U

P

T

UNIVERSIDAD POLITÉCNICA DE TULANCINGO

Área de Ingeniería.

Identificación de patrones creacionales en diseño de software mediante métodos difusos

Tesis presentada por
Victor Hugo Fernández Cruz

para obtener el grado de
Maestría en ingeniería con especialidad en Software

Asesor
M. en C.C. Israel Villar Medina

30 de abril de 2018

Índice general

Índice de figuras

Índice de cuadros

Introducción

El reconocimiento de patrones de diseño es un área de investigación muy interesante para programadores e ingenieros de software.

La apropiada selección y aplicación de los mismos durante el desarrollo de software garantiza la flexibilidad y calidad del software.(Villa, 2011)

El objetivo del trabajo es presentar una técnica para la identificación de patrones de diseño basado en la inteligencia artificial concretamente sobre lógica difusa.

Este modelo constituye una solución útil que puede ser usada en la fase de diseño de arquitectura independientemente del catálogo de patrones.

Sin embargo en esta investigación se presentan resultados basados en el catálogo del grupo de los cuatro (Gang of Four, GoF), clasificados como creacionales, bajo el paradigma Orientado a Objetos implementado sobre el lenguaje de programación Java.

Capítulo 1

Generalidades

La Arquitectura de Software es una disciplina que describe el plan estructural de los elementos de un sistema, el cómo los mismos interactúan y se ajustan a los objetivos de la aplicación (Alexander, 1979).

Frecuentemente la arquitectura de software está definida en términos de conjuntos de patrones que participan en diferentes etapas de la construcción del software.

Los patrones de diseño actúan como puente entre la Ingeniería del Software y el diseño de algoritmos y programación, así también lo hace, la arquitectura del software con los objetivos de negocio en el desarrollo de un sistema, esto también podría ser un patrón, en el cual se encuentra una solución integral al desarrollo de sistemas de software.

El concepto de patrón, se origina en el campo de la arquitectura de la construcción. Es *Christopher Alexander* quien por primera vez utiliza el término en forma similar a como luego ha sido usado ampliamente en la ingeniería del software. En la industria del software se están usando patrones para distintas áreas, esperando que su uso sea más habitual. En consecuencia la arquitectura de software es un artefacto fundamental en el contexto de un sistema de software ya que permite guiar y restringir varias actividades durante el desarrollo, y realizar tareas de mantenimiento y evolución de una manera más sistemática.

De esta manera los patrones de diseño de software pueden verse como estructuras de componentes de software y sus relaciones.

1.1. Planteamiento del problema

En la actualidad los patrones y lenguajes de patrones son las mejores herramientas que los desarrolladores de software han logrado encontrar para obtener el máximo provecho del conocimiento generado a lo largo de muchos años de experiencia, y son considerados la mejor estrategia para sistematizar el conocimiento de expertos en diferentes dominios y en las distintas áreas del proceso de desarrollo de software.

Se han creado y se continúan creando catálogos de patrones para problemas en distintos niveles de abstracción en el desarrollo del software, por ejemplo en la interpretación de la abstracción, a nivel arquitectura, análisis de requerimientos, procesos, metodologías, programación, calidad de software, no solo en el diseño de software, donde el catálogo de patrones se vea como un conjunto de recetas de diseño. Aunque existen clasificaciones de patrones, cada uno es independiente del resto, en la ingeniería del software, un patrón constituye el apoyo para la solución a los problemas más comunes que se presentan durante las diferentes etapas del ciclo de vida del desarrollo de software. En el catálogo de patrones de diseño *The Gang of Four -GOF-* se menciona que estos "describen soluciones simples y elegantes a problemas específicos en el diseño de software orientado a objetos"(Gamma *et al.*, 1994, pág 2,11), esto es un impacto importante y duradero en la disciplina del diseño de software. Debido a que *Design Patterns* se autopromueve sobre el software orientado a objetos únicamente, los desarrolladores de software que no participan en la comunidad de objetos pueden ignorarlo, pero con consecuencias de retraso en la evolución de desarrollo del Software, un equipo de desarrollo de software que lo integren más de tres personas, puede trabajar sin problemas, al ignorar a los patrones, el problema comienza, cuando éste software es implementado y necesita mantenimiento correctivo y preventivo, el problema se agranda cuando los integrantes del primer equipo son sustituidos por otros programadores, la interpretación y técnicas de los desarrolladores originales es diferente a los nuevos desarrolladores, y tal vez entre ellos mismos, cuando el equipo sustituto desea corregir o innovar código tiene que estudiar cada una de las formas, pensar como el equipo original, esto causa pérdidas de tiempo considerables que se traducen en dinero para la empresa desarrolladora, el resultado al desplegar el software, por este equipo puede ser aceptable y confiable, pero con el problema del mantenimiento correctivo y preventivo.

1.2. Hipótesis

El conocimiento de mecanismos para la solución en el desarrollo de sistemas orientados a objetos, así como la importancia del diseño de software como una etapa importante en el desarrollo de sistemas.

El uso de la inteligencia artificial, como una ciencia enfocada en la elaboración de programas basados en comparaciones, contribuyendo a un mayor entendimiento del conocimiento humano.

La lógica difusa como metodología que proporciona una manera simple y elegante de obtener una conclusión a partir de información de entrada vaga, ambigua, imprecisa, que modele matemáticamente y simule la ambigüedad del razonamiento humano, que induzca la deducción sobre el razonamiento subjetivo y abstracto en la identificación de patrones de diseño de software creacionales, en implementaciones de código como mecanismos de solución para mejorar, prevenir y corregir sistemas de software orientados a objetos.

1.3. Objetivos

Como consecuencia del problema expuesto se comprende que la arquitectura del software es el puente entre el sistema de software y las metas de negocio del cliente, así mismo los patrones de diseño, actúan como puente entre la ingeniería del software y diseño de algoritmos de programación.

La idea es, que los diseñadores de software usen y conozcan patrones para entender mejor la abstracción en el desarrollo de un sistema de software y dejar la interpretación subjetiva en la programación, mejorando la estandarización y la calidad durante el proceso de desarrollo de software. El paradigma *orientado a objetos* replantea algunos de los problemas, temas, prácticas y conceptos centrales de otros paradigmas, especialmente del *paradigma estructurado* y a la vez da lugar a nuevos temas, problemas, prácticas y conceptos.

1.3.1. Objetivo general

Identificar patrones creacionales de software, con métodos matemáticos de lógica difusa, en el diseño de aplicaciones de software orientados a objetos, desarrollados sobre el lenguaje de programación java.

1.3.2. Objetivos específicos

- Analizar el diseño del catálogo de patrones creacionales y el diseño incógnita para obtener la base comparativa, características del diseño orientado a objetos.
- Conocer los métodos y procedimientos de la lógica difusa para transformar las características de diseño a características difusas.
- Desarrollar el procedimiento difuso de inferencia para la identificación en grados de pertenencia del diseño incógnita a los patrones creacionales.
- Dar como resultado la metodología, que pueda ser interpretada, a fín de que el lector pueda tener una idea acertada de patrones de diseño de software, creacionales, orientados a objetos, en el desarrollo de sistemas.

1.4. Justificación

El objetivo planteado en este trabajo resuelve la parte de la ingeniería en la que se lleva al diseñador y al equipo de desarrolladores a utilizar los patrones creacionales del *Gang of Four GoF* que constituye un punto muy benéfico en el desarrollo, aplicando el paradigma orientado a objetos.

Se sabe que todos los diseñadores de software usan patrones, no importa si trabajan desde el paradigma estructurado, orientado a objetos o cualquier otro.

En el diseño de sistemas de software, la identificación de patrones es a menudo un factor de tiempo en el desarrollo de sistemas, el objetivo fundamental es, la búsqueda de métodos y técnicas que permitan producir software de calidad con un mínimo de tiempo de desarrollo, además de que, para arquitectos, desarrolladores y diseñadores sin experiencia estas herramientas de diseño les ayuden a la buena toma de decisiones en el proceso de desarrollo de software.

De esta forma, la necesidad de métodos y herramientas más flexibles que puedan ser utilizadas para identificar patrones de diseño en el desarrollo sistemas, es una oportunidad aprovechada en el presente trabajo para aportar una herramienta de solución a la evaluación de los desarrolladores en su apego a los patrones.

El Reconocimiento de Patrones (*Pattern Recognition*) es una técnica de carácter multidisciplinario cuyo objetivo de estudio son los procesos de identificación, caracterización, clasificación y pronóstico sobre objetos, físicos o abstractos con el propósito de extraer información que permita identificar propiedades de entre conjuntos de objetos, así como metodologías y técnicas relacionadas con dichos procesos (Bunke & Allermann, 1983).

Una de las áreas con mayor aporte en la identificación de patrones es la inteligencia artificial con la aplicación de los conceptos en sus distintas áreas como, redes neuronales, algoritmos genéticos, vehículos autónomos, realidad virtual, agentes (wizards), lógica difusa.

Esta última tomada como base para el presente trabajo toda vez qué no es un clasificador si no una técnica que nos indica que tanto se pertenece a un patrón y por lo tanto una idea clara de lo que sucede en el código fuente de la aplición.

1.5. Alcance

En la presente investigación se identificarán patrones de diseño de software clasificados como creacionales en el catálogo de patrones GoF, en sistemas o aplicaciones orientadas a objetos.

- Factory
- Singleton
- Abstract Factory
- Builder
- Prototype

De igual forma abarca únicamente a las aplicaciones o sistemas desarrollados en el lenguaje de programación java y el uso de procedimientos lógico difusos para identificar grados de pertenencia entre un diseño incógnita y los patrones creacionales.

La clasificación es una de las tres técnicas básicas junto a la diferenciación de la experiencia en objetos particulares, sus atributos, la distinción entre un todo y sus partes, que dominan el pensamiento humano en su proceso de comprensión del mundo, en otras palabras, a esto le llama abstracción (Laube *et al.*, 2005).

En este contexto se enfoca a la identificación de los patrones de diseño en sistemas de software existentes, esto es con el fin de obtener un mejor entendimiento de la abstracción de sistemas entre el diseño de algoritmos y la programación y poder hacer de ellos una base de componentes que tengan las características necesarias para que puedan ser reutilizados estrictamente dentro del concepto de patrón de diseño de software bajo el paradigma orientado a objetos.

En este trabajo se hace uso de la lógica difusa para identificar patrones presentes en sistemas o aplicaciones de software, y que ayuden al desarrollador sin experiencia o al desarrollador experimentado a comprender de mejor manera la solución de problemas durante el desarrollo de algún sistema de software, reducir la curva de aprendizaje en el desarrollo, aportando experiencia calidad y buenas prácticas.

Por otra parte la lógica difusa se define como un sistema matemático que modela funciones no lineales, que convierte la entradas de datos, en salidas acordes con los planteamientos lógicos difusos que usa el razonamiento aproximado, similar al del ser humano.

Con el uso de la lógica difusa para identificar la estructura en un sistema de software, la relación de sus elementos, modelarlo matemáticamente, ayudará a entender el diseño y mejorar la estructura de bajo nivel (*código fuente*), mejorar la calidad del software y ayudará al desarrollador en la creación de sistemas y/o aplicaciones de software.

1.6. Limitaciones

El procedimiento de identificación es basado en el lenguaje de programación java ya que, si se realiza esta identificación sobre otro lenguaje se tendría que contemplar

la semántica propia de la sintaxis de programación.

Otro factor a considerar como limitante desde el punto de vista del desarrollador es la falta de estandarización en los conceptos de orientación a objetos y a la interpretación subjetiva de la codificación del sistema por parte del programador.

La abstracción que es una limitante natural, de la que deriva la complejidad, implementación y uso de herramientas fundamentales de codificación en lenguajes orientados a objetos, así como la importancia de la etapa de diseño para el programador. Una pregunta importante en este contexto es:

¿cómo entiende cada uno de los integrantes del equipo de desarrollo específicamente el programador y el diseñador la solución a los objetivos del sistema?

Es esta la abstracción que describe la vaguedad subjetiva de interpretación, en cualquier nivel de la arquitectura de software, y en la que los programadores solo se centran en la solución y no en el entorno de esa solución es decir en estándares de calidad y buenas prácticas en el desarrollo, como el uso de patrones, que es un problema desde la formación de los nuevos diseñadores y desarrolladores que repercute en la creación de sistemas a nivel empresarial que solicita experiencia en el desarrollo, entre otros.

Capítulo 2

Marco Teórico

2.1. Introducción

En Ingeniería del Software se suelen analizar los diferentes ciclos de vida de construcción del software, metodología de construcción (cascada, espiral, prototipo, entre otros), explotación y calidad del software (métrica, ISO 9000-3, entre otros), y modelos de objetos (OMT, UML, entre otros).(Lado., 2010)

En general se han venido estudiando de forma separada, a la Ingeniería del Software y Diseño de Algoritmos y Programación, sobre esto se suelen aprender las estructuras de un lenguaje de programación y sus características y se implementan en ese lenguaje problemas de diferente índole. (Lado., 2010)

La Arquitectura de Software es una disciplina relativamente nueva, y es el plan estructural de los elementos de un sistema, el cómo interactúan y se ajustan a los objetivos de la aplicación.

Existen múltiples enfoques de la arquitectura, pero la mayoría considera la separación de las diferentes entidades en diversas vistas, frecuentemente está definida en términos de un conjunto de patrones de diseño.

Los Patrones de Diseño actúan como puente entre la Ingeniería del Software, diseño de algoritmos y programación, también lo hace la Arquitectura de Software con los objetivos del negocio, del cliente, en el desarrollo de un sistema, esto también podría ser un patrón en el cual encontramos una solución integral al desarrollo de sistemas de software.(Lado., 2010)

El concepto de patrón, se origina en el campo de la arquitectura, donde se utiliza el término en forma similar a como luego ha sido usado en la ingeniería del software.(DeMarco, 1979)

Christopher Alexander, hace mención que, "Un patrón describe primero un problema que ocurre una y otra vez en nuestro entorno, y a continuación describe el núcleo de la solución a dicho problema, de tal manera que se puede usar la solución millones de veces sin repetir la solución específica una sola vez."(Alexander, 1979). En el contexto de la ingeniería del software, es muy posible que el concepto haya sido usado por primera vez por Johnson (Johnson, 1992) y desde entonces ha evolucionado adquiriendo connotaciones propias.

Por esta razón conviene complementar esta caracterización con los aportes de especialistas en patrones usados en distintas áreas de la ingeniería del software por ejemplo:

1. En el famoso libro de Gang of Four, aparte del texto citado de Alexander, donde se acota que un patrón tiene cuatro elementos esenciales:

   a) Nombre: Es una palabra o frase corta (dos o tres palabras) que simboliza lo fundamental de un patrón, el tipo de problemas que resuelve y las características generales de la solución.

   b) Problema: Caracteriza el tipo de problemas y el contexto asociado a que apunta el patrón.

   c) Solución: Describe los aspectos fundamentales de la solución, incluyendo objetos, asociaciones, así como responsabilidades de los objetos y colaboraciones entre ellos.

   d) Consecuencias: describe los efectos generales sobre un diseño de software al adaptar la solución a un contexto específico, cuáles son sus beneficios e inconvenientes desde el punto de vista de factores como el desempeño, la flexibilidad, la reutilizabilidad, la extensibilidad (ampliación de las funciones) y la portabilidad de un sistema de software.

2. Los aportes de Martin Fowler a este tema son muy apreciados y conviene destacar que su definición de patrón es intencionalmente simple: Un patrón es una

idea que ha sido útil en algún contexto práctico y probablemente será útil en otros. (Fowler, 1997). Este autor argumenta que prefiere esta caracterización simple y poco restrictiva para mantenerse lo más cerca posible de la intencionalidad original de los patrones. Agrega que los patrones pueden presentarse en múltiples formas que reflejan especializaciones útiles para la clase de patrones que cada uno representa.

3. Los catálogos POSA1 (Buschmann & Henney, 2003) y POSA2 (Schmidt *et al.*, 1998, pág 3) han influido fuertemente en el desarrollo del concepto de patrón: Un patrón para la arquitectura de software describe un problema de diseño recurrente que surge en contextos de diseño específicos y presenta un esquema general de solución cuyas conveniencias han sido demostradas.

   El esquema de la solución se especifica describiendo los componentes constituyentes, sus responsabilidades y conexiones, así como la forma en que colaboran POSA1 (Buschmann & Henney, 2003). Aunque ésta caracterización se enmarca en el contexto del diseño arquitectónico de software, sin embargo, su significado puede ser generalizado abstrayendo el carácter técnico de los términos de arquitectura de software y componente.

El mérito de Gang of Four, y otros destacados catálogos de patrones es haber planteado a los patrones como un tema central para el diseño de software agregando un quinto elemento que es fundamental. Las interconexiones, ningún patrón es una isla argumenta Alexander (Alexander, 1979), lo cual significa básicamente que ningún patrón se utiliza en forma aislada y por ende ningún patrón puede ser comprendido sino en el contexto de la red de patrones en que ha sido especificado.

Por esta razón se puede verificar que en todos los catálogos de patrones, la especificación de cada patrón siempre incluye referencias a otros patrones.

La línea base en el tema de patrones de diseño la impone el catálogo Design Patterns: Elements of Reusable Object-Oriented Software (Gamma *et al.*, 1994, págs 81 - 331), en este libro se presenta un conjunto de 23 patrones de diseño identificados a partir del estudio y la experiencia del grupo Gang of Four, quienes se dedicaron a analizar los problemas recurrentes en el desarrollo de software y realizaron una clasificación y agrupación a partir de dos criterios, su propósito y alcance, las categorías definidas son:

1. **Patrones creacionales:** Se ocupan del proceso de creación de clases y objetos, son los encargados de abstraer el proceso de instanciación o creación de objetos, ayudan a que el sistema sea independiente de cómo sus objetos son creados, integrados y representados (Díaz *et al.*, 2005, pág. 13). Los patrones que hacen parte de esta categoría son cinco: Factory Method, Abstract Factory, Builder, Prototype y Singleton.

2. **Patrones estructurales:** Tratan de la composición de clases y objetos, se ocupan de cómo las clases y objetos se agrupan, para formar estructuras más grandes (Díaz *et al.*, 2005, pág. 22). Los patrones en esta categoría se encargan de lograr que los cambios en los requisitos de la aplicación no ocasionen cambios en las relaciones entre los objetos (Guerrero *et al.*, 2013, pág. 23). Los patrones que hacen parte de esta categoría son siete: Adapter, Bridge, Composite, Decorator, Facade, Flyweight y Proxy.

3. **Patrones de comportamiento:** Caracterizan las formas en que las clases o los objetos interactúan y distribuyen la responsabilidad. Son los encargados de las opciones de comportamiento de la aplicación, permitiendo que el comportamiento varíe en tiempo de ejecución, sin estos patrones cada comportamiento tendría que diseñarse e implementarse por separado (Díaz *et al.*, 2005, pág. 20). Los patrones que se agrupan en esta categoría son once: Interpreter, Template Method, Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Visitor.

2.2. Historia

En 1994 se publicó el libro Design Patterns: Elements of Reusable Object Oriented Software escrito por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides (Gamma *et al.*, 1994). Ellos recopilaron y documentaron 23 patrones de diseño aplicados usualmente por expertos diseñadores de software orientado a objetos.

Ellos no fueron los únicos que propusieron el uso de los patrones, pero la idea de patrones de diseño comenzó a tomar fuerza luego de la publicación de dicho libro.

En su libro de los ahora famosos Gang of Four, que en español es **la pandilla de los cuatro** se justifica que un patrón de diseño es una abstracción de una solución en un nivel alto, adicionalmente comenta que los patrones solucionan problemas que existen en muchos niveles de abstracción del desarrollo de software, y que estos abarcan las distintas etapas del desarrollo; desde el análisis hasta el diseño y desde la arquitectura hasta la implementación, incluso que se es neutral respecto al lenguaje, asume que sólo es orientado a objetos, el consejos que describe es en las relaciones, como la composición frente a la herencia, se centra en interfaces frente a implementación, el uso de tipos genéricos, ayudan en el diseño de un programa complejo, dan una descomposición de objetos inicial bien pensada, para que el programa sea escalable y fácil de mantener, fomenta la reutilización técnica y da un alto nivel de abstracción al aplicar el diseño a diferentes situaciones.

2.3. Uso de patrones y sus beneficios

En la industria se están usando patrones para distintas áreas del proceso de desarrollo de software, los tres procesos clásicos de la ingeniería del software (desarrollo, administración y aseguramiento de la calidad) a la fecha, es de esperar que el espectro de actividades cubiertas por los patrones se haya ampliado aún más. Por ejemplo, algunos autores han publicado patrones relacionados con el diseño de la interacción humano-sistema, los patrones que se usan en distintas actividades del proceso de construcción son, por decirlo así, los patrones por antonomasia.

Un patrón es un fragmento nombrado de informaciónón instructiva, que captura la estructura esencial y la visión interna de una familia de soluciones con probado éxito sobre un problema recurrente que surge dentro de un cierto contexto y fuerzas de sistema, en otras palabras, rehusar soluciones que funcionaron bien una vez.

En consecuencia la arquitectura de software es un artefacto fundamental en el contexto de un sistema de software ya que permite guiar y restringir varias actividades durante su desarrollo, y realizar tareas de mantenimiento y evolución de una manera más sistemática.

La arquitectura de software está definida en términos de un conjunto de patrones de diseño de aplicaciones donde la abstracción es el universo.

Por lo tanto la arquitectura de software tiene que ver con realizar una división de un sistema complejo en etapas tempranas de su desarrollo (esto no es exclusivo del software)(Velasco-Elizondo, 2013). De esta manera los patrones de diseño de software pueden verse como estructuras de componentes de software y de las relaciones entre ellos.

En su obra Epígrafe de Toufik Taibi (Taibi, 2007) se cita la siguiente declaración de Tom DeMarco ...este nuevo libro de Gamma, Helm, Johnson y Vlissides tiene un impacto importante y duradero en la disciplina del diseño de software.

Debido a que Design Patterns se autopromueve sobre el software orientado a objetos únicamente, los desarrolladores de software que no participan en la comunidad de objetos pueden ignorarlo, pero con consecuencias de retraso en la evolución de desarrollo de Software.

La idea sería que los diseñadores de software usen patrones para entender mejor las abstracciones de este trabajo.

Los paradigmas orientados a objetos replantea algunos de los problemas, temas, prácticas y conceptos centrales del paradigma estructurado y a la vez da lugar a nuevos temas, problemas, prácticas y conceptos.

