador6@...	cnp_filter	jadex.runtime.ConstantFilter@3b1f8a		30.01.10
40	DesarrolladorJugador6@...	cnp_filter	jadex.runtime.ConstantFilter@f0f29b		30.01.10
41	DesarrolladorJugador6@...	cnp_interaction_goal_creation_p...	RPlan(name=cnp_interaction_goal_creati...	#75@EmpresarioJ...	30.01.10
42	DesarrolladorJugador6@...	cnp_filter	jadex.runtime.ConstantFilter@3b1f8a		30.01.10
43	DesarrolladorJugador6@...	cnp_filter	jadex.runtime.ConstantFilter@f0f29b		30.01.10
44	DesarrolladorJugador6@...	cnp_interaction_goal_creation_p...	RPlan(name=cnp_interaction_goal_creati...	#79@Empresario1...	30.01.10
45	DesarrolladorJugador6@...	#66@EmpresarioJugador11	Nombre: SP0Desarrollo de Software, Prec...		30.01.10
46	DesarrolladorJugador6@...	#75@EmpresarioJugador11	Nombre: SP1Desarrollo de Software, Prec...		30.01.10
47	DesarrolladorJugador6@...	cnp_receiver_interaction#3	RPerformGoal(name=cnp_receiver_intera...		30.01.10
48	DesarrolladorJugador6@...	cnp_receiver_plan#3	RPlan(name=cnp_receiver_plan#3)	cnp_receiver_intera...	30.01.10
49	DesarrolladorJugador6@...	#79@Empresario10	Nombre: SP1Desarrollo de Software, Prec...		30.01.10
50	DesarrolladorJugador6@...	cnp_receiver_interaction#4	RPerformGoal(name=cnp_receiver_intera...		30.01.10
51	DesarrolladorJugador6@...	cnp_receiver_plan#4	RPlan(name=cnp_receiver_plan#4)	cnp_receiver_intera...	30.01.10

Figura IV-XXV Rastreo de eventos en la "Trace Table".

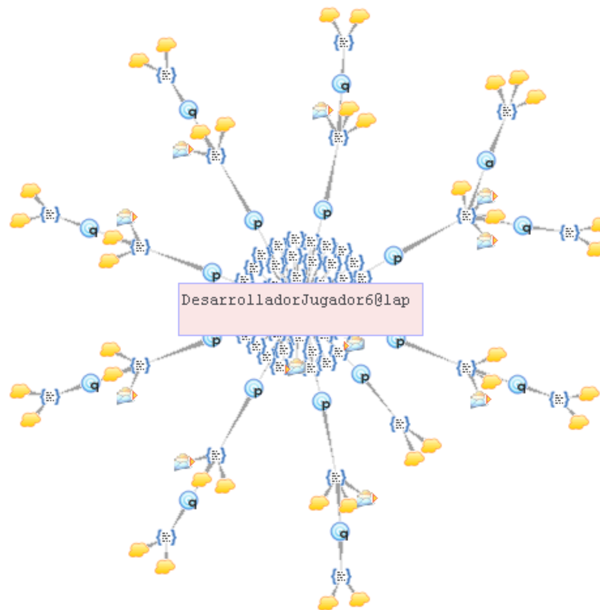


Figura IV-XXVI Rastreo gráfico de eventos de un agente.

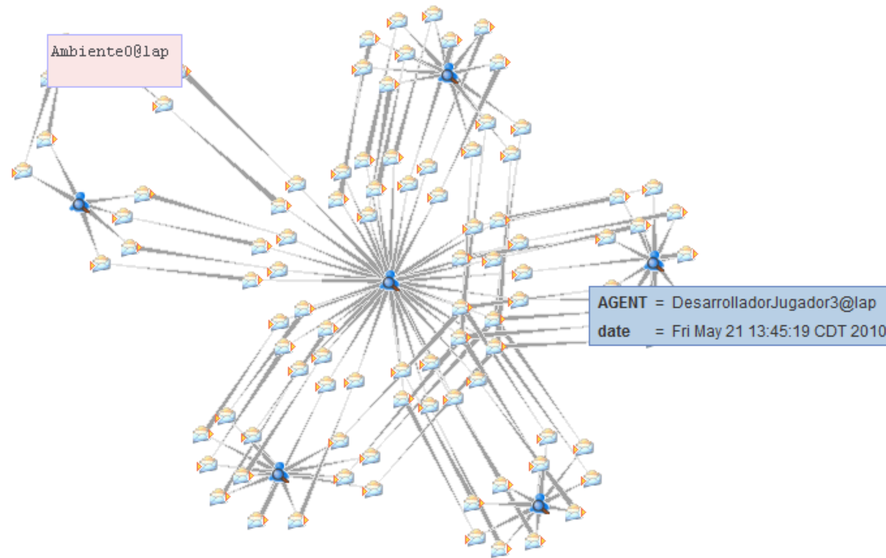


Figura IV-XXVII Rastreo de mensajes entre agentes.