Así es como *Design Patterns (GoF)* constituye un punto culminante en el paradigma orientado a objetos. Con este enfoque es sabido que todos los diseñadores de software usan patrones, no importa si trabajan desde el paradigma estructurado o desde el nuevo paradigma orientado a objetos, DeMarco vislumbra que los diseñadores de software podrán mejorar en su trabajo si logran comprender y reutilizar las abstracciones que usan como herramientas en su trabajo cotidiano. (DeMarco, 1979)

En un proceso de desarrollo de sistemas de información, los profesionales del software se enfrentan cada día a una multitud de problemas de distinta magnitud.

Cristoper Alexander en (Alexander, 1979) propone la idea de los patrones desde el ámbito de la arquitectura y se plasma en el software, esta idea de patrón se interpreta por su naturaleza genérica como:

Problema $\longrightarrow$ Solución $\longrightarrow$ Reutilización.

Dentro de la Ingeniería de software, se han creado y se continúan creando catálogos de patrones para problemas en distintos niveles de abstracción, por ejemplo,

dentro del proceso de desarrollo de software que existe a nivel programación, arquitectura, análisis, de requerimientos, proceso y metodologías, calidad de software y no solo de diseño de software.

En la actualidad los patrones y lenguajes de patrones son las mejores herramientas que los desarrolladores de software han logrado encontrar para obtener el máximo provecho del conocimiento generado a lo largo de muchos años de experiencia, y son considerados la mejor estrategia para sistematizar el conocimiento de expertos en diferentes dominios de aplicación y en las distintas áreas del proceso de desarrollo de software.

En la ingeniería del software, un patrón constituye el apoyo para la solución a los problemas más comunes que se presentan durante las diferentes etapas del ciclo de vida del software.

Los patrones de diseño según The Gang of Four -GOF- describen soluciones simples y elegantes a problemas específicos en el diseño de software orientado a objetos" (Gamma *et al.*, 1994, pág. 33). Por otro lado existen escenarios en los sistemas de software que no cuentan con una documentación que describa su arquitectura o si existe, no corresponde a la implementación actual del sistema de software. Varias razones pueden ser la causa de este escenario, entre las más comunes destacan la falta de cultura de documentación durante el diseño e implementación del sistema, la no actualización de documentación cuando el sistema cambió, o la erosión del diseño por falta de seguimiento durante la implementación del sistema. De esta forma, la identificación automática de patrones de diseño es relevante en este contexto.

2.4. El patrón

Un patrón describe con abstracción, una solución experta a un problema. Los patrones solucionan problemas que existen en muchos niveles de abstracción. Existen patrones que describen soluciones para todo, desde el análisis hasta el diseño y desde la arquitectura hasta la implementación.

Estos tipos de patrones se clasifican con la variable de nivel abstracción, dependiendo del contexto particular en el cual aplica según la etapa en el proceso de desarrollo entre los que se pueden clasificar:

1. Arquitectura.
2. Diseño.
3. Idioma.
4. Análisis.
5. Ambientes distribuidos.
6. Negocios.
7. Procesos.

El concepto de patrón no puede ser bien presentado si no se complementa con las consideraciones generales sobre su utilidad. ¿Para qué sirve un patrón?

La respuesta a esta pregunta introduce necesariamente el tema de las distintas clases o categorías de patrones que paulatinamente han surgido en el campo de la ingeniería del software.

Se pueden organizar los patrones según familias de patrones relacionados. La clasificación facilita la búsqueda del patrón más adecuado así como su comprensión.

Gamma clasifica los patrones según dos criterios fundamentales: su propósito y su alcance (Taibi, 2007).

1. El propósito refleja lo que realiza el patrón.
2. El alcance indica si el patrón aplica principalmente a clases u objetos.

Esencialmente un patrón permite describir un problema recurrente junto con su solución, de manera que pueda ser aplicada en diferentes problemas.

El Cuadro 2.2 muestra la clasificación propuesta por Gamma de algunos de los patrones más utilizados actualmente (Taibi, 2007), (dud)

	Creación	***Estructural***	***De Conducta***
Clase	Método de Fabricación	Adaptador (clases)	Interprete Plantilla
Objeto	Fábrica	Adaptador (objetos)	Cadena de Responsabilidad
	Constructor	Puente	Comando
	Prototipo	Composición	Iterador
	Singleton	Decorador	Intermediario
		Fachada	Observador
		Flyweight	Estado
		Apoderado	Estrategia
			Visitante
			Memoria

Cuadro 2.1: Clasificación de Patrones de Diseño

2.5. Descripción y plantillas de patrones

Dependiendo del autor, del nivel de abstracción y de la publicación misma se han presentado varios formatos para encapsular la información de un patrón. Los puntos más significativos que debe contener un patrón son:

1. Nombre: Significativo y corto, fácil de recordar y asociar a la información que sigue.
2. Problema: Un enunciado que describe las metas y objetivos buscados y el contexto.
3. Contexto: Define las precondiciones en las cuales ocurren el problema y su solución.
4. Fuerzas: Descripción de las fuerzas y restricciones relevantes en el problema y como interactúan o entran en conflicto.
5. Solución: Las relaciones estáticas y reglas dinámicas que describen cómo solucionar el problema.
6. Ejemplos: Uno o más ejemplos que ilustren el contexto, el problema y su solución.

7. Contexto Resultante: El estado en el cual queda el sistema después de aplicar el patrón y las consecuencias de hacerlo.

8. Racionalidad: Una explicación justificada de los pasos o reglas en el patrón.

9. Relaciones: relaciones estáticas y dinámicas del patrón con otros.

10. Usos conocidos: Describe ocurrencias del patrón conocidas y su aplicación dentro de los sistemas existentes.

La propuesta de Gamma (Gamma *et al.*, 1994, pág. 6), por ejemplo, es que los elementos esenciales de un patrón son los siguientes:

1. Un nombre del patrón. Es una forma abreviada que pueda darnos una idea del problema al que se aplica, sus soluciones y consecuencias. Al asignar un nombre, estamos facilitando la tarea de diseño puesto que nos comunicamos a un mayor nivel de abstracción. Es bastante difícil encontrar nombres adecuados que sirvan a este propósito.

2. El problema describe cuando aplicar el patrón. Aquí se explica el problema y su contexto. Un añadido útil es el de las condiciones de aplicabilidad del patrón.

3. La solución describe los elementos que constituyen el diseño, sus relaciones, responsabilidades y colaboraciones. Se insiste mucho en que esta solución es como una plantilla que provee una descripción abstracta de un problema de diseño y cómo una disposición general de elementos (en este caso clases y objetos) puede resolverlo.

4. Las consecuencias son los resultados y compromisos de aplicar el patrón. Estos son los elementos esenciales, pero cuando se trata de realizar una descripción concreta de un patrón, la plantilla propuesta estará compuesta por una serie de secciones que permiten una estructura más detallada que la ofrecida por la enumeración de los elementos esenciales.

   Siguiendo a Gamma, encontramos la siguiente lista y descripción de secciones dentro de la plantilla que describe cada patrón.

a) Nombre del Patrón y Clasificación.

b) Intención.

c) También conocido como (Sinónimo).

d) Motivación.

e) Aplicabilidad.

f) Estructura.

g) Participantes.

h) Colaboraciones.

i) Consecuencias.

j) Implementación.

k) Ejemplo en código.

l) Usos conocidos y Patrones relacionados.

El conjunto de patrones para un dominio específico, con el tiempo forman un lenguaje que luego es utilizado en el proceso de desarrollo, y permite resolver problemas con un alto nivel de abstracción, por medio de la reutilización de soluciones exitosas.

Los patrones de diseño son utilizados en niveles de micro arquitectura, donde los componentes son las clases y objetos y los mecanismos de interacción, son los mensajes.

Es precisamente el uso de herramientas estandarizadas, que visualice esta interacción de mensajes de la interrelación entre los elementos del diseño de un sistema basado en patrones, esta herramienta es conocida como Lenguaje de Modelado Unificado.(UML).

2.6. Lenguaje de modelado unificado UML.

Lenguaje Unificado de Modelado (LUM o UML por sus siglas en inglés, Unified Modeling Language), es la creación de Grady Booch, James Rumbaugh e Ivar Jacobson, Ellos trabajaban en empresas distintas durante la década de los años 80 y principios de los 90 y cada uno diseño su propia metodología para el análisis y diseño

orientado a objetos. A mediados de los 90 empezaron a intercambiar ideas entre si y decidieron desarrollar su trabajo en conjunto.

En 1994 Rumbaugh a Rational software corporation, donde ya trabajaba Booch y Jacobson Se crea el consorcio del UML conformado por: Intellicorp, DEC, Hwelett Packard, Microsoft, Oracle, Texas Instruments y Rational (Larman, 1999).

1. En 1997 se crea la versión 1.0 del UML - para generar un lenguaje estándar de modelado.

2. 1998 se creó de inmediatamente la versión 1.1 de UML.

3. 2004 se creó la versión 2.0 del UML.

El UML lenguaje de modelado unificado de sistemas de software más conocido y utilizado está respaldado por el OMG (Object Management Group).

Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema, UML ofrece un estándar para describir un "plano"del sistema (modelo), incluyendo aspectos conceptuales tales como procesos de negocio y funciones del sistema, y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y componentes reutilizables(Larman, 1999).

Es importante resaltar que UML es un "lenguaje de modelado"para especificar o para describir métodos o procesos.

Se utiliza para definir un sistema, para detallar los artefactos en el sistema y para documentar y construir.

En otras palabras, es el lenguaje en el que está descrito el modelo.

Se puede aplicar en el desarrollo de software entregando gran variedad de formas para dar soporte a una metodología de desarrollo de software.

Una exigencia de la gran mayoría de instituciones dentro de su Plan Informático estratégico, es que los desarrollos de software bajo cualquier arquitectura, se formalicen con un lenguaje estándar y unificado.

Es decir, se requiere que cada una de las partes que comprende el desarrollo de todo software de diseño orientado a objetos, se visualice, especifique y documente con lenguaje común.

Se necesitaba de un lenguaje que fuese gráfico, a fin de especificar y documentar un sistema de software, de un modo estándar incluyendo aspectos conceptuales tales como procesos de negocios y funciones del sistema(Larman, 1999).

No obstante no puede compararse con la programación estructurada, pues UML significa Lenguaje Unificado de Modelado, no es programación, solo se diagrama la realidad de una utilización en un requerimiento. Mientras que, programación estructurada, es una forma de programar como lo es la orientación a objetos, sin embargo, la programación orientada a objetos viene siendo un complemento perfecto de UML, pero no por eso se toma UML sólo para lenguajes orientados a objetos.

La estandarización de un lenguaje de modelado es invaluable, ya que es la parte principal del proceso de comunicación que requieren todos los agentes involucrados en un proyecto informático. Si se quiere discutir un diseño con alguien más, ambos deben conocer el lenguaje de modelado y no así el proceso que se siguió para obtenerlo(Larman & Applying, 2001).

2.6.1. Semántica y notación.

Una de las metas principales de UML es avanzar en el estado de la integración institucional proporcionando herramientas de interoperabilidad para el modelado visual de objetos. Sin embargo para lograr un intercambio exitoso de modelos de información entre herramientas, se requirió definir a UML una semántica y una notación.

La notación es la parte gráfica que se ve en los modelos y representa la sintaxis del lenguaje de modelado. Por ejemplo, la notación del diagrama de clases define como se representan los elementos y conceptos como son: una clase, una asociación y una multiplicidad. Y qué significa exactamente una asociación o multiplicidad en una clase?

Un metamodelo es la manera de definir esto (un diagrama, usualmente de clases, que define la notación). Para que un proveedor diga que cumple con UML debe cubrir con la semántica y con la notación.

Una herramienta de UML debe mantener la consistencia entre los diagramas en un mismo modelo. Bajo esta definición una herramienta que solo dibuje, no puede cumplir con la notación de UML (Larman, 1999).

Los diagramas de clases de UML forman la vista lógica.
Los diagramas de interacción de UML constituyen la vista de proceso.
La vista de desarrollo captura el software en su entorno de desarrollo.
Los diagramas de despliegue integran la vista física.
Los escenarios: el modelo de casos de uso.

2.7. Modelado de clases

Un diagrama de clases sirve para visualizar las relaciones entre las clases que involucran el sistema, las cuales pueden ser asociativas, de herencia, de uso y de contenimiento.

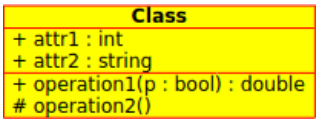


Figura 2.1: Representaconción de una clase UML

Un diagrama de clases está compuesto por los siguientes elementos:

1. Clase: Atributos, Métodos y Visibilidad.
2. Relaciones: Herencia, Composición, Agregación, Asociación y Uso.

1. ***Elementos Clase.***
   Es la unidad básica que encapsula toda la información de un Objeto (un objeto es una instancia de una clase). A través de ella podemos modelar el entorno en estudio (una Casa, un Auto, una Cuenta Corriente, entre otros).

   a) ***Atributos y Métodos:***

   1) **Atributos:** Los atributos o caractersticas de una Clase pueden ser de tres tipos, los que definen el grado de comunicación y visibilidad de ellos con el entorno.
   2) **Métodos:** Los métodos u operaciones de una clase son la forma en como, ésta interactúa con su entorno, éstos pueden tener las características siguientes:(SCHMULLER, 2000)

a′ **public (+):** Indica que el atributo será visible tanto dentro como fuera de la clase, es decir, es accesible desde todos lados.

b′ **private (-):** Indica que el atributo sólo será accesible desde dentro de la clase (sólo sus métodos lo pueden acceder).

c′ **protected (#):**Indica que el atributo no será accesible desde fuera de la clase, pero si podrán tener acceso los métodos de otras clase además de las subclases que se deriven esto es la herencia).

2. ***Relaciones entre Clases:***
   Definido el concepto de Clase, es necesario explicar cómo se pueden interrelacionar dos o más clases (cada uno con características y objetivos diferentes) (SCHMULLER, 2000).

   Cardinalidad de relaciones, en UML, la cardinalidad de las relaciones indica el grado y nivel de dependencia, se anotan en cada extremo de la relación y éstas pueden ser:

   a) uno o muchos: 1..* (1..n).

   b) 0 o muchos: 0..* (0..n).

   c) número fijo: m (m denota el número).

   a) Herencia. (Especialización/Generalización):
   Indica que una subclase hereda los métodos y atributos especificados por una Super Clase, por ende la Subclase además de poseer sus propios métodos y atributos, poseerá las características y atributos visibles de la Super Clase (public y protected).

   b) Generalización.

   En UML, una asociación de generalización entre dos clases, coloca a estas en una jerarquía que representa el concepto de herencia de una clase derivada de la clase base. En UML, las generalizaciones se representan por medio de una línea que conecta las dos clases, con una flecha en el lado de la clase base(SCHMULLER, 2000).

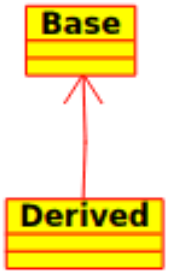


Figura 2.2: Representación visual de una generalización en UML

c) Asociaciones. Una asociación representa una relación entre clases, y aporta la semántica común y la estructura de muchos tipos de conexiones entre objetos.

Las asociaciones son los mecanismos que permite a los objetos comunicarse entre sí. Describe la conexión entre diferentes clases *(la conexión entre los objetos reales se denomina conexión de objetos o enlace).*

Las asociaciones pueden tener un papel que especifica el propósito de la asociación y pueden ser unidireccionales o bidireccionales **(indicando si los dos objetos participantes en la relación pueden intercambiar mensajes entre sí, o es únicamente uno de ellos el que recibe información del otro).** Cada extremo de la asociación también tiene un valor de multiplicidad, que indica cuántos objetos de ese lado de la asociación están relacionados con un objeto del extremo contrario (SCHMULLER, 2000).

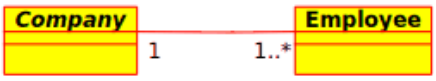


Figura 2.3: Representación visual de una asociación en UML.

En UML, las asociaciones se representan por medio de líneas que conectan las clases participantes en la relación, como la multiplicidad de cada uno de los participantes. La multiplicidad se muestra como un rango [mín...máx] de valores no negativos, con un asterisco (*) representando el infinito en el lado máximo (SCHMULLER, 2000).

d) Acumulación. Las acumulaciones son tipos especiales de asociaciones en las que las dos clases participantes no tienen un estado igual, pero constituyen una relación completa. Una acumulación describe cómo se compone la clase que asume el rol completo de otras clases que se encargan de las partes. En las acumulaciones, la clase que actúa como completa, tiene una multiplicidad de uno(SCHMULLER, 2000).

En UML, las acumulaciones están representadas por una asociación que muestra un rombo en uno de los lados de la clase completa.

Figura 2.4: Representación visual de una relación de acumulación en UML.

e) Composición. Las composiciones son asociaciones que representan acumulaciones muy fuertes. Esto significa que las composiciones también forman relaciones completas, pero dichas relaciones son tan fuertes que las partes no pueden existir por sí mismas. únicamente existen como parte del conjunto, y si este es destruido las partes también lo son.(SCHMULLER, 2000).

En UML, las composiciones están representadas por un rombo sólido al lado del conjunto.

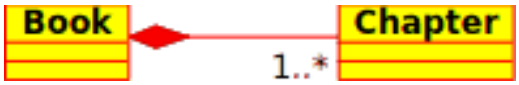


Figura 2.5: Representación visual de una relación de Composición en UML.

2.8. La Lógica difusa

En el mundo en que vivimos existe mucho conocimiento ambiguo e impreciso por naturaleza. El razonamiento humano con frecuencia actúa con este tipo de información. La lógica difusa fue diseñada precisamente para imitar el comportamiento del ser humano.

Las lógicas difusas procuran crear aproximaciones matemáticas en la resolución de ciertos tipos de problemas, pretende producir resultados exactos a partir de datos imprecisos, por lo cual son particularmente útiles en aplicaciones electrónicas o computacionales (Yen & Langari, 1998).

Recientemente, la cantidad y variedad de aplicaciones de la lógica difusa han crecido considerablemente. La lógica difusa es una lógica alternativa a la lógica clásica que pretende introducir un grado de vaguedad en las cosas que evalúa.

La lógica difusa en comparación con la lógica convencional permite trabajar con información que no es exacta para poder definir evaluaciones convencionales, contrario con la lógica tradicional que permite trabajar con información definida y precisa(Cruz, 2011).

La lógica difusa se puede aplicar en procesos demasiado complejos, cuando no existe un modelo de solución simple o un modelo matemático preciso.

Es útil también cuando se necesite usar el conocimiento de un experto que utiliza conceptos ambiguos o imprecisos (Tanaka, 1997).

2.8.1. Introducción

La mayoría de los fenómenos que se presentan en la naturaleza no son necesariamente determinísticos. Los fenómenos tienen implícito cierto grado de imprecisión en su naturaleza y ésta imprecisión puede estar relacionada con la forma, tiempo, espacio, estructura, modo, del fenómeno.

La lógica clásica presenta dos formas categóricas de calificar una proposición; que son por ejemplo, falsos o verdaderos. Pero ¿qué sucede cuando un fenómeno en la naturaleza presenta características que no admiten una evaluación tan categórica, estricta y determinista? Por ejemplo si nos encontramos frente a la situación de juzgar sobre si un individuo de 1,74 m. de estatura es alto o bajo. ¿Podremos concluir que es alto? o ¿que no lo es? Para esta situación, la lógica clásica que conocemos no nos da una respuesta satisfactoria y convincente, este problema lo resuelve la lógica difusa (Geo, 2008).

La lógica difusa permite manejar, definir, aclarar, procesar cierto tipo de situaciones, proposiciones o datos en los cuales se hallen implícitas características de interés inexactas, imprecisas o subjetivas.

La lógica difusa apareció en la mitad de la década de 1960 como una disciplina que cambiaba los conceptos de la lógica clásica o tradicional. Un profesor de Ingeniería Eléctrica y Ciencias de la Computación de la Universidad de California, Lofti A. Zade, introdujo este concepto en 1965 en sus trabajos de sistemas complejos no lineales (Tanaka, 1997).

Desde su aparición muchas publicaciones han ido surgiendo para explicar sus conceptos teóricos fundamentales y sus aplicaciones, especialmente desde los años ochenta cuando las aplicaciones basadas en lógica difusa experimentaron un gran éxito. La lógica difusa es una rama de la inteligencia artificial aunque su aplicación es muy amplia. Desde las ciencias sociales, economía, investigaciones de mercados, ingeniería, entre otros (Geo, 2008).

2.8.2. Lógica difusa

En general, los métodos de inteligencia artificial (IA) son una respuesta al deseo de aproximar el comportamiento y el pensamiento humano a diversos sistemas para la solución de determinadas problemáticas.

Por ello, no es de sorprender que actualmente se tiene sistemas muy avanzados que pueden emular ciertas características humanas, sin embargo aún nos encontramos muy lejos de poder recrear algunas otras(Tanaka, 1997).

En la actualidad los métodos de la inteligencia artificial (IA) tienen un gran auge y muchos investigadores se encuentran estudiando nuevas alternativas en el área.

Hoy en día es común el empleo de sistemas que utilizan la IA para su funcionamiento cotidiano, entre ellos los equipos electrodomésticos como lavadoras, hornos de microondas, cámaras de video, e inclusive sistemas de transporte.

La IA es una de las disciplinas más nuevas, formalmente se inicia en 1956 cuando se acuñó el término, no obstante que ya para entonces se había estado trabajando en ello durante cinco años, junto con la genética moderna, la IA es el campo en que la mayoría de los científicos de otras disciplinas les gustaría trabajar.

El estudio de la inteligencia es una de las disciplinas más antiguas, desde hace más de 2 000 años los filósofos se han esforzado por comprender cómo se ve, se aprende, se recuerda y se razona, así como la manera en que esas actividades deberían realizarse (Jensen & Lopes, 2011).

Búsqueda de soluciones
Sistemas expertos
Procesamiento del lenguaje natural
Reconocimiento de modelos
Robótica
Aprendizaje de las máquinas
Lógica
Incertidumbre y lógica difusa

Cuadro 2.2: Los temas fundamentales de la Inteligencia Artificial

Puede decirse que la lógica clásica es un caso particular de la lógica difusa, o dicho de otro modo, la lógica difusa es una extensión de la lógica clásica.

Debe aclararse que no es la lógica en sí a lo que se denomina "difusa o borrosa"sino a la definición del concepto a que se aplica. Algunos la llaman por esta razón "lógica de los enunciados vagos o lógica de los enunciados difusos".

Por lo tanto esta lógica tiene que ver mucho con el lenguaje que usamos, un lenguaje usado de manera imprecisa, poco clara, es lo que podemos llamar "lenguaje difuso"(Tanaka, 1997).

Cuando una persona responde a un cuestionario, o a preguntas de respuestas múltiples nos encontramos con un caso que puede ser abordado por la lógica difusa.

Las computadoras manejan datos precisos que se reducen a cadenas de unos (1) y ceros (0) y proposiciones que son ciertas y falsas, el cerebro humano puede razonar con información que involucra incertidumbre o juicios de valor como: el aire es frío o la velocidad es rápida, demás, las personas tienen un sentido común que les permite razonar en un mundo donde las cosas son parcialmente ciertas(Tanaka, 1997).

La lógica difusa es una rama de la IA que le permite a una computadora analizar información del mundo real en una escala entre lo falso y verdadero, los matemáticos dedicados a la lógica difusa, en la década de 1920 definieron un concepto clave:

"todo es cuestión de grado"

La lógica difusa manipula conceptos vagos como caliente o húmedo y permite a los ingenieros construir televisores, acondicionadores de aire, lavadoras y otros dispositivos que juzgan información difícil de definir.

Los sistemas difusos son una alternativa a las nociones de pertenencia y lógica que se iniciaron en la Grecia antigua (Tanaka, 1997), cuando los matemáticos carecen de algoritmos que dictan cómo un sistema debe responder a ciertas entradas, la lógica difusa puede controlar o describir el sistema usando reglas de sentido común que se refieren a cantidades indefinidas.

Los sistemas difusos frecuentemente tienen reglas tomadas de expertos, pero cuando no hay experto los sistemas difusos adaptativos aprenden las reglas observando cómo la gente manipula sistemas reales(Jensen & Lopes, 2011).

2.9. Nomenclatura y terminología.

Primero es indispensable establecer cierta nomenclatura y terminología. Cuando se habla de conjuntos nítidos, la variable típica a usar es la "x", en conjuntos difusos la función de pertenencia que se utiliza es la μ . ésta toma los valores entre cero (0) y uno (1), de esta forma la representación de los conjuntos difusos puede ser de dos maneras, de forma continua o discreta, como se presenta a continuación.

Un conjunto difuso se escribe con una tilde arriba del nombre del conjunto.

$$\mu_{\bar{A}}(x) = \{a, b, c\}$$

Ésta sintaxis se utiliza para diferenciarlos de los conjuntos nítidos, en la lógica difusa los conjuntos se pueden presentar en forma continua o discreta.

- *Conjunto difuso discreto:*

$$\mu_{\bar{A}}(x) = \left\{ \frac{\mu_A(X_1)}{X_1} + \frac{\mu_A(X_2)}{X_2} + ... \right\} = \left\{ \sum_i \frac{\mu_A(X_i)}{X_i} \right\}$$

En este punto es importante recordar que el signo (+) no indica suma sino unión.

Dicha forma de representación es muy empleada en los sistemas digitales como los microcontroladores, computadoras.

- *Conjunto difuso continuo:*

$$\mu_{\bar{A}}(x) = \left\{ \int \frac{\mu_A(X)}{X} \right\}$$

El símbolo de integral $\int$ denota unión de elementos del conjunto.

Un conjunto convencional se define por una función característica, que se conoce también como función de pertenencia.

2.9.1. Lógica simbólica

La lógica difusa tiene sus bases en la lógica simbólica. La lógica simbólica permite el establecimiento de un lenguaje artificial empleando símbolos para de esta forma representar argumentos lógicos complicados. Partiendo de *proposiciones,* es decir, de oraciones verdaderas o falsas, es posible traducirlas a un lenguaje de símbolos y representaciones, para posteriormente simplificar y ejecutar operaciones, e incluso traducir nuevamente hacia proposiciones de lenguaje ordinario.(Cruz, 2011)

Una proposición puede ser simple, con valor de Verdadero o Falso, o compuesta, dependiendo de los valores de verdad de componentes simples conectados a partir de operadores como ["y","o", "no"], entre otros.

El operador "y" se denomina conjunción, y se simboliza con $\wedge$, el operador "o" se denomina disyunción, y se simboliza con $\vee$. A continuación se muestran las definiciones de la conjunción y disyunción para dos proposiciones simples **p** y **q** .

p	q	p $\wedge$ q	p $\vee$ q
V	V	V	V
V	F	F	V
F	V	F	V
F	F	F	F

Cuadro 2.3: Tabla de verdad de la conjunción y la disyunción

La negativa de una proposición se denomina *negación.*

2.10. Conjuntos difusos discretos.

Se puede expresar en forma gráfica un conjunto difuso discreto como la unión de sus elementos el número de elementos depende del problema que se va a resolver. Es muy común que a través de una interpolación de cada uno de los elementos se pueda

construir una trayectoria con cada uno de ellos, la cual es la base de una función de membresía o pertenencia. Tomemos como ejemplo el conjunto difuso que se muestra a continuación:

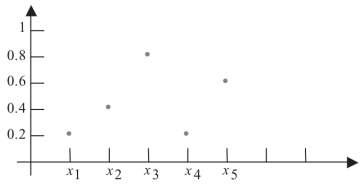


Figura 2.6: Gráfica difusa.

$$\mu_{\bar{A}}(x) = \left\{ \frac{0{,}2}{x_1} + \frac{0{,}4}{x_2} + \frac{0{,}8}{x_3} + \frac{0{,}2}{x_4} + \frac{0{,}6}{x_5} \right\}$$

El conjunto difuso $\tilde{A}$ se define a partir de la gráfica. Como se puede observar, los elementos de este conjunto están formados por fracciones, en las cuales el denominador es el elemento y el numerador es el grado de pertenencia del elemento, pero con una cierta función de membresía. (Cruz, 2011)

Matemáticamente puede expresarse de la siguiente manera:

Sea X un conjunto no vacío. Un conjunto difuso A en X está caracterizado por su función de pertenencia $\mu_A : x \to [0, 1]$ y μ_A es interpretado como el grado de pertenencia del elemento x en el conjunto **A**. Para todo $\forall x \in X$

$$A = \{(u, \mu_A(u)) \mid u \in X\}$$

Frecuentemente se escribe A(x) en lugar de μ_A. La familia de todos los conjuntos difusos en X se denota por F(x).

Si $X = \{x_1, x_2 \cdots x_n\}$ es un conjunto finito y **A** es un conjunto finito en **X**, entonces a menudo se denota A = u_1 / x_1 + $\cdots$ + u_n / x_n donde el término $\mu_i x_i$, i = 1,$\cdots$, **n** significa que μ_i es el grado de partencia del elemento X_i en **A** y el signo + representa la unión. Se define α corte o intervalo, α nivel, notado por A_α

$$A_\alpha = \{x \mid x \in R, \mu_A(x) \geq \alpha\}; \alpha \in [0, 1]$$

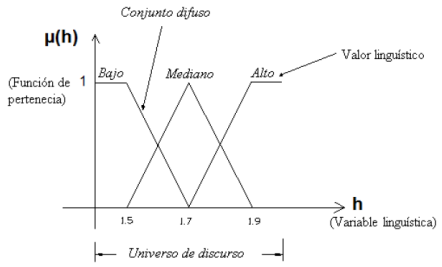


Figura 2.7: Ejemplo de conjunto difuso. (Klir y Yuan 1995).

2.10.1. Operaciones de conjuntos difusos.

Existen operaciones básicas que se definen por operadores binarios para el caso de lógica booleana; en el caso de la lógica difusa se presentan a continuación los operadores que se emplean para realizar las operaciones básicas (unión, intersección y complemento) (Ross, 2009).

Las operaciones básicas entre conjuntos difusos que se definirán son la unión, intersección y complemento, de la siguiente forma:

$$\mu_{\bar{A}\cup\bar{B}}(X) = \mu_{\bar{A}}(X) \vee \mu_{\bar{B}}(X)$$

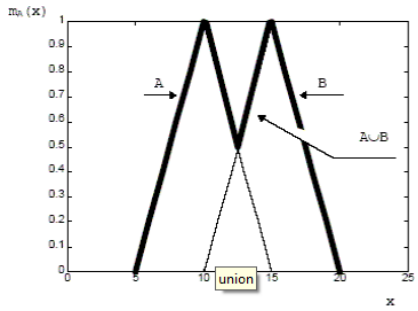


Figura 2.8: Operación Unión.

$$\mu_{\bar{A}\cap\bar{B}}(X) = \mu_{\bar{A}}(X) \wedge \mu_{\bar{B}}(X)$$

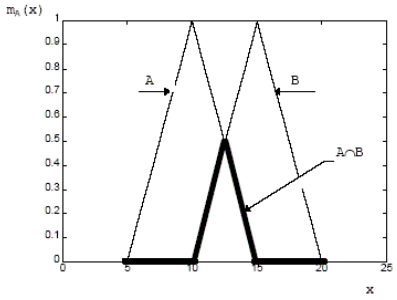


Figura 2.9: Operación Intersección.

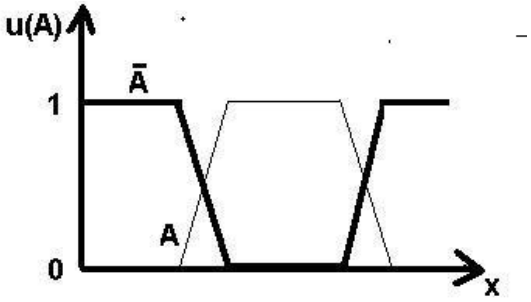


Figura 2.10: Operación Complemento.

$$\mu_{\bar{A}}(X) = 1 - \mu_{\bar{A}}(X)$$

La intersección se clasifica como una norma triangular (norma T) y la unión es una co-norma T (o norma S).

Norma triangular (T)

Las condiciones que definen a la norma T son exactamente las mismas del monoide abeliano parcialmente ordenado en el intervalo unitario real [0,1]. La operación monoidal de cualquier monoide abeliano parcialmente ordenado L se denomina una norma triangular sobre L. Una norma T es una función $\mathbf{T} : [0,1] \times [0,1] \to [0,1]$ que satisface las siguientes propiedades:

Conmutatividad: $\mathbf{T} : (a,b) = (b,a)$
Monotonía: $\mathbf{T} : (a,b) \leq (c,d)\ si\ a \leq c\ y\ b \leq d$

Asociatividad: $\mathbf{T}(a, \mathbf{T}(b,c)) = \mathbf{T}(\mathbf{T}(a,b), c)$
El número 1 actúa como elemento identidad: $\mathbf{T}(a,1) = a$

Considerando que una norma T es una operación algebraica binaria en el intervalo [0,1], la notación algebraica es común y la norma T se denota con un asterisco (*).

Una norma T se denomina *continua* si es continua al igual que una función, en la topología de intervalo usual de [0,1] (Reina & Moscovitz, 2008).

Una norma T se considera *arquimediana* si posee la propiedad arquimediana: para cada x, y en el intervalo abierto (0,1) hay un número natural n tal que x* . . . *x (n veces) es menor o igual a "y".

Una norma T continua y arquimediana se llama estricta si 0 es su elemento único nilpotente, de otra forma se denomina *nilpotente.*(Reina & Moscovitz, 2008)

El orden parcial usual para las normas T se considera a partir de un punto:

$$\mathbf{T}_1 \leq \mathbf{T}_2 si \mathbf{T}_1(a,b) \leq \mathbf{T}_2(a,b) \; para \; todo \; a,b \; en \; [0,1]$$

Como funciones, las normas T largas se consideran más fuertes que las menores. En la semántica de la lógica difusa, no obstante, entre más grande sea una norma T, menor será la conjunción que representa.

Las normas T se emplean para representar intersección en la teoría de conjuntos difusos.

Co-normas T (normas S)

Las co-normas T, también denominadas normas S, son duales a las normas T bajo la operación de reversión de orden, la cual asigna 1 - x a x en el intervalo [0,1].

Dada una norma T, la co-norma complementaria se define como $\perp$ (a,b) = 1-T (1-a, 1-b) . Esto generaliza las leyes de De Morgan.

Una co-norma T satisface las siguientes condiciones, las cuales pueden emplearse para una definición equivalente axiomática independientemente de las normas T:

Conmutatividad: $S(a,b) = S(b.a)$
Monotonía: $S(a,b) \leq S(c,d) \; si \; a \leq c \; y \; b \leq d$
Asociatividad: $S(a,S(b,c)) = S(S(a,b), c)$

El número 1 actúa como elemento identidad: $S\,(a,0) = a$

Las co-normas T se usan para representar *disyunción* en la lógica difusa, y *unión* en la teoría de conjuntos difusos.

2.10.2. Aseveraciones booleanas aplicadas a la lógica difusa

A continuación se muestra dos enunciados que en la lógica booleana son correctos, pero que en la lógica difusa no lo son.

$$\tilde{A} \cup \overline{\tilde{A}} = X \; \tilde{A} \cap \overline{\tilde{A}} = 0$$

En seguida se analiza más de cerca estas aseveraciones desde los principios y operaciones difusas antes expuestos. Siendo:

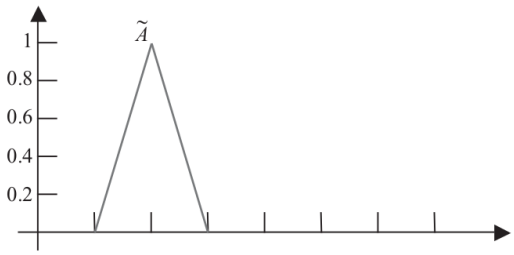


Figura 2.11: Representación gráfica conjunto difuso.

Con base en:

$$\mu_{\bar{A}}\,(X) = 1 - \mu_{\tilde{A}}\,(X)$$

Sabemos que el complemento de $\overline{\tilde{A}}\left(\tilde{A}\right)$ es $\mathbf{1} - \tilde{A}$, por lo que si desarrollamos de manera gráfica podemos ver que:

Si hacemos la unión de $\tilde{A}$ y $\overline{\tilde{A}}$ vemos que no da X, es decir, el universo. De igual forma, si hacemos la intersección podremos observar que no queda un conjunto vacío, como asevera el enunciado, dicho lo cual, tampoco se cumple. En conclusión, se puede observar que a primera instancia pareciera que las dos aseveraciones son correctas, pero, si bien en lógica convencional son ciertas, hay unos axiomas que en lógica difusa no se cumplen(Stytz & Block, 1993).

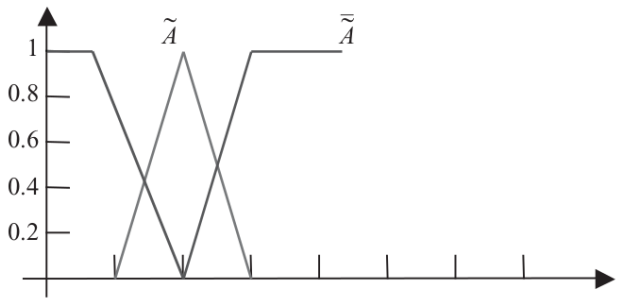


Figura 2.12: Gráfica difusa complemento.

2.11. Operaciones entre conjuntos difusos

A continuación se describe las operaciones más representativas entre conjuntos difusos, ejemplificadas algunas de ellas gráficamente para su mejor apreciación. Es posible considerar que el empleo de estas operaciones de lógica difusa afecta directamente el grado de pertenencia de los elementos que conforman el conjunto difuso, por lo que se tendría descripciones diferentes al emplear estos operadores. Dichas descripciones permiten realizar un mapeo entre los valores nítidos y los valores difusos. Por esta razón, éstas reciben el nombre de funciones de membresía, que son las responsables de mapear los elementos de un conjunto nítido con su grado correspondiente de membresía. Algunas operaciones son el resultado de operaciones con dos o más conjuntos, como es el caso del producto del conjunto difuso.(Cruz, 2011)

2.11.1. Producto de dos conjuntos difusos

$$\mu_{\tilde{A}\cdot\tilde{B}}(X) = \mu_{\tilde{A}}(X) \cdot \mu_{\tilde{B}}(X)$$

Mientras que la potencia es la manipulación de un solo conjunto, esto es elevarlo a una potencia alterando los valores de membresía de cada elemento del conjunto.

Descripción de algunas de las operaciones básicas:

Contención o Subconjunto. Se dice que A es subconjunto de B si todo elemento de A es también elemento de B, es decir, $\mu(A) \leq \mu(B)$. Se define como:

$$A \subseteq B \Leftrightarrow \mu(A) \leq \mu(B)$$

Suma algebraica. La suma algebraica de los conjuntos difusos A y B se define como:

$$\mathrm{C} = \mathrm{A} + \mathrm{B}$$

Su función de pertenencia viene dada por:

$$\mu\left(A + B\right) = \mu_{(A)} + \mu_{(B)} - \mu_{(A)}\mu_{(B)}$$

Producto algebraico. El producto algebraico de los conjuntos difusos A y B se define como:

$$\mathrm{C} = \mathrm{A} \cdot \mathrm{B}$$

Su función de pertenencia viene dada por:

$$\mu\left(A + B\right) = \mu\left(A\right) \cdot \mu\left(B\right)$$

Potencia de orden m. La potencia de orden **m** de un conjunto difuso A es un conjunto difuso cuya función de pertenencia viene dada por:

$$\mu\left(A^m\right) = \left[\mu\left(A\right)\right]^m$$

2.12. Funciones de membresía partes básicas

Para la representación de los grados de pertenencia de cada uno de los elementos que conforman el conjunto difuso, lo más natural es extraer los datos de los fenómenos que se va a representar y con ellos definir la forma de la función de membresía. De otra manera existen metodologías que permiten asignar grados de pertenencia a cada uno de los elementos del conjunto.

Existen funciones de membresía o de pertenencia convencionales y no convencionales que permiten realizar un mapeo de un universo nítido a un universo difuso (grados de pertenencia entre 0 y 1). Entre las funciones de membresía o pertenencia convencionales se tienen las siguientes.

2.12.1. Función de saturación

La función de saturación es la más sencilla de ellas. Tiene un valor de 0 hasta cierto punto y después crece con pendiente constante hasta alcanzar el valor de 1, en donde se estaciona. La figura 2.13. muestra la gráfica de esta función de membresía. Se puede notar que esta gráfica tiene sus cambios de pendiente en los valores 5 y 10.

Este tipo de funciones describe muy bien situaciones en donde se alcanza un nivel máximo a partir de cierto punto, por ejemplo la estatura de las personas o el rendimiento académico de un alumno. Se podría considerar que una persona es alta con grado de pertenencia unitario a partir de 1.90 m, o que un alumno es excelente con grado de pertenencia unitario a partir de un promedio general de 95.

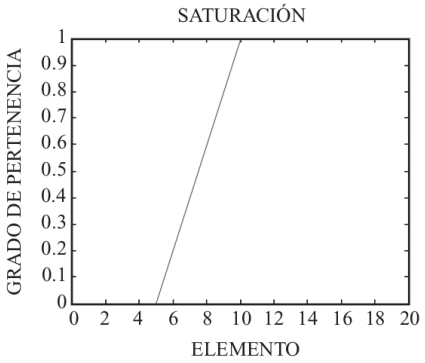


Figura 2.13: Función saturación.

2.12.2. Función hombro

La siguiente función que se muestra es la función hombro, que es, por decirlo de alguna manera, la contra parte de la función saturación. En este tipo de funciones se inicia en un valor unitario y se desciende con, constante pendiente hasta alcanzar el valor de 0. La figura 2.14. muestra la gráfica de esta función de membresía, la cual tiene sus cortes en 9 y 18.

Este tipo de función es útil cuando el grado de pertenencia es total en valores pequeños y decae conforme el valor de la variable aumenta; por ejemplo: el nivel de oxígeno en una pecera; mientras el número de peces no sobrepase un límite contemplado, el oxígeno es suficiente; a medida que el número de peces aumente, el oxígeno

será más limitado hasta que llegue el punto en que no sea suficiente y los peces mueran.

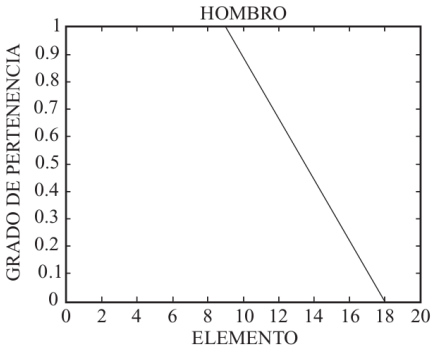


Figura 2.14: Función hombro.

2.12.3. Función triangular

A continuación se muestra la función triangular. Su forma, como su nombre lo indica, consta de una parte de pendiente positiva constante hasta alcanzar la unidad, y una vez que lo ha logrado desciende de manera uniforme. La figura 2.15, muestra un ejemplo de esta función, la cual tiene comienzo en el valor 8 y termina en 12, teniendo el pico en el valor 10.

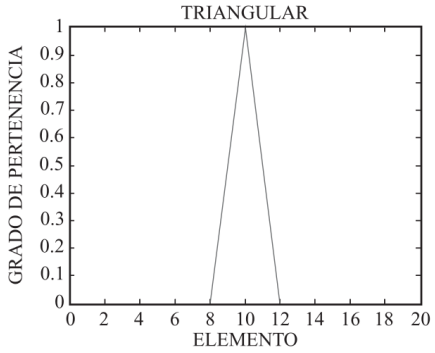


Figura 2.15: Función triangular.

La función triangular es muy adecuada para definir situaciones en las que se tiene un valor óptimo central, el cual se va perdiendo conforme uno se aleja de él.

Un ejemplo de esta situación es la temperatura corporal, que tiene un valor óptimo de 37^0 centígrados, pero que por debajo de 35^0 o por encima de 39^0 se considera peligrosa, es decir, el nivel de pertenencia al conjunto de temperaturas seguras en el cuerpo humano es 0.

2.12.4. Función trapecio o Pi

Una generalización de la función triangular es la función trapecio o función Pi. En el caso de esta función de membresía, no sólo se tiene un valor para el cual la pertenencia es unitaria, sino toda una franja que varía su ancho dependiendo del fenómeno observado. En la figura 2.16 se aprecia que la gráfica empieza a crecer de manera constante en 6, llega al valor unitario en 8 donde se conserva hasta 10 y decrece de manera uniforma hasta 12.

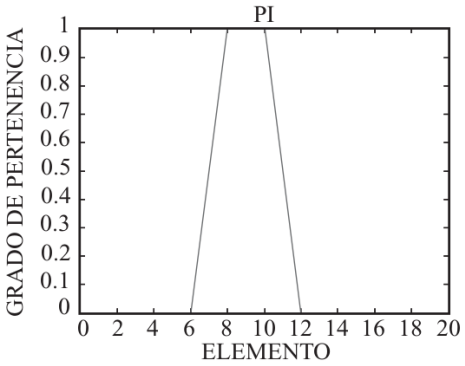


Figura 2.16: Función trapecio o Pi.