IV.IV. PRUEBAS.

El prototipo fue ejecutado en las cinco configuraciones de equipo que muestra la Tabla IV-I, y para cada configuración se ejecutaron las cinco pruebas base que muestra la Tabla IV-II.

No. De Equipo	Procesador	Memoria RAM	Sistema operativo	Resolución del monitor
1	Intel Pentium 4 3.2 GHz	1Gb	Windows XP	1280x1024
2	Intel Pentium 4 3.2 GHz	3Gb	Windows XP	1280x1024
3	Intel Pentium 4 3.2 GHz	3Gb	Ubuntu Linux 9.10	1280x1024
4	Intel Core 2 Duo P8700 2.53 GHz	2Gb	Windows 7 Professional	1280x800
5	Intel Atom N270 1.60 GHz	2Gb	Windows 7 Starter	1024x700

Tabla IV-I Hardware de prueba.

No. Prueba	No. de Agentes Ambiente	No. de Agentes Cliente	No. de Agentes Empresario	No. de Agentes Desarrollador
1	1	1	1	4
2	1	1	2	8
3	1	1	4	16
4	1	1	8	42
5	1	1	16	64

Tabla IV-II Configuraciones de prueba.

Dentro de estos escenarios de prueba el prototipo se comporta de manera estable en el desarrollo de la simulación base descrita en el tema anterior.

En la configuración número dos y en la configuración número cuatro, se ejecutaron pruebas con mayor cantidad de agentes (treinta y dos agentes empresarios y ciento veintiocho agentes desarrollador), en estas condiciones

de prueba se observaron fallas de ejecución debido a la cantidad de tareas que tiene que administrar la plataforma, específicamente los agentes dejan de responder y el Java Runtime Environment (JRE) notifica errores por falta de memoria.

Otro inconveniente encontrado al momento de realizar las pruebas es el espacio disponible en las pantallas, cuando se están monitoreando los agentes las resoluciones utilizadas resultan reducidas y para quienes no están familiarizados con la simulación pudieran parecer visualmente confusas. Este tipo de inconveniente se aminora usando monitores y resoluciones más grandes, la Figura IV-XXVIII muestra una ejecución en la configuración cuatro a una resolución de 1920x1080, también la característica de múltiples escritorios que ofrece Gnome en Ubuntu 9.10 resulta útil para distribuir las GUIs de los agentes (Figura IV-XXIX, Figura IV-XXX).

La ejecución de la simulación mostró un problema de actualización de las GUI´s de los agentes: aun cuando el estado mental del agente tenía toda la información correcta de acuerdo al funcionamiento programado, en ocasiones no actualizaba su GUI, este problema se debe a problemas de JADEX al momento de notificar a tareas que dependen del API swing de Java (Braubach & Pokahr, Jadex User Guide. Release 0.96, 2007).