La forma de esta función es muy utilizada, ya que como se mencionó se emplea cuando hay un rango de valores óptimos, alrededor de los cuales las condiciones no son adecuadas. Un buen ejemplo de esto es la iluminación de un salón de clases. Existe un rango en el cual la iluminación es agradable para las personas, pero por debajo de dicho rango la luz no es suficiente para leer el pizarrón, y por encima de él es molesto para la vista de los estudiantes.

2.12.5. Función S o sigmoidal

Finalmente tenemos la función S. La forma de esta función es similar a la de saturación. Sin embargo,como su nombre lo indica, el segmento de subida no es una línea recta, sino una curva de segundo orden,la cual cambia de concavidad en un punto dado, y una vez que llega a 1 se mantiene en este valor. En la figura 2.17, se muestra una función de este tipo: la gráfica comienza en 1, tiene un cambio de concavidad en 7 y alcanza el valor máximo en 15.

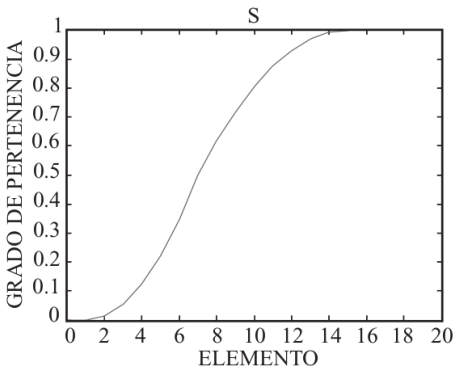


Figura 2.17: Función S.

Esta función también define fenómenos como los definidos por la función de saturación. La diferencia principal radica precisamente en que los cambios de pertenencia a cierto conjunto no son tan drásticos, por lo que se apega más a la realidad. La pertenencia a la clase media basada en el ingreso monetario mensual es un ejemplo que puede ser definido por esta función. (Cruz, 2011)

2.13. Descripción matemática de las funciones de membresía

Las funciones definidas a trozos (o función a trozos o función por partes) son aquellas que tienen distintas expresiones o fórmulas dependiendo del intervalo (o trozo) en el que se encuentra la variable independiente (x).

2.13.1. Función de pertenencia del tipo hombro o saturación derecha.

$$f(x) = \begin{cases} si & x \leq \alpha \rightarrow & 0 \\ si & \alpha \leq x \leq \beta \rightarrow & \frac{x-\beta}{\alpha-\beta} \\ si & x \leq \beta \rightarrow & 1 \end{cases} \qquad véase(2.12.2)$$

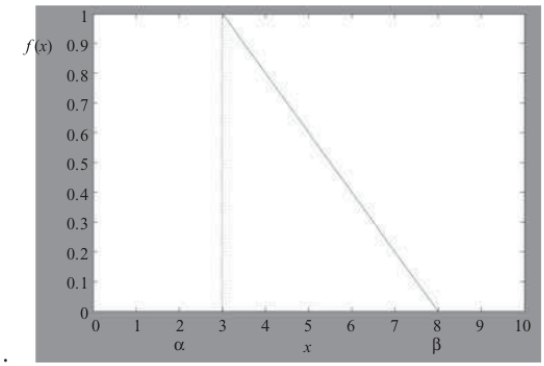


Figura 2.18: Función tipo saturación derecha.

2.13.2. Función de pertenencia del tipo hombro o saturación izquierda

$$f(x) = \begin{cases} \text{si } x \leq \alpha \rightarrow & 1 \\ \text{si } \alpha \leq x \leq \beta \rightarrow & \frac{x-\alpha}{\beta-\alpha} \\ \text{si } x \leq \beta \rightarrow & 0 \end{cases} \qquad véase(2.12.1)$$

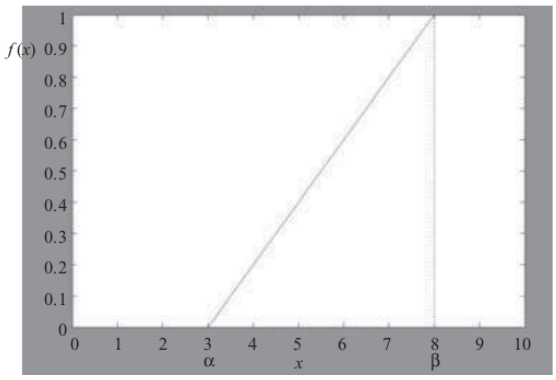


Figura 2.19: Función tipo saturación izquierda.

2.13.3. Función PI

$$f(x) = \begin{cases} \text{si } \alpha \leq x \leq \beta \rightarrow & \frac{x-\alpha}{\beta-\alpha} \\ \text{si } \beta \leq x \leq \varphi \rightarrow & 1 \\ \text{si } \varphi \leq x \leq \delta \rightarrow & \frac{x-\varphi}{\delta-\varphi} \\ 0 \text{ de otra manera} & \end{cases} \qquad véase(2.12.4)$$

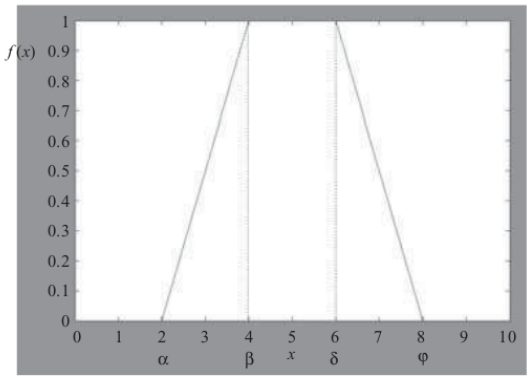


Figura 2.20: Función tipo PI.

2.13.4. Característica tipo triangular

$$f(x) = \begin{cases} \text{si } \alpha \leq x \leq \beta \rightarrow & \frac{x-\alpha}{\beta-\alpha} \\ \text{si } \beta \leq x \leq \delta \rightarrow & \frac{x-\delta}{\beta-\delta} \\ 0 \text{ de otra manera} \end{cases} \qquad véase(2.12.3)$$

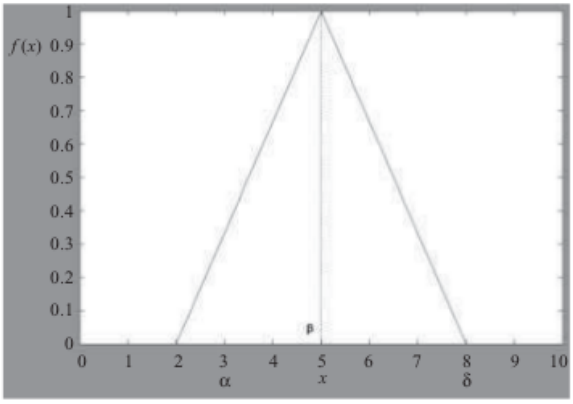


Figura 2.21: Función tipo triangular.

$$f(x) = \begin{cases} \text{si } x \leq \alpha \rightarrow & 0 \\ \text{si } \alpha \leq x \leq \beta \rightarrow & 2\left(\frac{x-\alpha}{\gamma-\alpha}\right)^2 \\ \text{si } \alpha \leq x \leq \gamma \rightarrow & 2\left(\frac{x-\gamma}{\gamma-\alpha}\right)^2 \\ 1 \text{ de otra manera} \end{cases} \qquad véase(2.12.5)$$

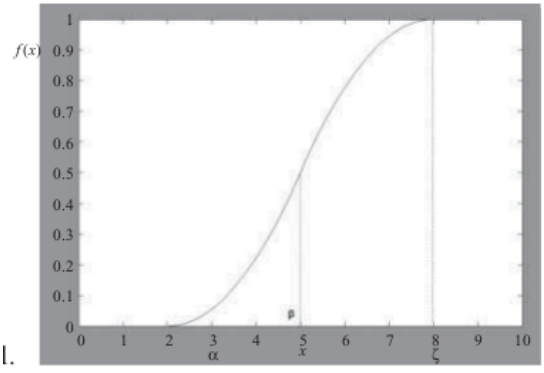


Figura 2.22: Función tipo sigmoidal.

2.14. Lógica difusa y variable lingüística

El concepto clásico de inclusión o exclusión, en la lógica difusa se introduce una función que expresa el grado de **pertenencia** de una variable hacia un atributo o **variable lingüística** tomando valores en el rango de 0 a 1.

El concepto de variables lingüísticas fue introducida por Zadeh (1965) para proporcionar un medio de caracterización aproximada de fenómenos que son demasiado complejos o también difíciles de describir en términos cuantitativos.

Una variable difusa está caracterizada por una terna **(X,U,R(X))** en la cual X es el nombre de la variable, U es el universo de discurso y *R(X)* es un subconjunto difuso de U el cual representa una restricción difusa impuesta por **X**.

Como ejemplo, X= Viejo con:

$U = \{10, 20, \cdots, 80\}$.

R(X)=0.1/20+0.2/30+0.4/40+0.5/50+0.8/60+1/70+1/80

es una restricción difusa sobre Viejo.

Una variable lingüística es una variable de orden más alto que la variable difusa, ésta toma variables difusas como sus valores.

Una variable lingüística está caracterizada por una quíntupla **(x, T(x), U, G, M)** en la cual x es el nombre de la variable; *T(x)* es el conjunto de términos de x , esto es, el conjunto de nombres de los valores lingüísticos de x con cada valor siendo una variable definida sobre **U**; G es una regla sintáctica para generar los nombres de valores de x y M es la regla semántica para asociar cada valor de x con su significado.

Por ejemplo, si Velocidad se interpreta como variable lingüística con U=[0,100] , esto es, x= Velocidad entonces su conjunto de términos T(Velocidad) podra ser **T(Velocidad)=MUY Lenta, Lenta, Moderada, Rápida** .

Aquí la regla sintáctica para generar los nombres (o las etiquetas) de los elementos en T(Velocidad) es muy intuitiva. La regla semántica M podría ser definida como:

M(Lenta)=El conjunto difuso para una velocidad menor a 40 kilmetros por hora con una función de pertenencia μ_{Lenta}.

M(Moderada)= El conjunto difuso para Una velocidad cercana a los 60 km/h con función de pertenencia $\mu_{Moderada}$.

M(Rápida)= El conjunto difuso para Una velocidad arriba de los 90 km/h con una función de pertenencia μ_{Rpida}.

En el ejemplo anterior, el conjunto de términos contiene únicamente un pequeño número de términos, y es práctico listar los elementos de T(x) y establecer una asociación directa entre cada elemento y su significado M.

Una variable lingüística x se dice que es estructurada si el conjunto de términos de T(x) y su regla semántica M pueden ser caracterizados algorítmicamente de modo que puedan ser vistos como procedimientos para generar los términos de T(x) y calcular el significado de cada término en T(x).

2.15. Medidas de similitud geométricas

Las medidas de similitud empleadas habitualmente por los métodos de aprendizaje no supervisado (esto es, los métodos de agrupamiento o clustering) suelen formularse como medidas de disimilitud utilizando métricas de distancia. Cualquier función D que mida distancias debe verificar las siguientes tres propiedades:

Propiedad reflexiva:

$$d(x, x) = 0$$

Propiedad simétrica:

$$d(x, y) = d(x, y)$$

Propiedad triangular:

$$d(x, y) \leq d(x, z) + d(z, y)$$

Existe una clase particular de medidas de distancia que se han estudiado exhaustivamente. Es la métrica R de Minkowski expuesta en el trabajo de Berzal (Galiano, 2002):

$$d(x,y) = \left(\sum_{j=1}^{J} \mid x_j - y_j \mid^r\right)^{\frac{1}{r}}, r \geq 1$$

Como una medida de distancia es, en realidad, una medida de disimilitud, podemos formular la distancia de Minkowsky como una medida de similitud de la siguiente forma:

$$S_{dr}(x,y) = 1 - d_r(x,y)$$

Existen tres casos particulares de la clase de funciones definida por la distancia de Minkowsky (Medina & Luque, 2008) que se utilizan habitualmente en la resolución de problemas de agrupamiento:

La distancia Euclídea (r=2)

$$d(x,y) = \sqrt{\sum_{j=1}^{J} (x_j - y_j)^2}$$

La distancia de Manhattan (r=1)

$$d(x,y) = \sum_{j=1}^{J} \mid x_j - y_j \mid$$

La métrica de domino (conforme r tiende a , la métrica de Minkowski viene determinada por la diferencia entre las coordenadas correspondientes a la dimensión para la que $\mid x_j - y_j \mid$ es mayor):

$$d_\alpha(x,y) = \underset{j=1\ldots J}{max} \mid x_j - y_j \mid$$

existen otros tipos de medidas de similitud tales como:

Medida de Similitud de Chen. Chen presentó una medida de similitud entre números difusos trapezoidales basada en la distancia geométrica.

Medida de Similitud de Lee. Lee propuso una medida de similitud para las números difusos trapezoidales y usó la medida de similitud para hacer frente a las opiniones difusas para la toma de decisiones.

Medida de Similitud de Hsieh y Chen. Propusieron una medida de similitud utilizando la distancia de la representación de la integración media ponderada.

Medida de similitud por Sridevi y Nadarajan. Este método se basa en la diferencia difusa de distancia entre puntos de números difusos, en lugar de distancias geométricas.

Capítulo 3

Estado del Arte

3.1. Introducción

En el desarrollo de software un objetivo fundamental es la instrumentación de métodos y técnicas que permitan producir software de calidad con un mínimo de tiempo de desarrollo.

En este capítulo se presenta el estado del arte en torno a la identificación de patrones de diseño y se describen diferentes investigaciones en las que se incluyen herramientas que implementan métodos para poder realizar el reconocimiento de patrones de diseño en programas escritos en algún lenguaje de programación, se mencionan las ventajas y desventajas de cada una de estas herramientas.

3.1.1. Meta especificación y catalogación de patrones de software con lenguajes de dominio específico y modelos de objetos adaptativos: una vía para la gestión del conocimiento en la ingeniería del software.

Los patrones emergen de la experiencia . Tienen un ciclo de vida que comienza con conocimiento tácito en la cabeza de una persona o grupo de individuos y termina con una descripción explícita y rígida de ese conocimiento que puede ser compartida. El modelo propuesto produce un cambio significativo en este ciclo haciéndolo más dinámico e interactivo, sentando las bases para la evolución y generación de conoci-

miento a partir de interacciones entre los miembros de una comunidad (que puede estar distribuida geográficamente). A partir de esta generación de conocimiento se promueve el refinamiento y mejora continua de los patrones, facilitando así su evolución constante para adaptarse a cambios no previstos en su contexto. (Welicki & Aguilar, 2014)

3.1.2. Patrones de diseño GoF en el contexto de Procesos de desarrollo de aplicaciones orientadas a la Web

En este trabajo se presenta el análisis de identificación de Patrones de Diseño definidos por The Gang of Four (GoF) en procesos de desarrollo de software orientados a la Web. Inicialmente se construye un conjunto de criterios para evaluar y seleccionar procesos de desarrollo formales de gran envergadura. Se establece el tamaño de la muestra para aplicar los criterios con estricto rigor metodológico, se realiza la inspección del código fuente para identificar el uso de patrones de diseño y se lleva a cabo un proceso que permite identificar los patrones de diseño que son utilizados por expertos del área de la ingeniería del software. Los resultados permiten concluir que en el sector productivo los patrones de diseño han sido aplicados. Sin embargo, su uso es reducido por falta de conocimiento de la existencia de estos patrones o por falta de experiencia para lograr su correcta utilización.

Un modelo de meta-especificación y catalogación de patrones y conceptos como respuesta a las necesidades de gestión del conocimiento en éste ámbito de la ingeniería del software. La arquitectura general de la solución propuesta se compone de un lenguaje de meta-especificación para describir a los patrones a un alto nivel de abstracción, un catálogo de patrones creados con ese lenguaje, una infraestructura de catalogación y una herramienta de explotación del catálogo. Para verificar la factibilidad de la solución propuesta se ha creado un prototipo y se evalúo respecto a otras soluciones y enfoques existentes para demostrar que el modelo propuesto supera las dificultades recurrentes encontradas en otros enfoques. (Guerrero *et al.*, 2013, pág.33)

3.1.3. Diseño detección de patrón mediante meta patrones

Uno de los enfoques para mejorar la comprensión del programa es extraer qué tipo de patrón de diseño se utiliza en software orientado a objetos existentes. En este trabajo se propone una técnica para la eficiente y precisa detección de ocurrencias de los patrones de diseño incluidas en los códigos fuente. Utilizan ambos análisis estático y dinámico para lograr la detección con alta precisión. Por otra parte, para reducir los costos de cálculo y mantenimiento, las condiciones de detección se especifican jerárquicamente basado en patrones meta de Pree como estructuras comunes de los patrones de diseño. El uso de Prolog para representar las condiciones de detección nos permite añadir y modificar fácilmente. Por último, se ha implementado una herramienta automatizada como Eclipse plugin y llevado a cabo experimentos con programas Java. Los resultados experimentales muestran la eficacia de este enfoque. (Hayashi *et al.*, 2008, pág. 45)

3.1.4. Aplicación de la teoría de grafos a la Ingeniería de Software Orientados a Objetos (O.O.)

Los ejemplos representativos de la utilidad de la teoría de grafos en los sistemas OO basado en los resultados de investigaciones recientes se presentan en este trabajo.

El estudio de las propiedades de grafos puede ser útil en muchos sentidos para la comprensión de las características de los sistemas de software subyacentes. La Teoría de Grafos, que estudia las propiedades de gráficos, ha sido ampliamente aceptada como un tema central en el conocimiento de los informáticos. Así que es la técnica orientada a objetos (O.O.) en la ingeniería de software, que se ocupa del análisis, diseño e implementación de sistemas que emplean las clases como módulos. Este último campo puede beneficiarse en gran medida de la aplicación de la teoría de grafos, ya que el principal modo de representación, son los diagramas de clases, es esencialmente un grafo dirigido. (Chatzigeorgiou *et al.*, 2006, pág. 120)

3.1.5. Técnica de Identificación del patrón de diseño Composite

En este trabajo se presenta una nueva técnica para identificar los patrones de diseño, compuestos por otro patrón existente en diseños. se propone dos métricas patrón: patrón de cobertura y superpuesta, que pueden ayudar a detectar un patrón compuesto. Los patrones compuestos reflejan propiedades de calidad que se consideran deseables en la solución de problemas, durante el dominio de un problema dado y el paradigma de programación seleccionado. Se identifican candidatos apropiados, se propone una evaluación con un conjunto de métricas de diseño, además de métrica de patrón, se propone la calibración de intervalos de valores para puntuaciones de las métricas con la intención de ofrecer a el diseñador la posibilidad de ajustar la técnica para cada tipo individual de software. En este trabajo, se presentan los pasos necesarios para detectar e identificar los candidatos adecuados de patrón Composite a través de patrones y la evaluación métrica de diseño. (Hericko & Beloglavec, 2005)

3.1.6. El reconocimiento de patrones.

El Reconocimiento de Patrones (Pattern Recognition) es una área de carácter multidisciplinario cuyo objetivo de estudio son los procesos de identificación, caracterización, clasificación y pronóstico sobre objetos, físicos o abstractos con el propósito de extraer información que permita establecer propiedades de/o entre conjuntos de dichos objetos, así como las metodologías y técnicas relacionadas con dichos procesos (Bunke & Allermann, 1983), (dud), (dev).

Los objetos físicos pueden ser:

1. Especiales como: caracteres, imágenes entre otros.
2. Temporales como: formas de onda (voz), series entre otros.
3. Abstractos como: razonamiento, soluciones a problemas.

Así tenemos, por ejemplo, patrones visuales basados en imágenes aéreas o satelitales, de problemas de clasificación y diagnóstico en algunos campos (como la medicina o la balística).

También se puede aplicar a problemas relacionados con el campo del control inteligente, en el cual los sistemas complejos neuronales suministran la capacidad de aprendizaje y la lógica borrosa permite la extracción de las reglas de clasificación o diagnóstico.

A través del tiempo el Reconocimiento de Patrones ha ido evolucionando y tomando distintos nombres como Machine Learning (Aprendizaje Automático) o el más reciente como Data Mining (Minería de Datos) o Knowledge Discovery of Data (KDD, Descubrimiento de Conocimiento en bases de datos) ha medida que se han ido incorporando algunos otros métodos y/o técnica al reconocimiento de patrones como dentro del software.

La investigación de la reingeniería acerca de patrones de diseño, se enfoca a dos áreas: una de ellas se refiere a la identificación de patrones en códigos existentes y la otra es aplicar métodos de refactorización para incluir patrones en los sistemas, con el fin de poder obtener componentes reusables de estos sistemas y poder incluirlos en frameworks o marcos de aplicaciones, todos estandarizados y bajo el paradigma orientados a objetos donde se puedan controlar, evolucionar y administrar estos componentes(Dodge *et al.*, 2008, pág. 230).

Existen algunas familias de problemas que han sido objeto de estudio en la Estadística y paralelamente de la Inteligencia Artificial, proponiendo cada una de estas disciplinas soluciones distintas para alcanzar un mismo objetivo.

Entre estos problemas, uno de los más conocidos es el del Reconocimiento de Patrones que puede ser de clasificación, caracterización o de pronóstico.

El reconocimiento de patrones y en particular la clasificación es uno de los problemas que han sido objeto de estudio de ambas disciplinas, proporcionan diferentes métodos para descubrir en un espacio de representación cuáles son las clases subyacentes, en el problema de clasificación no supervisada. En realidad, las técnicas de clustering son las más populares para separar datos en grupos y una de las técnicas de Minería de Datos más utilizadas (Laube *et al.*, 2005, pág. 126). Es más, la clasificación está entre las tres técnicas básicas (junto a la diferenciación de la experiencia en objetos particulares y sus atributos y la distinción entre un todo y sus partes), que dominan el pensamiento humano en su proceso de comprensión del mundo (Laube *et al.*, 2005, pág. 57).

3.1.7. Implementación de Patrones de diseño en el desarrollo de software.

Dentro de este enfoque existen varias investigaciones, como los trabajos desarrollados por (Budinsky *et al.*, 1996, pág. 102), quienes desarrollaron una herramienta que automatiza la implementación de patrones de diseño; esto es realizado al proporcionarle al sistema algunas piezas de información, nombres de aplicación específicos para los participantes en un patrón, se crea la declaración y definiciones de clases que implementan el patrón; posteriormente, el usuario tiene que adicionar este código al resto de la aplicación.

Otra investigación esta enfocada a la inclusión de patrones de diseño como lenguaje de construcción, donde Jan Bosh de la Universidad de Kariskrona I Ronneby (Bosch *et al.*, 1998, pág, 56) presenta un modelo de objeto en capas, que provee un lenguaje de soporte para la programación.

En otro enfoque, se utilizan los patrones de diseño ya existentes para .solucionar nuevos problemas, como describen Ted Foster y Hiping Zhao (Foster & Zhao, 1998) en el modelado de un sistema de transportes, donde utilizan patrones de diseño que proveen un formalismo para capturar componentes concurrentes, frecuentemente dentro de los modelos de transportes público.

Otras investigaciones han analizado patrones de diseño y han propuesto más clases especializadas de características de lenguaje, con el fin de soportar más limitados subconjuntos de patrones.

Como por ejemplo, Baumgartner, Laufer y Russo (Chambers *et al.*, 2000, pág. 78) quienes analizaron patrones de diseño desde muchas fuentes, e identificaron pocas características clave que podrán adicionarse a la estructura del lenguaje C++ para hacer a los patrones de diseño más fáciles de expresar, todas estas características del lenguaje están presentes en algunos otros lenguajes.

3.1.8. Investigación de patrones de diseño

En este contexto se enfoca a la recuperación de los patrones de diseño en sistemas de software existentes, esto es con el fin de obtener un mejor entendimiento de los sistemas y poder hacer de ellos una base de componentes que tengan las caracterísi-

cas necesarias para que puedan ser reusados. Un método inductivo para descubrir patrones de diseño desde sistemas de software orientados a objetos (Shull *et al.*, 1998, pág. 130).

Se presenta un método inductivo (no automático) que ayuda a descubrir patrones de diseño en sistemas de software orientados a objetos. Proporciona un conjunto de procedimientos definidos rigurosamente para que puedan ser repetibles y utilizables por personas que no están familiarizadas con procesos de ingeniería inversa.

El método propuesto aquí se denomina BACKDOOR *(Backwards Architecture Concerned with Knowledge Discovery of Relationships)*. La principal salida del proceso al aplicar el método, es una base de conocimientos que describe que patrones han sido usados a la fecha por una organización.

Una vez que se han identificado los patrones potenciales, son revisados por los desarrolladores para ver si existe algún significado en el conjunto de clases identificadas. Este método consta de 6 pasos secuenciales e iterativos:

1. Revisión de la documentación de la especificación del problema y el diseño. Se estudia la especificación del problema para identificar restricciones y tópicos del problema, así como para obtener ideas de la funcionalidad proporcionada.

2. Desarrollo de un modelo preliminar del sistema desde la declaración de clases. Aquí se descubre como interactuar las clases (se revisa la declaración de las mismas); se detectan relaciones como herencia, comunicación entre clases.

3. Refinar la notación de objetos desde la implementación de las clases. Se da una mirada detallada al código de implementación de las clases, es decir, se revisa minuciosamente la implementación de las clases para tratar de identificar relaciones entre las mismas.

4. Utilizando el modelo refinado de la arquitectura del sistema, identificar candidatos potenciales a patrones basados en herencia y ligas de comunicación entre clases. Este paso se enfoca a relaciones interesantes en las arquitecturas, buscando similitudes estructurales recurrentes. Relaciones como: clases mediador, interfaz a otras clases, herencia paralela, liga entre dos subsistemas.

5. Análisis de los candidatos a patrones detectados en el se hace una comparación de los patrones potenciales identificados contra los patrones del conjunto de referencia. El conjunto de referencia al principio está compuesto por los patrones de diseño descritos por Erich Gamma en su libro, pero si después se detectan patrones de diseño nuevos, estos se agregan al conjunto de patrones de referencia.

6. Intervención de diseñadores e implementadores para aclarar problemas de diseño. En este paso se tienen entrevistas con los diseñadores y codificadores responsables para un mejor entendimiento del sistema.

Su principal desventaja es que se trata de un método manual, por lo tanto requiere de mucha intervención por parte del usuario. Recuperación de diseño mediante búsqueda automática de patrones de diseño estructurales en software orientado a objetos (Krämer & Prechelt, 1996) se muestra una herramienta denominada PAT (Program Analysis Tool), en esta se extrae información de diseño directamente de los archivos de cabecera de C++ y se almacenan en un repositorio. Los patrones son expresados como reglas Prolog y la información de diseño es también trasladada; una simple consulta de Prolog es usada para la búsqueda de todos los patrones.

En esta herramienta únicamente se reconocen instancias de algunos patrones de diseño estructurales mencionados por Erich Gamma (Gamma *et al.*, 1994, págs. 137 - 207), los cuales son: adapter, Bridge, Compocite, Decorator y Proxy.

Los pasos que sigue PAT para la búsqueda de patrones de diseño son:

1. Cada patrón es representado como un diagrama OMT (Object Modeling Technique) estático, estos diagramas constituyen el repositorio P (patrones).

2. Un programa es usado para convertir P en una regla para cada patrón de diseño.

3. Se utiliza la herramienta CASE para realizar el mecanismo de análisis estructural CASE extrae información de diseño de los archivos de cabecera de C++ y la representa en el repositorio en la notación OMT. La parte resultante del repositorio es llamada D (diseño). La información relevante que es extraída de los archivos de cabecera de C++ es:

   a) Nombre de clases.

   b) Nombre de atributos.

   c) Nombre de métodos.

   d) Relaciones de herencia.

   e) Relaciones de agregación y asociación.

4. Otro programa de este sistema es usado para convertir D en la representación Prolog.

5. Una consulta Q de Prolog detecta todas las instancias de patrones de diseño de P en el diseño D examinado.

La desventaja de PAT es que detecta únicamente patrones de diseño estructurales y no detecta patrones de creación y de comportamiento.

Por otra parte el articulo denomonado:

Identificación automática de patrones de diseño (Bansiya, 1998)

Se presenta una herramienta (DP++) que automatiza la detección, identificación y clasificación de patrones de diseño en programas C++. DP++ actualmente identifica algunos de los patrones estructurales y algunos de los patrones de comportamiento (Behavioral) descritos por Erich Gamma (Gamma *et al.*, 1994, págs. 221 - 345). Esta herramienta se basa en las relaciones estructurales entre las clases y objetos, para identificar usos de patrones de diseño en programas orientados a objetos; relaciones estructurales como:

1. Clases abstractas.

2. Clases base y subclases.

3. Plantillas de clases (template classes).

4. Relaciones de herencia.

5. Agregación por contención física de variables de instancia.

6. Agregación por referencia y apuntador a variables de instancia.

Esta herramienta utiliza un algoritmo de reconocimiento para cada patrón de diseño reconocido en ella. Además despliega una ventana con el modelo de clases de la estructura del programa y una ventana con una vista de árbol que despliega la jerarquía de clases del proyecto.

La principal desventaja de esta herramienta, es que no detecta patrones de diseño de creación (creational patterns).

Capítulo 4

Desarrollo

4.1. Identificación de patrones de diseño GoF.

4.1.1. Introducción

En este capítulo se describe la metodología, en la que se desarrolla la propuesta de investigación denominada, técnica de identificación de patrones de diseño de software, en sistemas y/o aplicaciones, bajo el paradigma Orientado a Objetos, en el cual se presume están desarrollados y sobre el cual se identifican los patrones de diseño de software en el que fueron creados estos sistemas.

La presente metodología se basa en el catálogo de los 23 patrones de diseño, clasificados por el grupo de los cuatro conocidos como *The Gang of Four -GOF-*, ya mencionados anteriores.

Se plantea la estructura general del proyecto y se comprueba que los principios de la lógica difusa son útiles para el modelado matemático de software.

Lo anterior es con el fin de diseñar sistemas de software y/o aplicaciones, que permitan mejorar el desempeño para los que fueron o serán creados, y por lo tanto contribuir a obtener calidad en el desarrollo, mejoramiento en técnicas de solución a problemas en la programación de sistemas, que enfrentan programadores y diseñadores principiantes o experimentados, que vienen de otros paradigmas, reducir de manera significativa la curva de aprendizaje y malas prácticas, mejorar la estandarización entere programadores, diseñadores y analistas, entre otras causas, como

mejorar el proceso de desarrollo y aprendizaje del software, que se extiende en cada nivel de la arquitectura de software.

Cabe hacer mención que la presente metodología solo es el proceso y modelado lógico difuso, que sigue a la identificación de patrones creacionales, bajo el paradigma orientado a objetos que implementa el lenguaje de programación java y no la herramienta de software que en un futuro se pretende desarrollar bajo algún lenguaje de programación, que automatice el proceso de identificación de patrones en sistemas de software.

En consecuencia y acorde con lo antes mencionado, a la identificación de patrones de diseño en sistemas incógnita se obtendrá como salida información no solo de los patrones al cual perténse el diseño del sistema analizado a bajo nivel, si no también información que permitirá mejorar detalles en el desarrollo, estructura, estilo y técnica de programación, calidad, proceso de desarrollo, estandarización y en muchas otras partes de la arquitectura de software, que permitan las buenas prácticas en el desarrollo a programadores, diseñadores y analistas, sin o con poca experiencia en el software orientado a objetos.

4.1.2. Sistemas de información y lógica difusa.

Los sistemas de información realizan cuatro actividades básicas: entrada, almacenamiento, procesamiento y salida de información. Por otra parte la lógica difusa se define como un sistema matemático que modela funciones no lineales, y convierte las entradas en salidas acordes con los planteamientos lógicos que usa el razonamiento aproximado, similar al del ser humano, de esta manera se ve, a un sistema de información como un sistema difuso, como el modelo que matemáticamente, que ayudará a entender el diseño y mejorar la estructura plasmándolo en diseños de sistemas, bajo este paradigma.

4.2. Descripción general de la metodología.

La presente metodología, identifica el diseño de un sistema mediante técnicas de lógica difusa, que a partir de los patrones de diseños Gof clasificados como creacionales, previamente filtrados en piezas de diseño *(Métodos, Atributos)* y piezas de

patrón *(Clases, Interfaces)*, es capaz de extraer la información que contiene el sistema a identificar *(Sistema incógnita)*, es decir, de localizar el diseño de un patrón creacional.

De esta manera, una vez obtenidas las piezas de patrón y de diseño del catálogo GoF, se averiguar de qué patrón son las características de las piezas detectadas en el diseño incógnita filtrando dicho diseño, es decir, descomponiendo el sistema analizado en piezas de diseño y de patrón haciendo una comparación entre la base de conocimiento y las piezas analizadas, en la matriz relación de esta manera se sabrá a qué patrón o patrones corresponde dicho diseño incógnita.

Matriz relación diseño de patrones.
Piezas de Diseño (Pd).
Piezas de Patrón (Pp).

$$A^{Pd \times Pp} \begin{pmatrix} 1 & 1 & ... & 1 & 0 \\ 1 & 1 & ... & 0 & 0 \\ 1 & 0 & ... & 1 & 0 \\ 0 & 0 & ... & 1 & 0 \end{pmatrix}$$

Para ello es necesario contar con una base de conocimiento en la que se encuentren almacenadas las piezas de diseño y de patrón GoF características *(relación de sus piezas)* de los patrones conocidos y utilizados como creacionales, es decir, información de las características de los patrones, utilizando la matriz de diseño de patrones creacionales analizada para registrar la base de conocimiento del catálogo Gof.

Posteriormente se lleva acabo el análisis del sistema incógnita para conocer su diseño, detectando las piezas de diseño y de patrón, estableciendo la matriz relación incógnita como una relación binaria de la misma forma a como se establece la matriz relación base de conocimiento creacional GoF, (figura 4.2) .

Una vez detectadas las piezas de diseño presentes en el sistema o aplicación incógnita, y haciendo uso de los principios, procedimientos y operaciones de números y conjuntos difusos, se realiza la comparación del conjunto de piezas de diseño

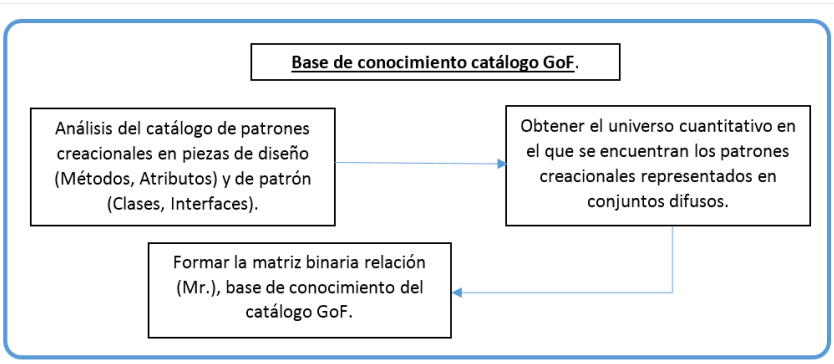


Figura 4.1: Base de conocimiento catálogo GoF.

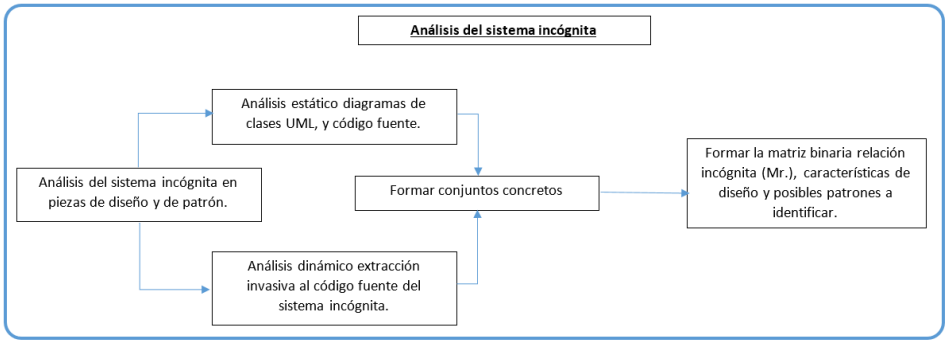


Figura 4.2: Análisis del sistema incógnita.

con la matriz relación base de conocimiento, para descubrir las características y determinar a cuál de los distintos patrones creacionales corresponde el diseño incógnita, a esto se le denomina grado de pertenencia (gp), esta tarea de comparación puede resultar o no laboriosa en función, entre otras cosas, de la experiencia del analista, programador o diseñador y también de los datos que se tenga a cerca de la muestra bajo estudio(clases métodos, atributos, relaciones, conocimiento del paradigma entre otros) para poder descartar las piezas de diseño o de patrón de algunos patrones, de otros catálogos (figura 4.3).

Con el fin de facilitar esta tarea de comparación e identificación, se propone hacer uso de distintas herramientas que ayuden en el analisis conceptual estático y dinámico como, UML, reflexión, orientación a aspectos y, en el proceso de la metodología de identificación, el uso de técnicas de lógica difusa, que completa el ciclo iniciado con el análisis de la identificación de los patrones de diseño GoF, (*esto no es exclusivo a*

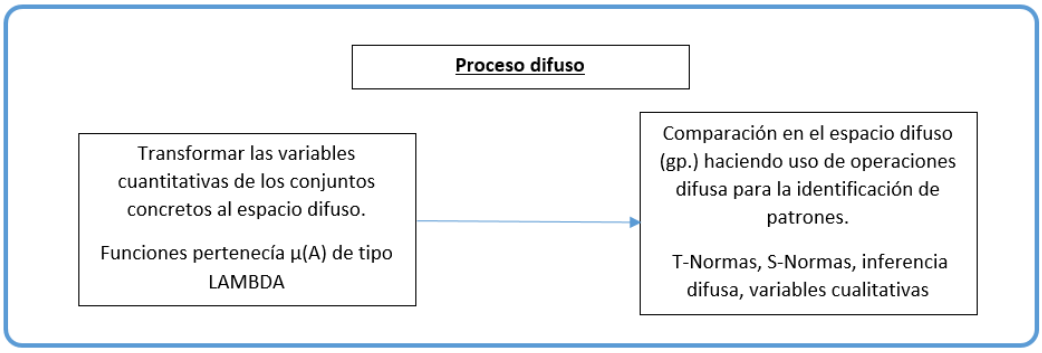


Figura 4.3: Proceso difuso.

patrones creacionales ni mucho menos a los 23 patrones GoF), y determinar cuales son los patrones de diseño de software presente en el sistema incógnita (figura 4.4).

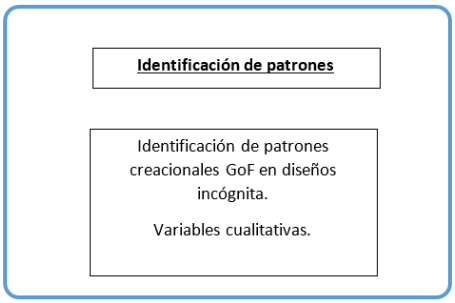


Figura 4.4: Identificación de patrones.

4.3. Reseña de la técnica difusa.

El proceso de identificación sigue las cuatro actividades de un sistema de información, de este modo se entiende a la presente investigación como la estructura que modela matemáticamente con métodos y procedimientos de la lógica difusa, como el motor de inferencia para identificar a los patrones de diseño (figura 4.5), el cual cumple con el objetivo de realizar la comparación entere las piezas de diseño y piezas

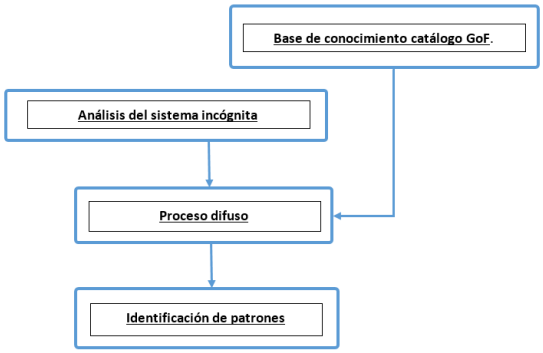


Figura 4.5: Actividades para la identificacióon de patrones GoF.

de patrón obtenidas de la matriz relación basadas en el catalogo GoF, matriz binaria, por cada uno de los patrones de la clasificación creacionales *(Apéndice B)* base de conocimiento, (figura 4.6) y que representa el universo de discurso en el cual se

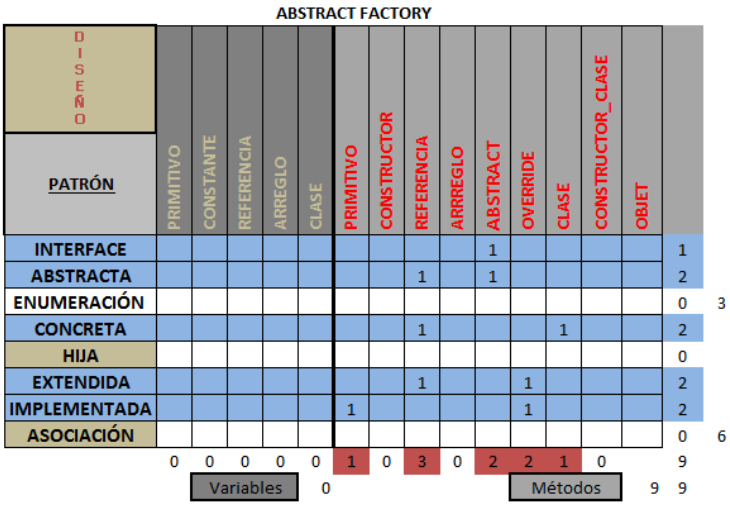


Figura 4.6: Matriz relación, patrón Abstract Factory.

buscaran las coincidencias con el análisis del sistema o *aplicación incógnita, matriz relación incógnita,* teniendo en cuenta que, del conjunto de piezas del diseño detectadas, no todas tienen por qué pertenecer a un mismo patrón sino que puede ser que pertenezcan a más de uno e incluso, que alguna piezas del diseño no pertenezca a ningún patrón y que sean producto del diseño o que pertenezcan a otro catálogo de patrones.

Para ello se propone el esquema (figura 4.7), basado en técnicas del control difuso, para el caso de la presente metodología solo será como referencia, en el proceso de comparación pieza a pieza del diseño del sistema o aplicación a identificar con cada patrón creacional, obteniendo los patrones con los que existan piezas del diseño coincidentes en el sistema a analizado.

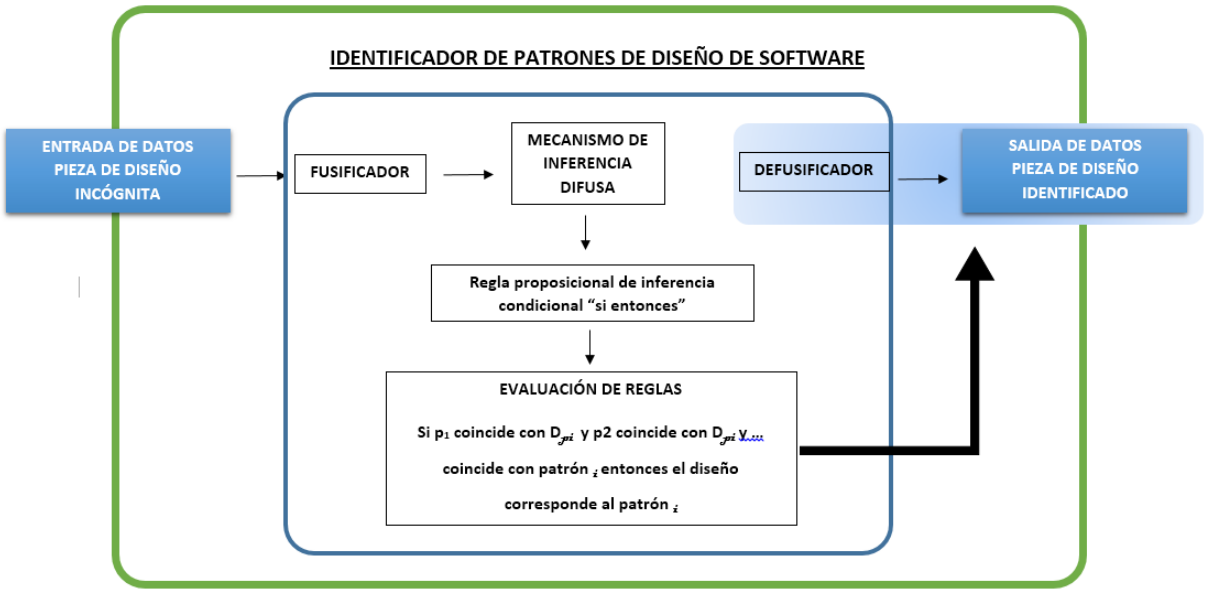


Figura 4.7: Esquema general del control difuso.

Las variables de entrada serán las clases, interfaces, métodos, atributos, accesos y sus relaciones de las piezas de diseño detectadas en el sistema o aplicación incógnita representadas cuantitativamente en piezas de diseño y de patrón (figura 4.8).