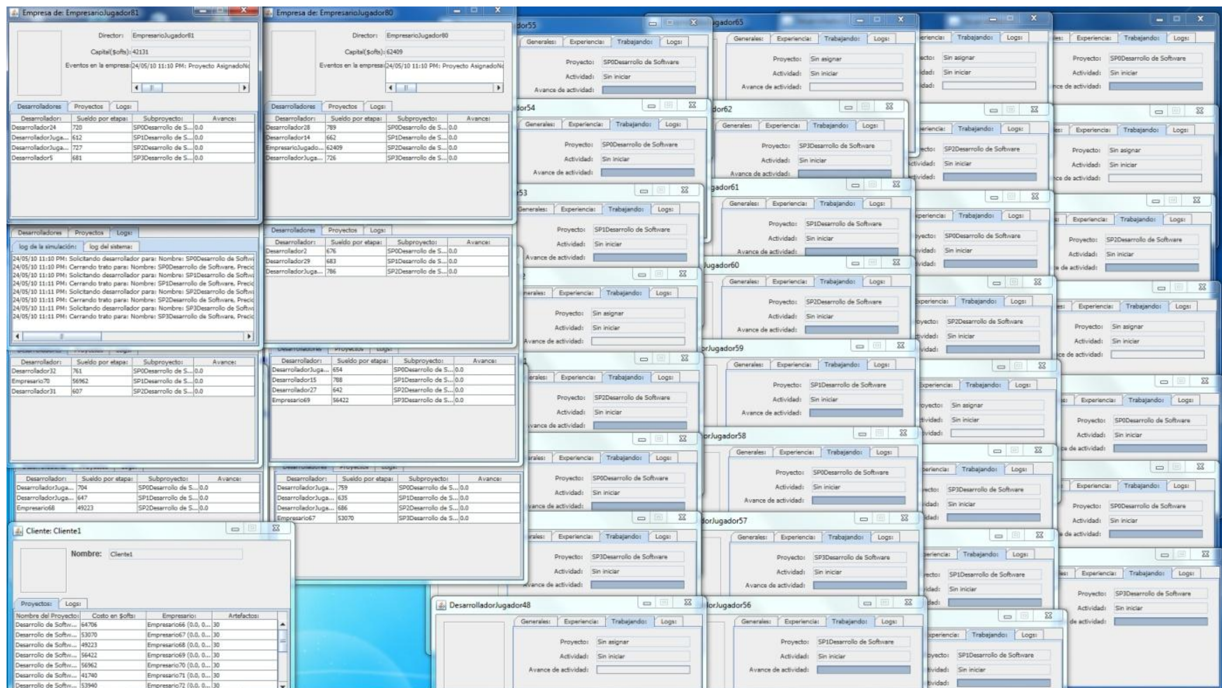


Figura IV-XXVIII Pruebas de ejecución en Windows 7 Profesional.

Los detalles de uso de las herramientas para la administración y monitoreo de agentes JADEX pueden consultarse en (Braubach & Pokahr, Jadex Tool Guide. Release 0.96, 2007). El paquete de distribución `d6tions.jar`, el código fuente y la documentación del prototipo d6tions se pueden obtener del CD que acompaña este documento.

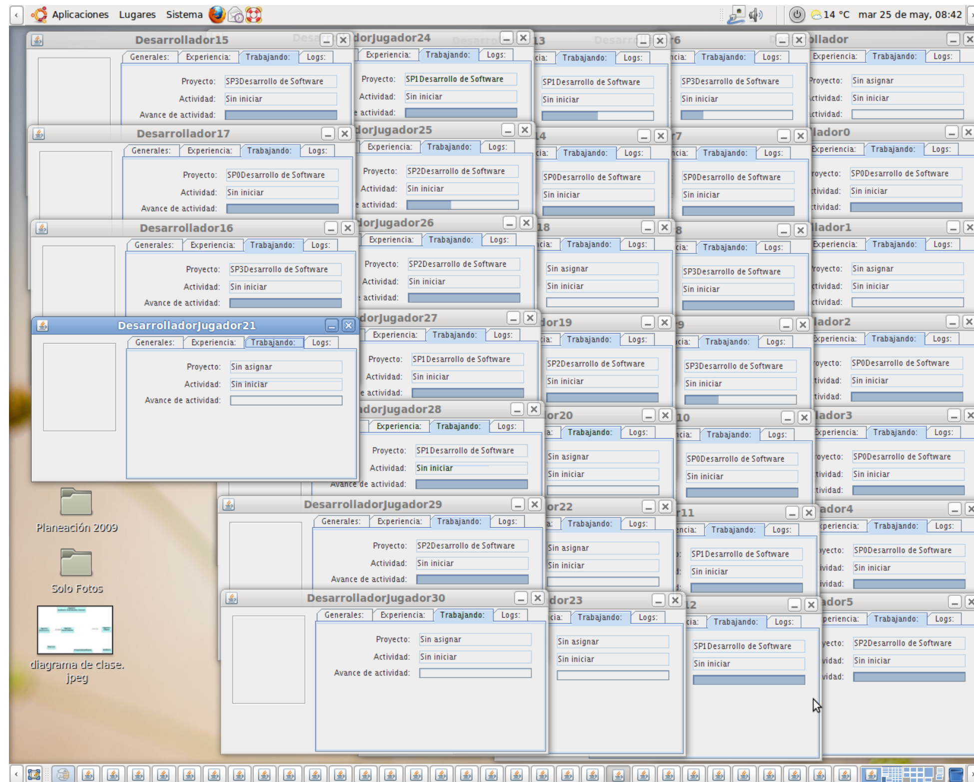


Figura IV-XXIX Prueba de Ejecución en Ubuntu 9.10 (a).