Pakage	C	I	PP	M	A	PD
Modelo	2	2	4	6	2	8
Vista	3	4	7	10	0	10
Controler	3	4	7	9	0	9
DAO	1	2	3	6	2	8
Persistencia	1	0	1	6	2	8

Figura 4.8: Piezas de diseño y de patrón Sistema Incógnita.

Estas piezas pueden ser obtenidas mediante inspección por el experto diseñador haciendo uso de la documentación propia del sistema como esquemas de clase bajo el

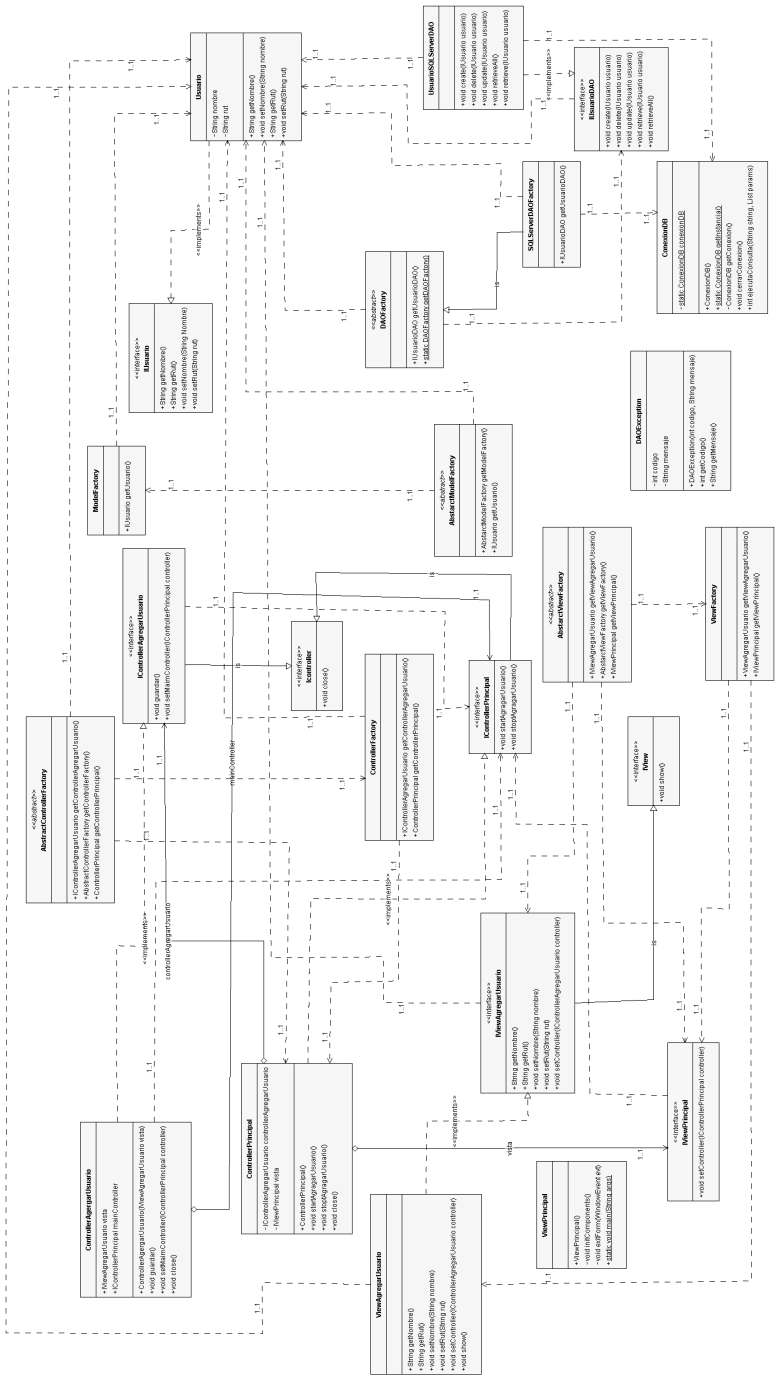


Figura 4.9: Diagrama de clases del sistema incógnita.

estándar UML (figura 4.9), o bien, mediante el análisis invasivo del sistema incógnita *(descomposición dinámica)* representado a través de la matriz binaria relación *(Mr)*(figura 4.10), teniendo así, tantas entradas al sistema como piezas de diseño

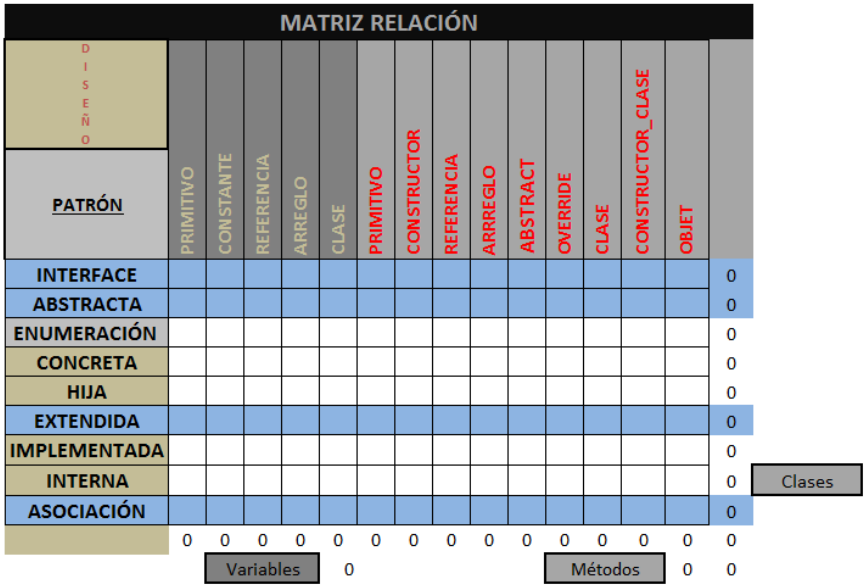


Figura 4.10: Matriz relación entre las piezas del patrón y de diseño.

se detecten durante el análisis del sistema incógnita. Seguido a la identificación de la información de las piezas de diseño y de patrón se realiza la suma de piezas en la matriz binaria, que representa la variable cuantitativa usada en la fusificación de la lógica difusa,(figura 4.11),que establece el universo de discurso en el que se encuentran los patrones creacionales representados en conjuntos difusos (figura 4.12).

Patrón	C	I	PP	M	A	PD	UD
FACTORY	2	1	3	4	0	4	7
SINGLETON	1	0	1	4	2	7	8
ABSTRACT FACTORY	6	3	9	9	0	9	18
BUILDER	9	2	11	6	4	10	21
PROTOTYPE	5	7	12	10	2	12	24

Figura 4.11: Universo de discurso(UD) patrones creacionales Piezas de diseño(PD) y de Patrón(PP).

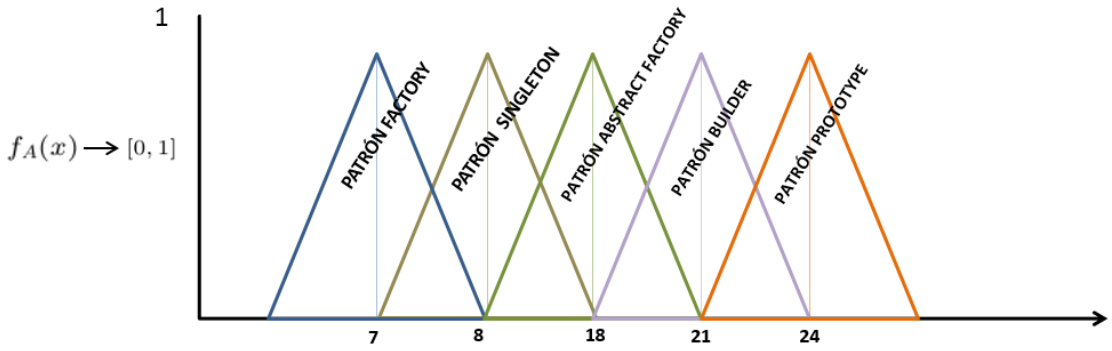


Figura 4.12: Universo de discurso creacionales representado en conjuntos difusos.

Usando la función pertenencia que determinar cuantitativamente a los conjuntos y subconjuntos difusos piezas de diseño y de patrón del sistema incógnita se deducen los valores en grados de pertenencia entre la matriz del sistema incógnita y la matriz relación de los patrones creacionales GoF, obteniendo valores cuantitativos como piezas existan en el sistema incógnita (figura 4.8).

Esta transformación se realiza haciendo uso de las funciones pertenencia (LAMBDA o triangular) para obtener los grados de pertenencia de entre las piezas de diseño del sistema incógnita centrados en las piezas de diseño de los patrones Gof, resultando en una primera instancia las cualidades que podría tener el sistema analizado, basado en la fundamentación axiomática similar a la teoría de probabilidades, *(Axioma Relación de Orden).*

$$\textit{Para todo } (x, y) \in \mathbf{L} \textit{ se tiene que:}$$
$$\mathrm{x} \leq \mathrm{y} \textit{ si existe } \mathrm{z} \in \mathbf{L} \textit{ tal que } \mathrm{y} = \mathrm{x} \vee \mathrm{y}$$

A cada elemento de la retícula se le puede aplicar una función de pertenencia, que indicará su valor de verdad en el intervalo [0, 1] (Reina & Moscovitz, 2008).

La función de pertenencia, denotará que:

$$\mathrm{p} : \mathrm{L} \longrightarrow [0, 1]$$

De igual forma se puede entender a la propiedad transitiva en relación a los conjuntos crisp, recordando que la lógica difusa es una extención a la lógica bivalente,

donde las tres operaciones básicas en conjuntos clásicos que son complemento, intersección y unión, pueden generalizarse a los conjuntos difusos en más de una manera (Gottwald, 2001).

Las variables de salida que se quieren obtener, a partir del conjunto de las piezas de diseño detectado, haciendo uso de la base de datos de patrones creacionales, características de cada patrón, son los patrones a los que puede corresponder el sistema o aplicación bajo estudio.

Con la metodología propuesta, se sabrá que parte del sistema incógnita tienen algunas piezas de diseño que coincidan con las piezas del cataálogo creacional GoF, a lo que se le denomina *grado de pertenencia*, y dirá, que realmente el sistema corresponde a ese patrón, en función de la coincidencia entre sus piezas de diseño.

La correspondencia entre patrón y el sistema analizado, está en función de la coincidencia entre sus piezas de diseño, deduciendo realmente a que patrón corresponde el diseño incógnita, bajo reglas definidas como de inferencia, fundamentadas en las reglas de inferencia de la lógica clásica, éstas son usadas continuamente en el lenguaje natural, extendidas a la lógica difusa para obtener conclusiones válidas.

Las reglas difusas de inferencia se pueden entender de varias formas, conceptualmente, matemáticamente, formalmente entre otras, estas reglas son conocidas en la lógica clásica como modus ponens, modus tollens, silogismo hipotético.

Las reglas de inferencia permiten obtener valores de verdad a partir de valores de verdad probados, la inferencia difusa basada en reglas generaliza el modus ponens permiten que, sus conclusiones inferidas sean modificadas por el grado para el cual el antecedente se satisfaga, lo anterior es la esencia de la inferencia difusa basada en reglas. El razonamiento aproximado, obtiene las conclusiones difusas a partir de proposiciones difusas, utilizando a los conjuntos difusos, para este trabajo se propone hacer uso de la regla composicional de inferencia como herramienta para la identificación recordando que el conocimiento humano se expresa en términos de reglas difusas del tipo:

(IF antecedente THEN consecuente)

describiendo a la condición y conclusión respectivamente, la regla composicional de inferencia traduce el modus ponens de la lógica clásica al lógica difusa expresando la regla como:

SI proposición difusa ENTONCES proposición difusa

Donde la primera proposición podrá definir una relación difusa **R** de cualquier tipo expresada como.

$$B' = A' \circ R$$
$$\mu_B(x) = \mathbf{max}\left[T\left(\mu_A(x), \mu_R(x,y)\right)\right]$$

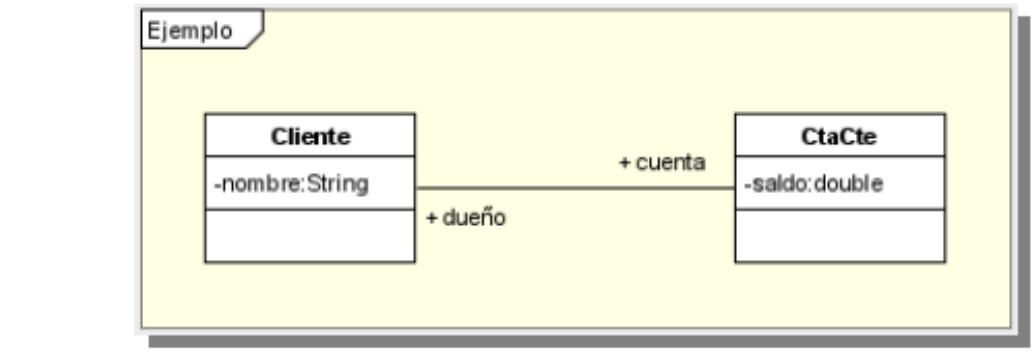
La regla composicional de inferencia proporciona un mecanismo para evaluar el resultado de una regla difusa, la inferencia difusa es un conjunto lingüístico, los antecedentes y consecuentes incluirán proposiciones difusas compuestas, combinando múltiples entradas y salidas

4.4. Proceso de identificación.

Como primer paso en la metodología para la identificación de patrones creacionales sobre el diseño del sistema incógnita, se definen los datos de entrada, como base de comparación que indique mediante el análisis pieza a piezas de patrón y de diseño (clases, interfaces, métodos atributos y relaciones)(Figura: 4.13) obtenidos del código fuente del diseño a identificar, de la misma forma a como se realizo sobre la base de conocimiento definido en (Apéndice A) la estructura de un patrón o patrones, estructura que pertenezca al catálogo de patrones probados y sobre los cuales se basara la técnica, siendo estos, el catálogo de los 23 patrones GoF *Clasificación creacionales*, (figura 4.14.) descritos en el libro *Design Patterns - Elements of Reusable Object Oriented Software*, (Gamma *et al.*, 1994, págs 79 - 321) desarrollados y probados por *Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides* y resumidos en *Design Patterns in Java Tutorial.*(Cooper, 2000) (Apéndice C)

Con la comparación entre la base de conocimiento (Apéndice B) y el sistema incógnita, se busca las diferentes piezas que comprueben el diseño del patrón existente en dicho sistema o aplicación.

Para esta técnica el análisis de códigos,(Apéndice C) es basada en el análisis de aspectos con el objetivo de tener la base para el reconocimiento dinámico bajo este paradigma, obteniendo el resultado en la matriz relación binaria descrita en la sección (Apéndice B) de los diseños, propuestos en el lenguaje java representados en




```
public class Cliente {

    private String nombre;

    public CtaCte cuenta;

}
```

```
public class CtaCte {

    private double saldo;

    public Cliente dueño;

}
```

Figura 4.13: Análisis de diseño incógnita

el esquema de patrones bajo la clasificación creacionales del catálogo GoF (figura 4.15), se extrae las piezas diseño de cada uno de estos patrones descritos en "Design Patterns in Java Tutorial " (Cooper, 2000)

Información que permite saber de cada patrón el número de piezas de diseño, y de patrón, del mismo modo se sabrá el universo entero sobre el cual se define la existencia de cada patrón con su diseño así como el universo total al cual pertenecen todos los patrones de clasificación creacionales y que se pueden representar en conjunto difusos(Figura 4.16).

La expresión matemática en conjuntos concretos basados en el artículo *(A composite desing-pattern identification)*(Heričko & Beloglavec, 2005), en el cual se hace un análisis para detectar patrones compuestos que sirven como una meta-capa de un meta-nivel independiente entre un diseño específico y artefactos conceptuales de dichos patrones donde se representa al diseño y al patrón en términos de conjuntos expresados como:

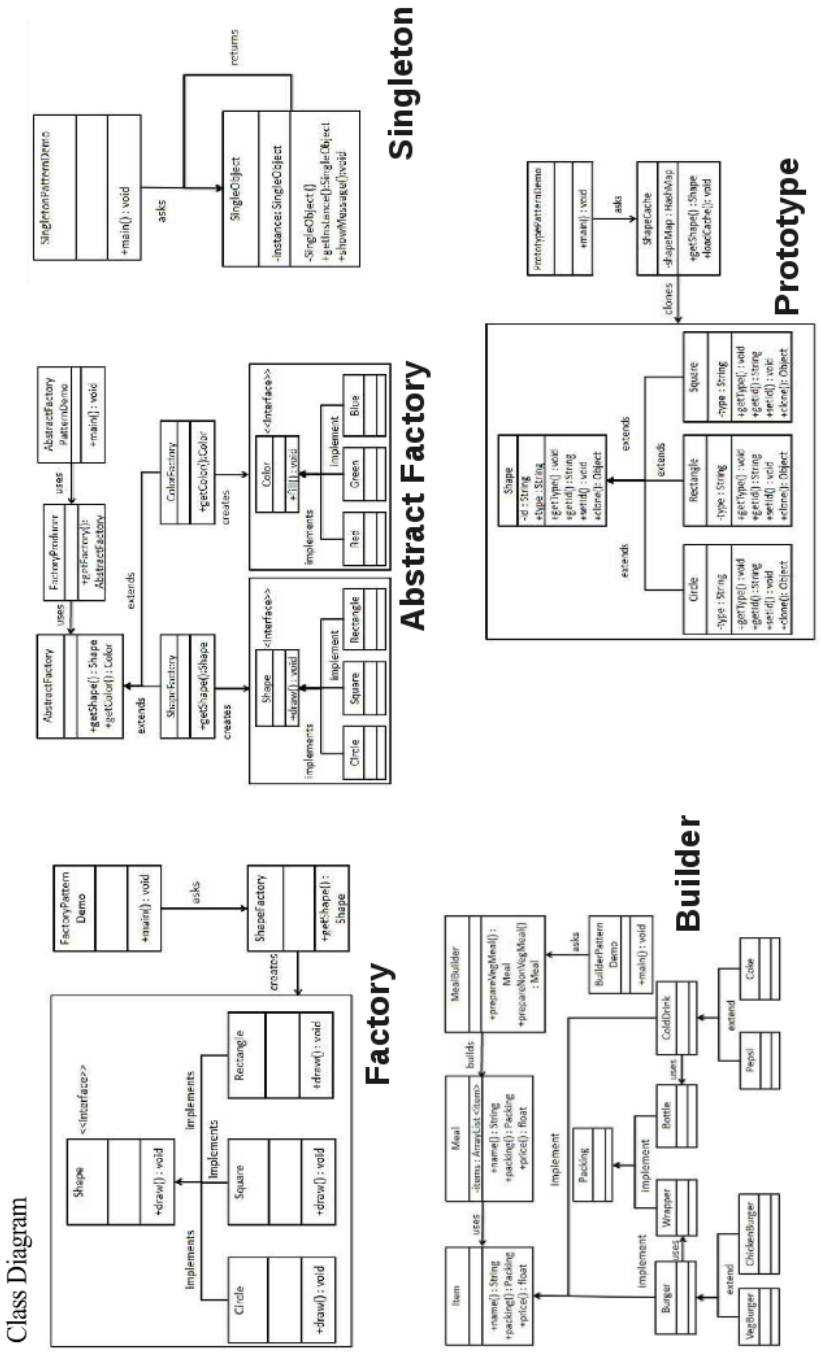


Figura 4.14: Representación UML del Catálogo de los 23 patrones GoF.

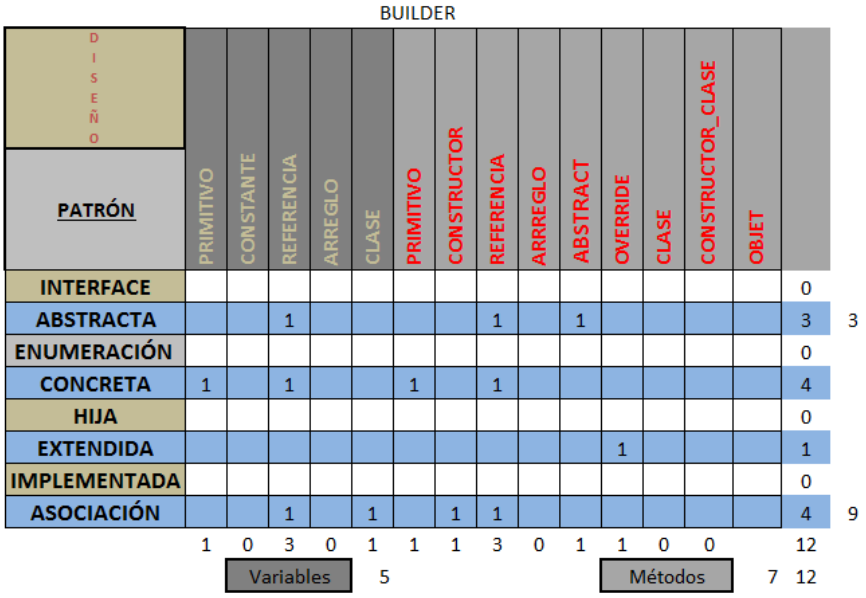


Figura 4.15: Matriz binaria relación patrón base Builder

Elementos del diseño.

$$d = \{e_1^d, \cdots, e_m^d\}$$

Donde d es un diseño o un fragmento del diseño y e_x^d es un elemento del diseño.

Elementos del patrón.

$$p = \{e_1^p, \cdots, e_m^p\}$$

Donde p es un patrón y e_x^d es un elemento del patrón.

Esto se puede interpretar como que $\boldsymbol{p}$ es un patrón y que e_x^p es un elemento del patrón y para cada e_x^p hay una serie de sub-elementos $e_x^p = \{s_i, \cdots, s_j\}$ así mismo se interpreta que e_x^p es un sub-elemento del patrón p.