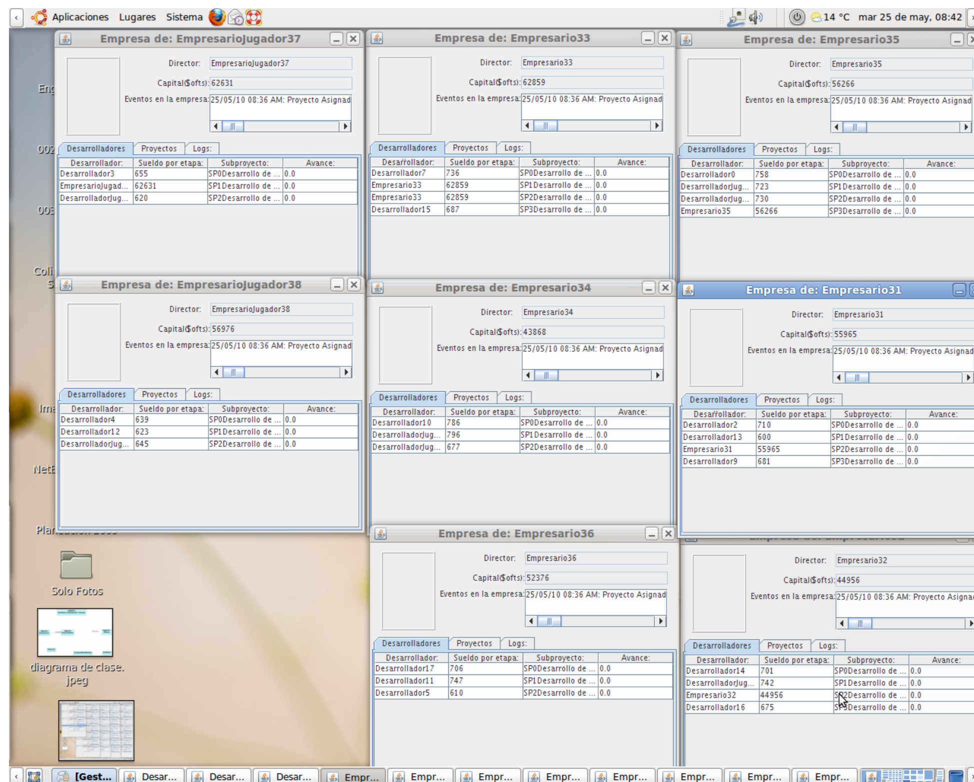


Figura IV-XXX Prueba de Ejecución en Ubuntu 9.10 (b).

V. EVALUACION.

Actualmente el simulador d6tionsx solamente se ha implementado a nivel prototipo, su evaluación se basa en un comparativo con las otras aproximaciones para el uso de simuladores en la enseñanza de la ingeniería de software que se presentaron en el tema II.

Simulación	Modo	Jugadores	Roles	Presentación	Simulación de:	Ambiente simulado:
SESAM	Por computadora	1	Project Manager	Texto	Proceso	1 empresa, varios desarrolladores.
Problems & Programers	Presencial	2	Project Manager	Juego de cartas	Proceso	2 empresas, varios desarrolladores.
SimSE	Por computadora	1	Project Manager	Gráfica (animación)	Proceso	1 empresa, varios desarrolladores.
SimjavaSP	Por computadora	1	Project Manager	Web	Proceso	1 empresa, varios desarrolladores.
Juegos de Rol	Presencial	Multijugador	Varios	Dinámica de grupo	Escenarios específicos	Escenarios específicos
d6tions	Presencial	Multijugador	Empresario, Desarrollador	Juego de equipos	Proceso y uso de conocimientos en ISW.	Varias empresas, varios desarrolladores.
d6tionsx	Por computadora	Multijugador	Empresario, Desarrollador	Gráfica	Proceso y uso de conocimientos en ISW.	Varias empresas, varios desarrolladores.

Tabla V-I Otras simulaciones en la enseñanza de la ingeniería de software.

Todas las aproximaciones concuerdan en: que la ingeniería de software es un tema complejo de enseñar, que la enseñanza de la ingeniería de software debe de ir más allá de las lecturas y la formación tradicional en clase, que la simulación es una herramienta aplicable y de valor en la enseñanza de la ingeniería de software; y se centran en la enseñanza del proceso.