El diseño puede ser presentado de una manera similar a:

$$d = \{e_1^d, \cdots, e_m^d\}$$

Un diseño o un fragmento del diseño d, para cada elemento e_x^d en la que también existe una serie de sub-elementos de diseño como en el patrón, representados en:

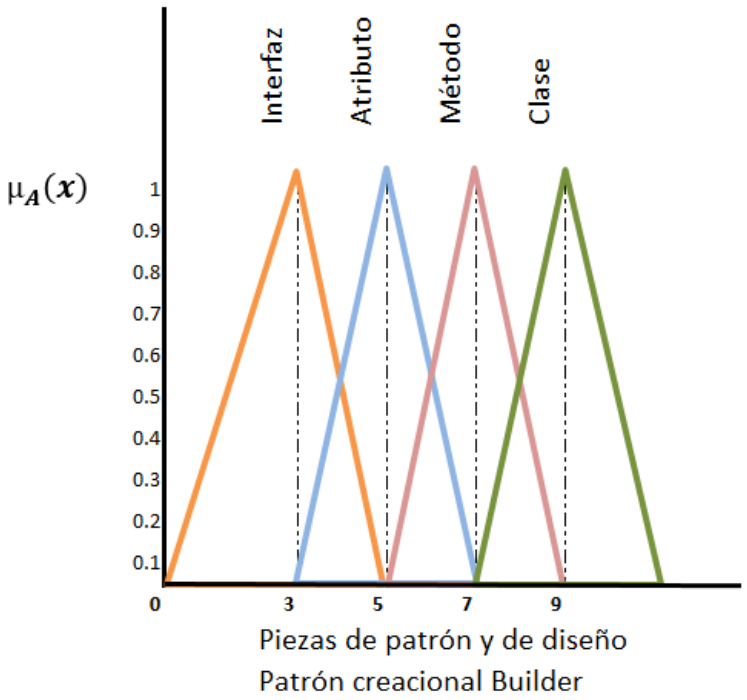


Figura 4.16: Conjuntos difusos Patrón Builder

$$e_x^d = \{s_i, \cdots, s_j\}$$

Un elemento principal de un diseño (métodos y atributos) se puede cubrir con los múltiples elementos de patrón que pertenecen a varios patrones.

En la matriz de análisis las columnas representan las piezas de diseño, mientras que las filas representan las piezas de patrón.

La matriz se puede presentar a través de varios niveles de detalle, que revelan la cobertura del patrón en un nivel que es apropiado para realizar el análisis.

A nivel sub-elemental, los valores de la matriz sólo pueden ser presentados con los valores de 0 ó 1.

El valor 1 significa que un sub-elemento del modelo se instancia en el sub-elemento que se presenta en una fila de la matriz.

En el nivel principal y elemental, la idea es determinar cuántos sub-elementos diseño (atributos y métodos) cubren el principal elemento de un patrón (clase o interface).

Se puede ver que la presunción de conjuntos de los elementos analizados como

piezas del patrón y de diseño están representadas como datos de entrada para el análisis de la técnica presentada.

Los datos necesarios para el análisis de patrón y el diseño requiere que los patrones aplicados en un diseño, deben ser concebidos como conjuntos de los elementos de construcción relacionados, que incluyen clases, interfaces, métodos y atributos. Los métodos y atributos, que son prescritos por un patrón, presentan las piezas de construcción para clases e interfaces de patrones.

Las clases e interfaces se conocen como los principales elementos de un patrón o un diseño, mientras que los métodos y atributos de una clase o interfaz se denominan como sub-elementos de un patrón o un diseño.

La matriz define con precisión la forma de patrones instanciados en fragmentos individuales de diseño.

La matriz puede presentar en un modelo de conjunto, un elemento a un nivel de sub-elemento de detalle, el resultado de esto es el valor de la suma de elementos.

En todo el nivel de patrón, los valores como se espera representan la suma de todas la piezas de diseño como de patrón. (figura 4.17)

Pakage	C	I	PP	M	A	PD	UD
Modelo	2	2	4	6	2	8	*12*
Vista	3	6	9	10	0	10	*19*
Controlador	3	6	9	9	0	9	*18*
Dao	1	2	3	5	2	7	*10*
Persistencia	3	0	3	6	2	8	*11*

PATRÓN	C	I	PP	M	A	PD	UD
FACTORY	2	3	5	4	0	4	*9*
SINGLETON	6	0	6	4	2	7	*13*
ABSTRACT FACTORY	3	6	9	9	0	9	*18*
PROTOTYPE	5	7	12	10	2	12	*24*
BUILDER	9	3	16	7	5	12	*28*

Figura 4.17: Suma de piezas de diseño como de patrón.

4.4.1. Normalización del universo de discurso de los datos.

Para representar un conjunto difuso que tenga un solo punto con grado de pertenencia igual a uno, y varíe suavemente a ambos lados de dicho punto, y tenga un comportamiento similar a la distribución normal de probabilidad, y se pueden utilizar las función difusas mencionadas.

El componente del modelo que realiza esta tarea se le conoce como funciones de Pertenencia. Usualmente, en un objeto que sea descrito con varios atributos, cada atributo puede tener un valor dentro de un rango desde un mínimo hasta un máximo, de acuerdo al tipo de variable y el caso en cuestión.

Por ejemplo, si se trata de la edad de un ser humano, el rango pudiera ir de cero (0) años y un valor máximo digamos 120 años (u otro valor a convenir). Mediante la normalización del universo de discurso los datos estandarizan las variables de entrada, para facilitar el procesamiento de los datos. La aplicación de la normalización de los datos dependerá de la medida de comparación que se utilice en el modelo de comparación. Por ejemplo, si es utilizada la medida de Tversky (Tversky & Gati, 1982) en conjunto con el cálculo de área, no es necesario normalizar pues esta se encuentra implícita en la medida de comparación, caso contrario si la medida a utilizar es geométrica como el centro de gravedad o distancia Euclídea, donde hay que conocer el universo del discurso para normalizar los datos de entrada y llevarlos a un valor entre [0,1].

La normalización de los atributos se realiza escalando los valores de forma que se ajusten en un rango especificado, tal como de -1.0 a 1,0 o 0.0 a 1.0. Para métodos basados en distancia, la normalización ayuda a prevenir que atributos con rangos iniciales grandes tengan mayor peso que atributos con rangos inicialmente pequeños. Existen varios métodos de normalización, entre ellos el denominado normalización min-max, que realiza una transformación lineal sobre los datos originales. Supóngase que minA y maxA sean los valores mínimo y máximo de un atributo A. La normalización min-max mapea un valor ν de A a ν' en el rango nuevo minA y nuevo maxA (Han & Kamber, 2006).

Para comparar objetos caracterizados por múltiples atributos imprecisos, en los que los atributos van a ser representados mediante conjuntos difusos, es conveniente que los valores de los universos de discurso estén normalizados sobre el intervalo [0,1].

Dado que en este caso nuevo maxA = 1 y nuevo minA= 0, utilizando el método normalización min-max se tiene:

$$\nu' = \frac{\nu - minA}{maxA - minA}$$

En la figura siguiente se presenta dos funciones de pertenencia en un universo de discurso sin normalizar, y luego su versión equivalente con universo de discurso normalizado utilizando min-max.

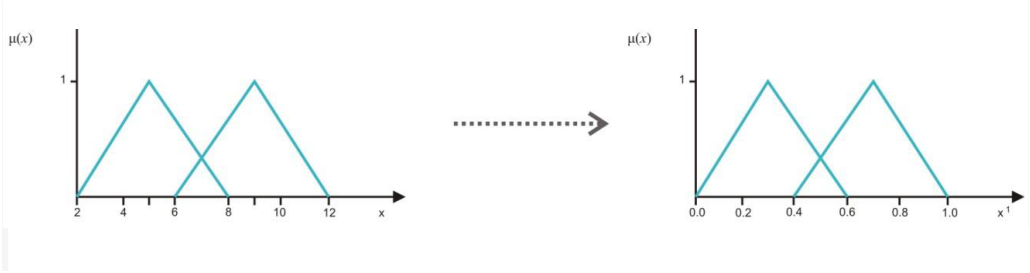


Figura 4.18: Funciones de pertenencia con U sin normalizar (Izq.) y U normalizado con min-max (Der.).

4.4.2. Diseño difuso.

Los parámetros del diseño del sistema, *(esquema de control difuso propuesto)*, (figura 4.7) son los siguientes:

1. Conjuntos difusos y sus funciones características asociadas.

2. Conjunto de reglas difusas que modelan el sistema.

3. Mecanismo de inferencia: elección de los operadores matemáticos correspondientes a los operadores lógicos que aparecen en las reglas, conectivos lógicos, implicación y agregación de las reglas.

4. Defusificador *(opcional)*: método matemático mediante el cual, a partir del conjunto difuso de salida se obtiene el valor crisp (concreto) de la variable de salida.

Los conjuntos difusos de entrada establece el grado de coincidencia o no-coincidencia de las piezas de diseño detectadas con las piezas de diseño características de cada uno de los patrones. Por tanto, definiremos dos conjuntos difusos de entrada, coincidencia y no-coincidencia, para cada uno de los patrones, así, si la base de datos utilizada cuenta con N patrones, tendremos 2N conjuntos difusos de entrada, N conjuntos de coincidencia y N de no-coincidencia.

Las funciones características asociadas a cada conjunto difuso de entrada se construyen de la siguiente manera:

Buscamos una función que, centrada en cada una de las piezas de diseños características del patrón, asigne el mayor grado de pertenencia (valor 1) a la posición exacta de la piezas de diseño, un grado de pertenencia menor a los valores de posiciones de las piezas del diseño incógnita adyacentes en un margen *(Crossover Point)*, para permitir un cierto error *(0.5 intervalo [0,1])* en el cálculo de la posición de la piezas de diseño, y un grado de pertenencia cero para las posiciones de las piezas del diseño incógnita fuera de ese margen (Figura 4.19).

Esta descripción se fundamenta en la propiedad Monotonicidad de la función pertenencia el cual se cumple que:

- **Propiedad de Monotonicidad:**
  si x1 es más próximo a $\boldsymbol{x}$ *que el valor* $\boldsymbol{x2}$*, entonces* $A(\boldsymbol{x1}) > A(\boldsymbol{x2})$

Esta función está definida mediante tres parámetros *(a, b, m)* de la siguiente manera:

Donde "$\boldsymbol{m}$" es el centro de la función, "$\boldsymbol{a}$ "es la semi-anchura (distancia desde el centro de la función hasta el punto en el que la función alcanza la mitad de su valor máximo) y "$\boldsymbol{b}$" es un factor relacionado con la pendiente de la función.

Para construir la función característica de coincidencia utilizaremos funciones Lambda centradas en las piezas de diseños (Interfaz, métodos, clases y atributos) características del patrón

Para cada patrón, la función característica asociada al conjunto difuso coincidencia, μ_{Ci}, presentará una función Lambda en cada una de sus piezas de diseños características y la asociada al conjunto difuso no-coincidencia, μ_{NCi}, que tendrá la forma.

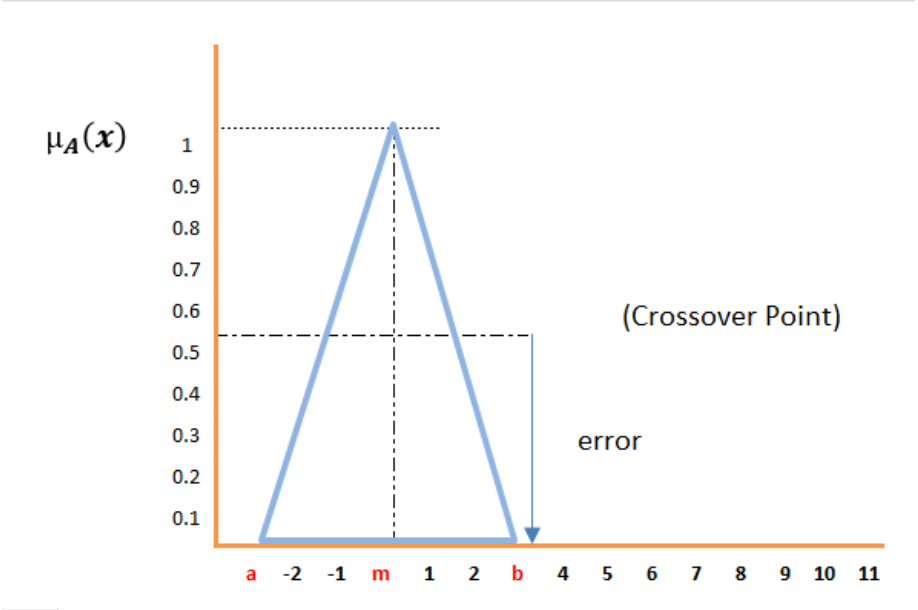


Figura 4.19: Gráfico de la función pertenencía

$$\mu_A(x) = \begin{cases} 0 & x < a \\ \frac{(x-a)}{(b-a)} & a \leq x \leq m \\ \frac{(c-x)}{(c-b)} & b \leq x \leq c \\ 0 & c \leq x \end{cases} \tag{4.1}$$

En la figura 4.16, podemos ver las funciones Lambda , centrada en las piezas de diseño característica "p_c" y acompañada de su expresión matemática.

Se aprecia también que cuando la función de pertenencia alcanza un grado de 0.5, la truncamos, para asegurarnos de no permitir un margen de error mayor al elegido.

En la figura 4.20 se muestran las funciones características completas de coincidencia, obtenida como la suma de las pirámides centradas en cada una de las piezas de diseños del patrón característico, para un patrón-i con "m" piezas de diseños característicos $Pi_1, Pi_2, ..., Pi_m$

Debido a que los conjuntos difusos y funciones características de entrada, coincidencia y no-coincidencia, se han definido de forma que resultan independientes para cada patrón, el sistema difuso propuesto puede interpretarse como N sistemas difusos, uno para cada patrón, cada uno con dos conjuntos difusos de entrada (coin-

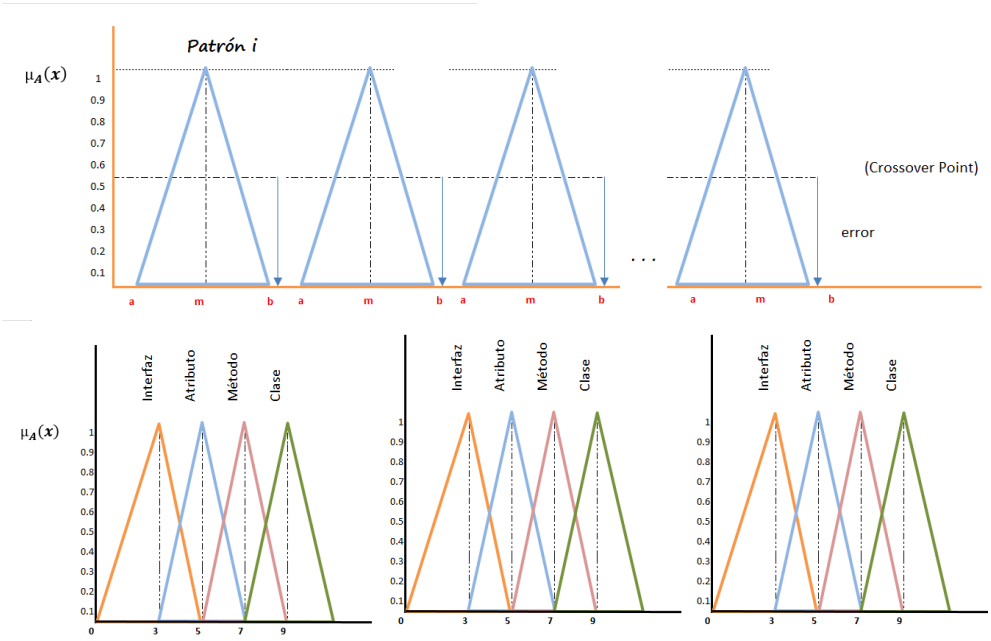


Figura 4.20: Función de características de entrada patrón diseño.

cidencia y no-coincidencia) y dos funciones características asociadas. En la figura 4.21. se muestra cómo sería este sistema para cada uno de los patrones.

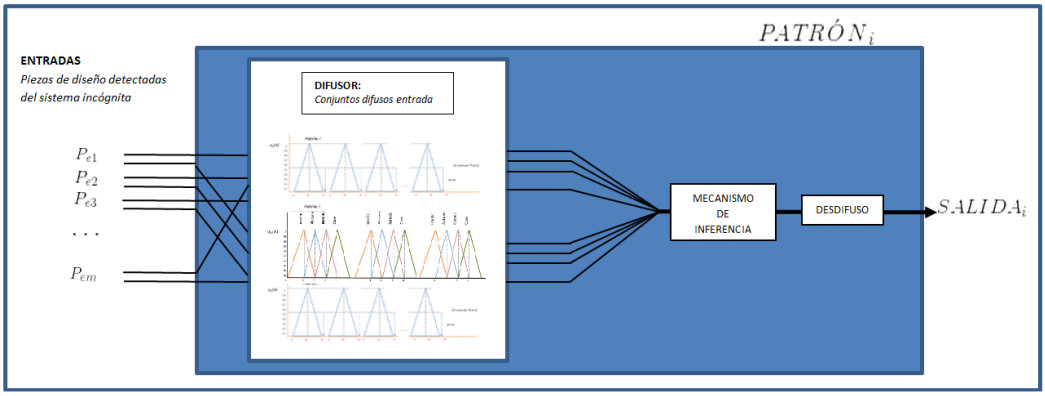


Figura 4.21: Sistema difuso independiente para cada patrón.

La variable de salida es, el nombre del patrón o los patrones, con los que el sistema o aplicación encuentra coincidencias entre las piezas de diseños de entrada y sus piezas de diseños del patrón características, por otro lado, lo que denominamos grado de pertenencia para cada uno de los patrones identificados en función del grado

de coincidencia de las piezas de diseños de entrada con las del patrón y por último el número de piezas de diseños coincidentes respecto al número de piezas de diseños totales del patrón.

Para cada patrón clasificado como creacional (Factory, Abstract Factory, Singleton, Builder, Prototype) con el que se encuentra coincidencia entre sus piezas de diseños y con la entrada de piezas incógnita, el procedimiento propuesto nos proporciona el nombre de ese patrón, por medio del grado de pertenecía obtenido directamente de la inferencia de reglas, las piezas de diseños detectadas y además si entre ellas se encuentra la piezas de diseño o piezas de patrón fundamentales.

Para calcular el grado de pertenencia se propone el conjunto difuso de salida (grado de pertenencia) para cada patrón, cuyo soporte es un único punto, lo que se conoce con el nombre de singleton (Yen & Langari, 1998).

La función característica asociada a este conjunto difuso, *μgp (Pi)*, es la que se muestra en la figura 4.22, una función que toma el valor unidad para el patrón con el que se realiza la comparación en cada momento y cero en cualquier otro caso.

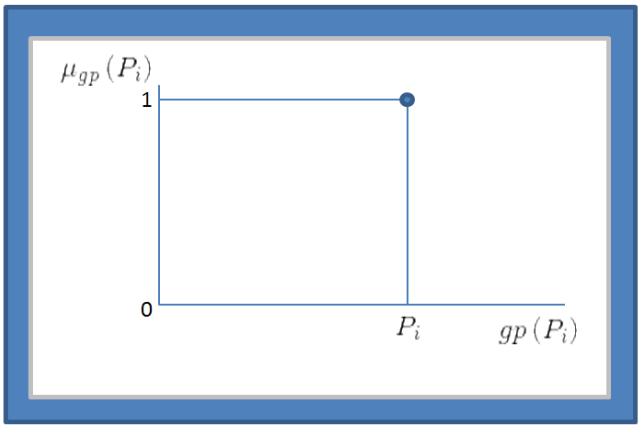


Figura 4.22: Función de características de entrada.

Agrupando la salida para los N patrones considerados en la base de datos, el conjunto difuso de salida sería como se muestra en la figura 4.23.

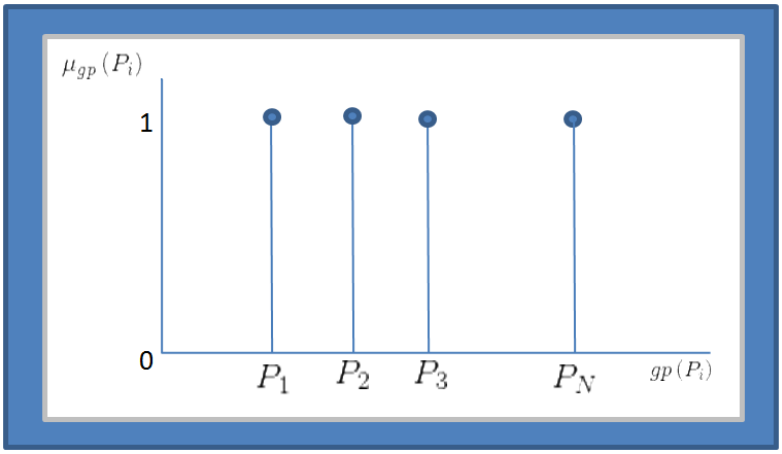


Figura 4.23: Función características de salida completa.

4.4.3. Reglas Difusas

Una regla difusa del tipo ***if-then*** es una representación para capturar el conocimiento *(generalmente conocimiento humano)* que es impreciso e inexacto por naturaleza.

Se utilizan variables lingüísticas para describir condiciones elásticas (condiciones que pueden ser satisfechas por un grado) en la parte ***if*** de la regla difusa.

La característica principal de la inferencia difusa basada en reglas es su capacidad para ejecutar inferencia bajo una combinación (matching) parcial. calculado el grado del dato de entrada al combinar las condiciones de una regla. **La Fig. 3.5** ilustra una forma de calcular el grado de combinación entre una entrada difusa A y una condición difusa A, el grado de combinación es el consecuente (la parte then) de una regla para formar una conclusión inferida por la regla difusa.

Para realizar la comparación de las piezas de diseños $Pe_1, Pe_2, Pe_3, \ldots, Pe_m \ldots$, del sistema o aplicación incógnita "e_i "con todos los patrones $P_1, P_2, P_3, \ldots, P_n$ de la base de datos y obtener el patrón, o patrones (en el caso de que se trate de una mezcla) que correspondan las piezas de diseños del sistema o aplicación, se ha establecido una sub-regla base del tipo:

si Pe_1 *coincide con* P_1 *y* Pe_2 *coincide con* P_1 *y ... y* Pe_m *coincide con* P_1
entonces "e_1*" es* Pe_1 *con gardo de pertenencia gp en le intervalo 0,1.*

En ellas se observa que compara cada una de las piezas de diseños detectadas con las piezas de diseños de un patrón; esto se repite para cada patrón y por tanto se define una sub-regla base como la anterior para cada uno de los N patrones.

El número de antecedentes de la sub-regla depende del número de piezas de diseños que hayan sido detectadas en el sistema o aplicación incógnita.

En realidad, como se ha visto en las reglas semejantes a esta, se puede descomponer en tantas reglas como combinaciones posibles existen entre los antecedentes y los conjuntos difusos de entrada, esto es la de finición del conjunto potencia.