La simulación d6tionsx agrega lo siguiente a las propuestas analizadas:

- El uso explicito de tecnologías multiagente, que nos permite modelar la simulación desde su elemento básico más importante: el individuo. El individuo modelado representa a un actor en la simulación y de su comportamiento individual y en interacción con otros actores, se puede obtener el macro modelo que representa el complejo ambiente necesario para producir software, que queremos mostrarle y enseñarle a los alumnos. La tecnología orientada a agentes, presenta ventajas para la implementación de este tipo de simulaciones. El hecho de poder iniciar por el modelado de entidades individuales, a las que podemos programarles comportamiento y funciones de socialización, nos permite partir desde el individuo, la interacción basada en protocolos estandarizados (la socialización), hasta llegar al modelado de organizaciones o sociedades. Cada uno de estos elementos puede aplicar a la simulación de ambientes de producción de software, los individuos: profesionales, clientes, desarrolladores, etc.; su interacción: negociación, comunicación, etc.; y sus organizaciones: empresas, equipos.
- La opción de que sean de uno a muchos alumnos participando en la simulación (ambiente multijugador o multiparticipante). Esto permite a varios participantes compartir la experiencia de observar el mismo ambiente pero desde el rol y las condiciones que se van generando para cada uno. Esto es juntar distintos puntos de vista sobre lo que sucede en una misma simulación.

- La posibilidad de que el participante experimente más de un rol en la simulación. Quienes quieran hacer carrera en la industria actual de software, deberán adoptar diversos roles (programadores, documentadores, administradores de proyectos), entonces parte importante de la formación es dar la opción al participante de desempeñar y aprender de estos distintos roles.
- El reto del conocimiento y la habilidad (preguntas que simulan el trabajo de producción de software) que el participante ha adquirido previamente en áreas especificas de la ingeniería de software (análisis, diseño, programación, pruebas, documentación, administración, etcétera). Es importante hacer ver al participante que si bien siempre es importante seguir adquiriendo nuevos conocimientos, en un ambiente real hay que demostrar y aplicar los conocimientos y habilidades aprendidas en la formación.

Al implementarse como un sistema multiagente, d6tionsx nos da la posibilidad de crear tantas de estas entidades como requiramos, esto permite que el número de actores dentro del modelo sea una variable al experimentar con nuestra simulación (0..* Clientes, 0..* Empresarios), permitiendo la creación de escenarios variables.

Desde un macro nivel d6tionsx nos permite observar todo el ambiente, los desarrolladores, los empresarios, el cliente, como se organizan. Además en un micro nivel nos permite observar cada uno de estos elementos en lo individual, con esto las características que poseen se pueden tratar como variables a observar dentro de la simulación. Por ejemplo, los desarrolladores exhiben su habilidad y conocimiento en distintas áreas (análisis, programación, diseño) como una calificación que va variando de acuerdo a su actuar dentro de la simulación.

D6tionsx permite la inclusión del alumno como un actor mas dentro de la simulación, es decir d6tionsx permite a los participantes de la simulación tomar el rol de Empresario o Desarrollador según su elección, esto mediante un agente interfaz que lo representa y le permite interactuar con otros agentes o con los objetos que se producen. Con esto el participante en sí, se vuelve otra entidad variable a observar en la simulación.

VI. RESULTADOS, CONCLUSIONES Y TRABAJO FUTURO.

El simulador d6tionsx descrito en esta tesis, presenta el modelado inicial de una simulación de ambientes de desarrollo de software basada en sistemas multiagente. Al quedar en una versión prototipo, marca lineamientos iniciales y abre las posibilidades para que en versiones posteriores se incluyan más características que se deseen exponer en la simulación.

Un sistema multiagente resulta una herramienta aplicable para la implementación de un simulador de ambientes de desarrollo de software por que permite asociar los elementos que lo componen, agentes, a una abstracción de los elementos del modelo a representar: desarrolladores, empresarios, clientes. Específicamente el uso de agentes BDI, es aprovechado para modelar a los actores, personas en esta simulación, ya que estos agentes representan un modelo filosófico del razonamiento y actuar humano.

Los agentes d6tionsx representan a profesionistas que tienen distintos niveles de conocimiento en materia de ingeniería de software interrelacionándose para producir un producto de software. La simulación d6tionsx aprovecha protocolos estandarizados en tecnología multiagente para habilitar la comunicación entre actores, y para modelar y representar dinámicas como la asignación de proyectos o la contratación de desarrolladores, que son ejercicios de negociación y organización que ocurren en cualquier ambiente de desarrollo de software y que se desea que los alumnos conozcan. D6tionsx implementa un ejercicio de preguntas y respuestas para que los alumnos, a la par de observar la simulación, participen demostrando los conocimientos y habilidades que han adquirido referentes a las distintas áreas de la ingeniería de software, con esto modela y representa la idea de que los alumnos deben usar sus conocimientos en el uso de tecnologías y la aplicación de procesos para producir software, ya que la ingeniería de software no es sólo programación, no es sólo diseño, no es sólo seguir un proceso, sino la integración de estas partes en un todo.

Con los agentes actores, usando sus conocimientos que representan su capacidad para usar tecnologías y seguir procesos, y la interacción que se da entre cliente, empresarios y desarrolladores, observable desde el sistema multiagente en operación y a través de las herramientas de monitoreo JADEX, d6tions provee un ambiente simulado de ingeniería de software.

D6tionsx persigue ser una herramienta auxiliar en los cursos de ingeniería de software. La visión de d6tionsx, es darles la posibilidad a los alumnos de experimentar, de forma virtual, con los elementos que componen un ambiente de desarrollo de software. En este ambiente están las personas que aplican tecnologías y siguen procesos para producir un producto de software compuesto de varios artefactos.

Para lograr esa visión queda como trabajo futuro la implementación y uso de una versión completa en cursos de ingeniería de software para la consiguiente evaluación de su efectividad. La implementación completa debe incluir el modelado formal de más conceptos de ingeniería de software, como modelos de procesos específicos, uso de tecnologías en particular, u otros roles de los actores.

En cuanto a la mejora de implementación, la interfaz gráfica actual puede considerarse abstracta al solamente presentar ventanas con información de los agentes; otra oportunidad de trabajo futuro es cambiar esta interfaz a agentes animados que hagan más entendible el simulador para personas que apenas comiencen a relacionarse con la ingeniería de software.

Ya en operación de una versión completa, por lo que se refiere a la evaluación de conocimientos, se podría aprovechar la oportunidad de utilizar la parte de preguntas y respuestas en el simulador para obtener realimentación de los conocimientos que tienen los alumnos, ya sea en temas específicos o conforme avanzan

en su formación. Así como la oportunidad de mejorar la parte de preguntas y respuestas para incluir herramientas formales de evaluación de conocimiento.

Se dice que actualmente los ingenieros de software unicamente se forman con la experiencia que da el tiempo, otro tema que queda por analizar es la utilidad de estos simuladores para generar esa formación basada en experiencia, en lapsos de tiempo más cortos.

BIBLIOGRAFÍA

ACM/IEEE-CS Task Force on Computing Curricula. (2004). *Software Engineering 2004 Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering.* Retrieved Agosto 16, 2010, from Software Engineering 2004: http://sites.computer.org/ccse/SE2004Volume.pdf

Advanced Development Methods Inc. (2010). *Scrum*. Retrieved Agosto 2010, 16, from Control Chaos: http://www.controlchaos.com/

Agile Alliance. (2010). *Agile Alliance*. Retrieved Agosto 16, 2010, from www.agilealliance.org

Ambler, S. W. (2005). *The Elements of UML 2.0 Style.* Cambridge University Press.

Association for Computing Machinery. (2009). *Software Engineering Code of Ethics and Professional Practice*. Retrieved Octubre 15, 2009, from www.acm.org: www.acm.org/about/about/se-code

Baker, A. (2004). *Problems and Programmers*. Retrieved Agosto 2010, 16, from Problems and Programmers: http://www.problemsandprogrammers.com

Bauer, F. L. (1972). Software engineering. *Information Processing* , 530.

Boetje, J. (2006). Foundational actions: teaching software engineering when time is tight. *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education* (pp. 285 - 288). Bologna, Italy: ACM.

Bratman, M. (1987). *Intention, Plans, and Practical Reason.* Harvard University Press,Cambridge.

Braubach, L., & Pokahr, A. (2007). *Jadex Tool Guide. Release 0.96.* Retrieved Agosto 16, 2010, from Online Documentation: Jadex BDI Agent System: http://jadex-agents.informatik.uni-hamburg.de/docs/jadex-0.96x/toolguide/toolguide.pdf

Braubach, L., & Pokahr, A. (2007). *Jadex Tutorial. Release 0.96.* Retrieved Agosto 16, 2010, from Online Documentation: Jadex BDI Agent System: http://jadex-agents.informatik.uni-hamburg.de/docs/jadex-0.96x/tutorial/tutorial.pdf

Braubach, L., & Pokahr, A. (2007). *Jadex User Guide. Release 0.96.* Retrieved Agosto 16, 2010, from Online Documentation: Jadex BDI Agent System: http://jadex-agents.informatik.uni-hamburg.de/docs/jadex-0.96x/userguide/userguide.pdf

Case, A. F. (1985). Computer-aided software engineering (CASE): technology for improving software development productivity. *ACM SIGMIS Database* , 35 - 43.

Chalamish, M., Sarne, D., & Kraus, S. (2007). Mass programmed agents for simulating human strategies in large scale systems. *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems.* Honolulu, Hawaii: ACM.

Claypool, K., & Claypool, M. (2005). Teaching software engineering through game design. *ACM SIGCSE Bulletin* , 123 - 127.

Cockburn, A. (2007). Teaching the next generation software engineering. *Journal of Computing Sciences in Colleges* , 118 - 118.

Drappa, A., & Ludewig, J. (2000). Simulation in software engineering training. *Proceedings of the 22nd international conference on Software engineering* (pp. 199 - 208). Limerick, Ireland: ACM.

DSDM Consortium. (2010). *DSDM Atern - The Agile Project Delivery Framework*. Retrieved Agosto 16, 2010, from dsdm.org: http://www.dsdm.org

Finkelstein, A., & Dowell, J. (1996). A Comedy of Errors: the London Ambulance Service case study. *Proc. 8th International Workshop on Software Specification & Design* (pp. 2-4). IEEE CS Press.

Gleick, J. (1996). *A bug and a Crassh.* Retrieved Agosto 16, 2010, from James Gleick: http://www.around.com/ariane.html

GSwE2009. (2010). *Graduate Software Engineering 2009*. Retrieved Agosto 20, 2010, from Graduate Software Engineering 2009: http://www.gswe2009.org/

Hazzan, O., & Tomayko, J. (2005). Teaching human aspects of software. *Proceedings of the 27th international conference on Software engineering* (pp. 647 - 648). St. Louis, MO, USA: ACM.

Heldman, K. (2003). *Project Management JumpStart.* Sybex.

Henry, T. R., & LaFrance, J. (2006). Integrating role-play into software engineering courses. *Journal of Computing Sciences in Colleges* , 32 - 38.

Hood, D. J., & Hood, C. S. (2006). Teaching software project management using simulations. *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education* (pp. 289 - 293). Bologna, Italy: ACM.

IBM. (2010). *IBM Rational Unified Process (RUP)*. Retrieved Agosto 2010, 16, from ibm.com: http://www-01.ibm.com/software/awdtools/rup/

IEEE computer society. (2010). *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. Retrieved Agosto 16, 2010, from IEEE computer society: http://www.computer.org/portal/web/swebok

IEEE Foundation for Intelligent Physical Agents. (2010). *Foundation for Intelligent Physical Agents*. Retrieved Agosto 16, 2010, from www.fipa.org

IEEE. (1990). *IEEE Standard Glossary of Software Engineering Terminology.* New York, NY: IEEE.

ISO/IEC. (2001). *ISO/IEC 9126-1:2001 Software engineering -- Product quality -- Part 1: Quality model.* ISO.

Jaccheri, L., & Morasca, S. (2006). On the importance of dialogue with industry about software engineering education. *Proceedings of the 2006 international workshop on Summit on software engineering education* (pp. 5 - 8). Shanghai, China: ACM.

Jalote, P. (2005). *An Integrated Approach to Software Engineering, Third Edition.* Springer.

Khan, S., Makkena, R., McGeary, F., Decker, K., Gillis, W., & Schmidt, C. (2003). A multi-agent system for the quantitative simulation of biological networks. *Proceedings of the second international joint conference on Autonomous agents and multiagent systems* (pp. 385 - 392). Melbourne, Australia: ACM.

LeJeune, N. F. (2006). Teaching software engineering practices with Extreme Programming. *Journal of Computing Sciences in Colleges* , 107 - 117.

Leveson, N. (1995). *Medical devices: The Therac-25.* Retrieved Agosto 16, 2010, from Therac-25 Accidents: An Updated Version of the Original Accident Investigation Paper (by Nancy Leveson).

Liu, J., Marsaglia, J., & Olson, D. (2002). Teaching software engineering to make students ready for the real world. *Journal of Computing Sciences in Colleges* , 43 - 50.

McCarthy, J. (1979). Ascribing mental qualities to machine. In M. Ringle, *Philosophical Perspectives in Artificial Intelligence.* Humanities Press.

Microsoft. (2010). *Microsoft*. Retrieved Agosto 16, 2010, from Microsoft: http://www.microsoft.com

Navarro, E. (2010). *SimSE Online*. Retrieved Agosto 16, 2010, from SimSE An Educational Game Based Software Engineering Simulation Environment: http://www.ics.uci.edu/~emilyo/SimSE/

Norvig, P., & Russell, S. (2003). *Artificial Intelligence A Modern Approach. SECOND EDITION.* Prentice Hall.

Object Management Group, Inc. (2010). *UML® Resource Page*. Retrieved Agosto 20, 2010, from Unified Modeling Language: www.uml.org

Odell, J., Van Dyke Parunak, H., & Bauer, B. (2001). Representing agent interaction protocols in UML. *First international workshop, AOSE 2000 on Agent-oriented software engineering* (pp. 121 - 140). Limerick, Ireland: Springer-Verlag.

Oracle Corporation. (2010). *NetBeans*. Retrieved Agosto 16, 2010, from NetBeans.org: http://www.netbeans.org/

Pender, T. (2003). *UML Bible.* John Wiley & Sons.

Pieterse, V., Kourie, D. G., & Sonnekus, I. P. (2006). Software engineering team diversity and performance. *Proceedings of the 2006 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries* (pp. 180 - 186). Somerset West, South Africa: South African Institute for Computer Scientists and Information Technologists.

Pokahr, A. (2010). *JADEX Software Projects Overview*. Retrieved Agosto 20, 2010, from JADEX Software Projects: http://jadex.informatik.uni-hamburg.de

Pressman, R. (2009). *Software Engineering: A Practitioner's Approach. 7 edition.* McGraw-Hill.

Real Academia Española. (2010). *DICCIONARIO DE LA LENGUA ESPAÑOLA - Vigésima segunda edición*. Retrieved Agosto 2010

Rusu, A., Rusu, A., Docimo, R., Confesor, S., & Paglione, M. (2009). Academia-academia-industry collaborations on software engineering projects using local-remote teams. *Proceedings of the 40th ACM technical symposium on Computer science educatio* (pp. 301-305). Chattanooga, TN, USA: ACM.

Searle, J. (1969). *Speech Acts.* Cambridge University Press.

Shannon, R. E. (1998). Introduction to the art and science of simulation. *Proceedings of the 30th conference on Winter simulation* (pp. 7 - 14). Washington, D.C., United States: IEEE Computer Society Press.

Shaw, K., & Dermoudy, J. (2005). Engendering an empathy for software engineering. *Proceedings of the 7th Australasian conference on Computing education - Volume 42* (pp. 135 - 144). Newcastle, New South Wales, Australia: Australian Computer Society, Inc.

Software Engineering Institute. (2010). *Capability Maturity Model Integration (CMMI)*. Retrieved Agosto 16, 2010, from Software Engineering Institute: http://www.sei.cmu.edu/cmmi/index.cfm

Software Engineering Institute. (1990). *SEI Report on Undergraduate Software Engineering Education.* Pittsburgh, Pennsylvania: Software Engineering Institute, Carnegie Mellon University.

Software Engineering Institute. (2010). *Self-Study PSP Material*. Retrieved Agososto 2010, 16, from Software Engineering Institute: http://www.sei.cmu.edu/tsp/tools/student/

Sommerville, I. (2006). *Software Engineering 8.* Addison Wesley.

U.S. Bureau of Labor Statistics. (2010). *Computer Software Engineers and Computer Programmers.* Retrieved Agosto 16, 2010, from Occupational Outlook Handbook, 2010-11 Edition: http://www.bls.gov/oco/ocos303.htm

Wells, D. (2000). *Extreme Programming Project*. Retrieved Agosto 16, 2010, from Extreme Programming: A gentle introduction: http://www.extremeprogramming.org/map/project.html

Wells, D. (2009). *Extreme Programming: A gentle introduction*. Retrieved Agosto 16, 2010, from Extreme Programming: http://www.extremeprogramming.org/

Žagar, M., Bosnić, I., & Orlić, M. (2008). Enhancing software engineering education: a creative approach. *Proceedings of the 2008 international workshop on Software Engineering in east and south europe* (pp. 51-58). Leipzig, Germany: ACM.