Así, el número de reglas teóricamente posibles P_t para cada patrón (teniendo en cuenta que sólo tenemos un consecuente), será:

$$P_t = mL_m \, para \; m = 1 \; ... \; M$$

Donde $\boldsymbol{m}$ es el número de antecedentes (en este caso piezas del diseño detectadas) y $\boldsymbol{Lm}$ el número de conjuntos difusos posibles para cada antecedente (en este caso dos, coincidencia y no-coincidencia). Entonces, para cada uno de los patrones, tendremos un conjunto de reglas teóricamente posibles semejante al siguiente:

si Pe_1 *coincide con* P_1 *y* Pe_2 *coincide con* P_1 *y ... y* Pe_m *coincide con* P_1
si Pe_1 *coincide con* P_1 *y* Pe_2 *coincide con* P_1 *y ... y* Pe_m *no coincide* P_1
entonces "P_1*" es* Pe_1 *con gardo de pertenencia gp en le intervalo 0,1.*

Hay que tener en cuenta que este conjunto de reglas variará en función del número de antecedentes, es decir, del número de piezas del diseño detectadas en el diseño incógnita bajo estudio.

Por otra parte, de este conjunto de reglas debemos eliminar las que a priori son absurdas o se crea que no pueden ocurrir observando el conjunto potencia y las combinaciones correspondientes de interés.

En este caso se ha establecido que, si se encuentra coincidencia entre una pieza de entrada y una de las piezas del diseño del patrón, se considerará que existe la posibilidad de que ese patrón forme parte del diseño incógnita, aunque con un grado de seguridad que variará en función del grado de coincidencia entre las piezas del diseño.

Por tanto, la única regla que debe ser eliminada es la última, en la que se afirma que el patrón forma parte de la entrada cuando en realidad ninguna de las piezas del

diseño de la entrada coincide con las del patrón y por tanto el grado de pertenencia ***(gp)*** en este caso no tiene sentido calcularlo.

Expresado en forma matricial y asociando el 1 a grado de pertenencia de la variable de entrada a la función coincidencia y el 0 a grado de pertenencia de la variable de entrada a la función no-coincidencia, el conjunto de reglas será:

Conjunto de reglas

$$A = \begin{pmatrix} 1 & 1 & ... & 1 \\ 1 & 1 & ... & 0 \\ 1 & 0 & ... & 1 \\ 1 & 0 & ... & 0 \\ & & ... & \\ & & ... & \\ 0 & 1 & ... & 1 \\ 0 & 1 & ... & 0 \\ 0 & 0 & ... & 1 \end{pmatrix}$$

4.4.4. Mecanismo de inferencia

El Mecanismo de Inferencia aclara y determina, cuáles son los operadores elegidos para la traducción matemática de las reglas que modela el identificador.

El conocimiento humano se expresa en términos de reglas difusas.

SI ENTONCES

SI proposición difusa ENTONCES proposición difusa.

Las proposiciones difusas pueden ser del tipo Atómicas y Compuestas.

Agregación.

Para realizar la agregación de las reglas, elegiremos entre las dos t-conormas más habituales; el operador máximo o el operador suma. En este caso, utilizaremos el operador máximo por simplicidad de cálculo.

Realizaremos la agregación del conjunto de reglas que tienen el mismo consecuente, es decir, en este caso el conjunto de reglas anteriormente propuesto para cada patrón.

Como ya se comentó la traducción matemática de las reglas se hace uso de las propiedades de conmutatividad de los operadores elegidos (Kosko, 1992) y, para simplificar el cálculo, se realiza en primer lugar la agregación de las reglas.

Se halla el máximo de los valores escalares resultantes de los antecedentes, que llamaremos λ_{pi} puesto que tendremos un valor de λ para cada patrón, y después se realiza la implicación.

$$\lambda_{pi} = max\{min\{\mu_{Ajk}\,(pei_j) : j = 1,\ ...,\ m\} :\ k = 1\,,\ ...\,,\ P_t\}$$

Donde A_{jk} representa en la regla k-ésima, el conjunto difuso asociado a la variable de entrada j-ésima: si $A_{jk} = 1$ el conjunto difuso asociado es el de coincidencia, y si A_{jk} =0 el conjunto difuso asociado es el de no-coincidencia.

El resultado de λ_{pi} es el que se modificará, según el operador elegido, la función característica del conjunto difuso de salida grado de pertenencia al realizar la implicación.

$$\mu_{gp} = prod\ \ ó\ \ min\{\lambda_{pi}\,,\ \mu_{gr}\} = \lambda_{pi}$$

El resultado de la inferencia será el obtenido de resolver el multi-antecedente y realizar la agregación.

Como resultado final de aplicar el conjunto de reglas, la función característica de salida propuesta μ_{gp} quedará modificada y convertida en μ'_{gp} tal y como se muestra en la (figura 4.24).

Finalmente se puede unir los resultados obtenidos para cada uno de los patrones como se muestra en la figura 4.25.

Estructura Final del Sistema Diseñado.

A modo de resumen se presentan a continuación los principales parámetros elegidos para su diseño.

1. Variables de entrada y salida:

2. Entrada: posiciones frecuenciales de las piezas del diseño detectadas en el diseño incógnita.

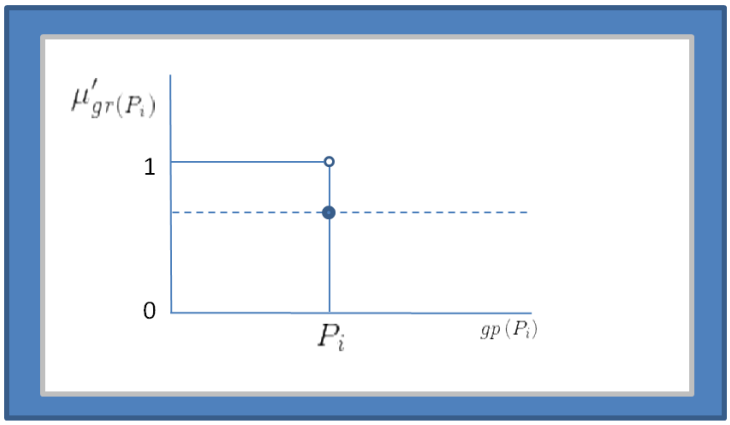


Figura 4.24: Función característica resultante a la salida de cada patrón.

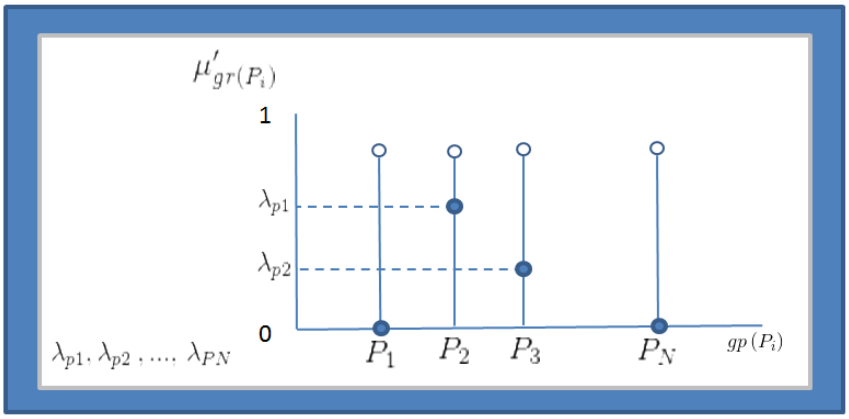


Figura 4.25: Función característica completa resultante a la salida.

3. Salida: grado de seguridad de la correspondencia del patrón con el diseño incógnita.

4. Funciones características:

5. Número: - entrada: 2 funciones características para la variable de entrada coincidencia y no-coincidencia de las piezas del diseño de la entrada a cada patrón.

6. Salida: función característica para la variable de salida grado de pertenencia.

7. Forma: entrada: función Lambda o Triangular (a,m, b)

8. Salida: singleton ideal, grados de pertenecia superior al 0.5 considerado como el patrón identificado.

9. Reglas que contribuyen a la identificación de los Patrones Gof presentes en el diseño incógnita bajo estudio:

   Conjunto de reglas = [1 11, 1 10, 1 0 1, 1 0 0, 0 1 1, 0 1 0, 0 0 1].

   Regla no utilizada = [0 0 0].

10. Mecanismo de inferencia: la traducción matemática de las reglas se realiza mediante los siguientes operadores matemáticos:

    a) El operador lógico y entre antecedentes: se traduce al operador mín.

    b) La implicación lógica que en este caso es entonces: es indiferente ya que lo que hace es conservar el valor ya obtenido.

    c) La agregación de las reglas: se realiza mediante el operador max.

    d) Defusificador: en este caso no es necesario realizar desdifusión ya que el conjunto difuso de salida está formado por un solo punto.

El diagrama del proceso de identificación de patrones en un sistema incógnita es el que se presenta a continuación (figura 4.26).

4.4.5. Comparación de grado de similitud entre atributos de diseño.

Para aplicará el modelo de inferencia y establecer el grado de similitud entre los diseños, Modelo del sistema incógnita y Factory patrones Gof, con atributos representados en conjuntos difusos triangulares, con los parámetros indicados en la tabla 4.1.

	C	I	M	A
Modelo	2	2	6	2
Factory	2	3	4	0

Cuadro 4.1: Parámetros de entrada.

Para ello, es necesario determinar algunas funciones o tareas:

Para evaluar la propuesta, se probará que al aplicar el modelo diseñado para obtener el grado de similitud que existe entre dos objetos A y B, cuyos atributos son difusos, se cumplen ciertas condiciones que garanticen soluciones apropiadas. Entre estas condiciones están las siguientes:

Determinación de etiquetas lingüísticas

1. Sean A y B dos objetos con atributos ***muy diferentes*** (con atributos muy distanciados en el universo de discurso) y $S_{gp}(A, B)$ el grado de similitud que hay entre ellos. Debe producir un grado de similitud global igual a cero (0) o muy cercano a cero, es decir:

$$S_{gp}(A, B) = 0 \text{ muy cercano a } 0.$$

2. Sean A y B dos objetos con atributos ***poco similares*** (con atributos bastante distanciados en el universo de discurso) y $S_{gp}(A, B)$ el grado de similitud que hay entre ellos. Debe cumplirse que:

$$0 < S_{gp}(A, B) < 0{,}5$$

3. Sean A y B dos objetos con atributos ***bastante similares*** (con atributos poco distanciados en el universo de discurso) y $S_{gp}(A, B)$ el grado de similitud que hay entre ellos. Debe cumplirse que:

$$0 < S_{gp}(A, B) < 1$$

4. Sean A y B dos objetos con atributos ***exactamente iguales*** y $S_{gp}(A, B)$ el grado de similitud que hay entre ellos, debe producir un grado de similitud global igual a 1, es decir:

$$0 < S_{gp}(A, B) = 1$$

5. Sean A y B dos objetos con atributos ***diferentes*** y $S_{gp}(A, B)$ el grado de similitud que hay entre ellos. Debe cumplirse que $S_{gp}(A, B) = S_{gp}(B, A)$. Esto lo podemos denominar condición de simetría.

Para realizar la evaluación de la propuesta se tomaron en cuenta los casos de estudio, utilizados por (Sequera, 2010) en su investigación con (Bashon *et al.*, 2011).

Determinación del universo del discurso.

Este caso trata con descripciones (más o menos aproximadas) de sitemas de software y patrones creacionales a detectar. Los atributos involucrados son la Clases, Interfaces, Métodos y Atributos. Los datos disponibles pueden ser representados en el cuadro 6.2 (el cual se reproduce a continuación, para facilitar su visualización)

	Cases	Interfaces	Métodos	Atributos
Modelo	muy diferentes	muy diferentes	muy diferentes	muy diferentes
Vista	poco similares	poco similares	poco similares	poco similares
Control	bastante similares	bastante similares	bastante similares	bastante similares
Dao	exactamente iguales	exactamente iguales	exactamente iguales	exactamente iguales
Persistencia	diferentes	diferentes	diferentes	diferentes

Cuadro 4.2: Etiquetas lingüísticas para determinar similitud de atributos.

Un dominio está conectado a cada atributo; este dominio es el conjunto de todos los valores que pueden ser tomados por el atributo. Por lo tanto, el dominio de cada uno de los atributos presentados por (Sequera, 2010) se determinó con los siguientes valores: Clase C, U= [0,2], Interfaz I, U= [2,3], Método M, U= [3,4], Atributo A, U = [-2,0].

Determinación de Funciones de Pertenencia. Se utilizará para esta evaluación funciones de pertenencia triangulares o Lambda que podrán representar todos los atributos difusos mostrados en el cuadro 6.3. Partiendo del universo del discurso señalado arriba, se estiman los valores para las diferentes etiquetas lingüísticas presentadas.

Por ejemplo, para el atributo Clase se tiene U = [0,3] entonces, para la etiqueta clase muy parecida se dice que la función de pertenencia asociada incluye el valor más alto dentro del universo del discurso quedando: Muy parecida (1.4,1.6,1.8,2), parecida con valores un poco menores a Muy parecida, Parecida = (0.8,1,1.2,1.4), más bien parecida = (1.8,2,2.2,2.4), entre otros.

Para el atributo Interfaz del diseño con U= [2,4] se tienen algunas etiquetas lingüísticas como muy Parecida, parecida y relativamente parecida. Para decir que una interfaz es parecida puede estimarse un máximo cuantitativo de 3 relaciones

	C	I	M	A
	2	2	6	2
Modelo	Parecida	Poco parecida	Relativamente parecida	Poco parecida
	(1.8, 2, 2.2, 2.4, 2.8)	(0, 1, 2, 2.2)	(3.6, 4, 4.2, 4.4)	(0.5,0.8,1,2)
	2	3	4	0
Factory	Muy parecida	Muy parecida	Parecida	Relativamente parecida
	(1.4, 1.6, 1.8, 2)	(2.6, 2.8, 3, 3.2)	(3.4, 3.6, 3.8, 4)	(0,0,0.5,0.8)

Cuadro 4.3: Estimación de valores de etiquetas lingüísticas.

en ese caso muy Parecida = (2.6, 2.8, 3, 3.2), para parecida = (2.2, 2.4, 2.6, 2.8) poco parecida que incluye al mínimo valor del universo del discurso poco parecida = (0,1,2,2.2). En el cuadro 6.1 se muestran los valores asignados a los distintos atributos de Modelo y Factory.

Partiendo del cuadro 6.3, un diseñador puede opinar que las etiquetas lingüísticas parecida y Muy parecida de la variable clase del diseño son bastante similares; las etiquetas lingüísticas poco parecida y muy parecida de la variable interfaz son poco similares, las etiquetas poco parecida y relativamente parecida son muy diferentes y las etiquetas relativamente parecida y parecida de la variable Método son bastante similares.

De lo anterior, el observador pudiera establecer a priori que para este caso, se considera que los objetos son poco similares, es decir que $0 <$ Sg(Modelo, Factory) < 0.5. Como una primera parte de la comprobación, en el presente trabajo se utilizarán los datos del cuadro 6.3 determinado las funciones o tareas siguientes:

1. Universo de discurso normalizado de los datos, en caso de ser necesario.

2. Comparación entre valores lingüísticos de los 4 atributos, utilizando medidas de similitud.

3. Ponderación global de comparaciones entre atributos entre Modelo y Factory.

Normalización de los atributos. El cuadro 6.4 muestra los atributos para Modelo y Factory Normalizados.

	C	I	M	A
Modelo	2	2	6	2
Variable L	Más bien parecida	Poco parecida	Relativamente parecida	Poco parecida
Normalizado	(0.6, 0.6, 0.7, 0.8)	(0, 0, 0, 0.1)	(0.3, 0.5, 0.6, 0.7)	(0.62, 0.7, 0.75, 1)
Factory	2	3	4	0
Variable L	Muy parecida	Muy Parecida	Parecida	Relativamente parecida
Normalizado	(0.46, 0.53, 0.6, 0.66)	(0.3, 0.4, 0.5, 0.6)	(0.2, 0.3, 0.4, 0.5)	(0.5, 0.5, 0.62, 0.7)

Cuadro 4.4: Atributos Normalizados de los conjuntos Modelo y Factory.

En el caso de las medidas basadas en distancia, es necesario previo a aplicar la medida de similitud, el proceso de normalización de los datos, por consiguiente se tomarán los datos mostrados en el cuadro 6.4. Utilizando la Distancia Euclídea como medida de similitud aplicamos la siguiente ecuación:

$$d\left(a,b\right)=1-\sqrt{\sum_{j=1}^{J}\left(a_j-b_j\right)^2}$$

Atributos	C	I	M	A	Similiud
Objeto	*s(a1,b1)*	*s(a2,b2)*	*s(a3,b3)*	*s(a4,b4)*	S(Modelo, Factory)
d=(a,b)	0.77	0.13	0.63	0.59	0.53

Cuadro 4.5: Distancia Euclídea como medida de similitud entre Modelo y Factory.

Comprobación de resultados. El grado de similitud global utilizando la distancia Euclídea es de 0.53.

Si el diseñador estableció a priori que para este caso, los objetos Modelo y Factory son poco similares, es decir que ***0 <Sg(Modelo, Factory) < 0.5***, implica que la medida de distancia Euclídea no logra resultados satisfactorios, más si es lo contrario hay una cierta similitud entre ambos componentes.

De igual manera se pondrá calcular la similitud del resto de los atributos.

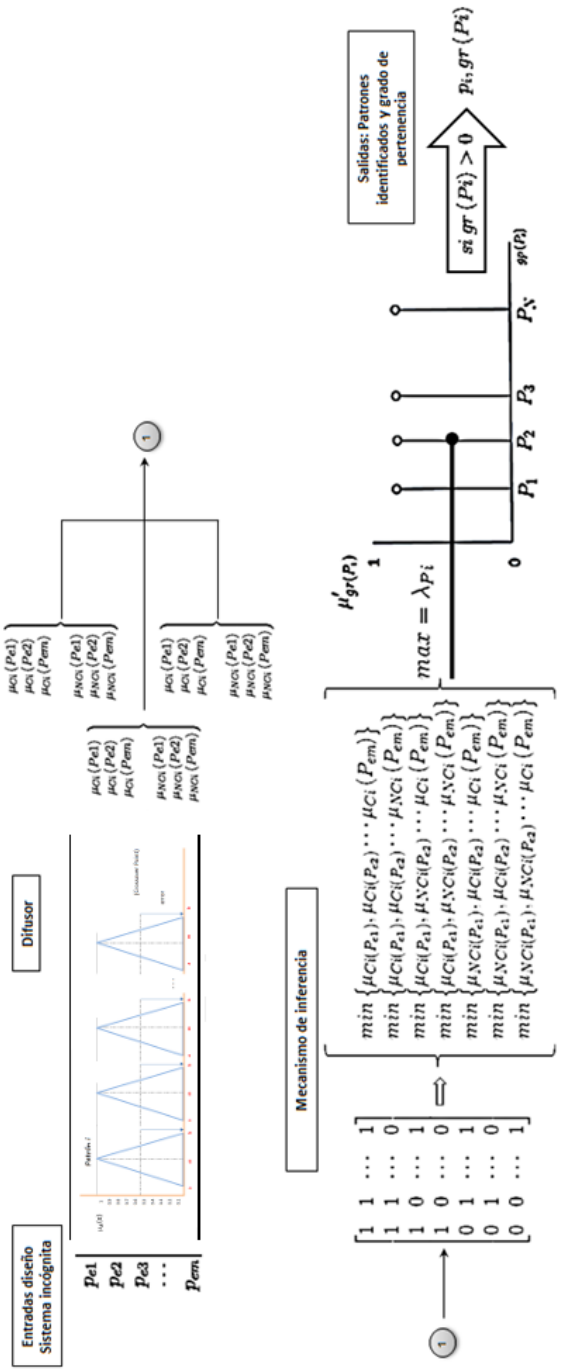


Figura 4.26: Diagrama de bloques del sistema.

Capítulo 5

Caso de estudio

5.1. Análisis

5.1.1. Análisis de componentes de patrón

Para la implementación y prueba en el proceso de identificación de patrones de diseño de software creacionales, desarrollados bajo el paradigma orientado a objetos clasificados y organizados en el catálogo de patrones conocido como de Gan of Four, representados y desarrollados en ésta investigación sobre el lenguaje de programación Java.

Se realizó, como primer paso en esta investigación, el análisis de componentes del catálogo de patrones clasificados como creacionales, análisis sobre esquemas de clases UML como se muestra en el Apéndice C, separando y clasificando las características de los patrones creacionales en piezas de diseño, (Métodos y Atributos) y piezas de patrón (Clases e Interfaces), entendiendo e interpretando la estructura de patrones de diseño GoF Figura 4.14, representado de manera cuantitativa, continuando con análisis cuantitativo haciendo la suma de cada componente que conforma el patrón, sobre la matriz binaria relación por cada patrón creacional, conformando así, la base de datos del conocimiento Apéndice B sobre la cual se hace el proceso de identificación de patrones, en la investigación, se realiza el análisis de componentes del sistema incógnita, sobre esquemas de clases UML Figura 5.1, separando y clasificando las características del sistema en piezas de diseño y piezas de patrón, que proporciona el

análisis de bajo nivel (código fuente) en clases interfaces métodos y atributos, de cada meta nivel de la estructura del sistema a analizar, representado cuantitativamente en la matriz binaria relación del sistema incógnita figura 5.2, obteniendo de el contexto en cual gira dicho sistema y que representa el universo de discurso cuantitativo. Durante el análisis del sistema incógnita, se observa que en el diseño, existe una meta capa que encapsulan a los posibles patrones creacionales a identificar, meta capa que clasifica la estructura compuesta del sistema, la cual se denomina como parte *Modelo, Vista, Controlador, Dao, Persistencia,* para este caso de estudio, en los cuales por cada uno de ellos se obtienen las piezas de diseño características de la estructura del sistema analizado *(Clases, Interfaces, Métodos, Atributos)*, implementado en cada meta nivel que conforma el sistema analizado.

5.1.2. Representación de características del universo de discurso

Para cada meta capa y sus características que establecen el universo del discurso, en el contexto en el que se extiende el sistema incógnita, se representan dichas características sobre una matriz binaria relación Figura 5.2 en el cual y con el fin de identificar a los patrones creacionales conocidos como *Factory, Singleton, Abstarct Factory, Builder, Prototype,* y sus características *(Clases, Interfaces, Métodos, Atributos)*, que los representan en piezas de diseño y piezas de patrón, base de conocimiento, y que se buscan dentro del sistema incógnita, identificando al patrón creacional en el cual se estructura la parte analizada de dicho sistema incógnita, identificación que podría ser hecha por el experto diseñador basado en la interpretación del análisis que le brinda la experiencia de tiempo, en el diseño sistemas de software.

Para esta búsqueda se hace uso de los procedimientos y métodos de solución de la lógica difusa rama de la Inteligencia artificial.

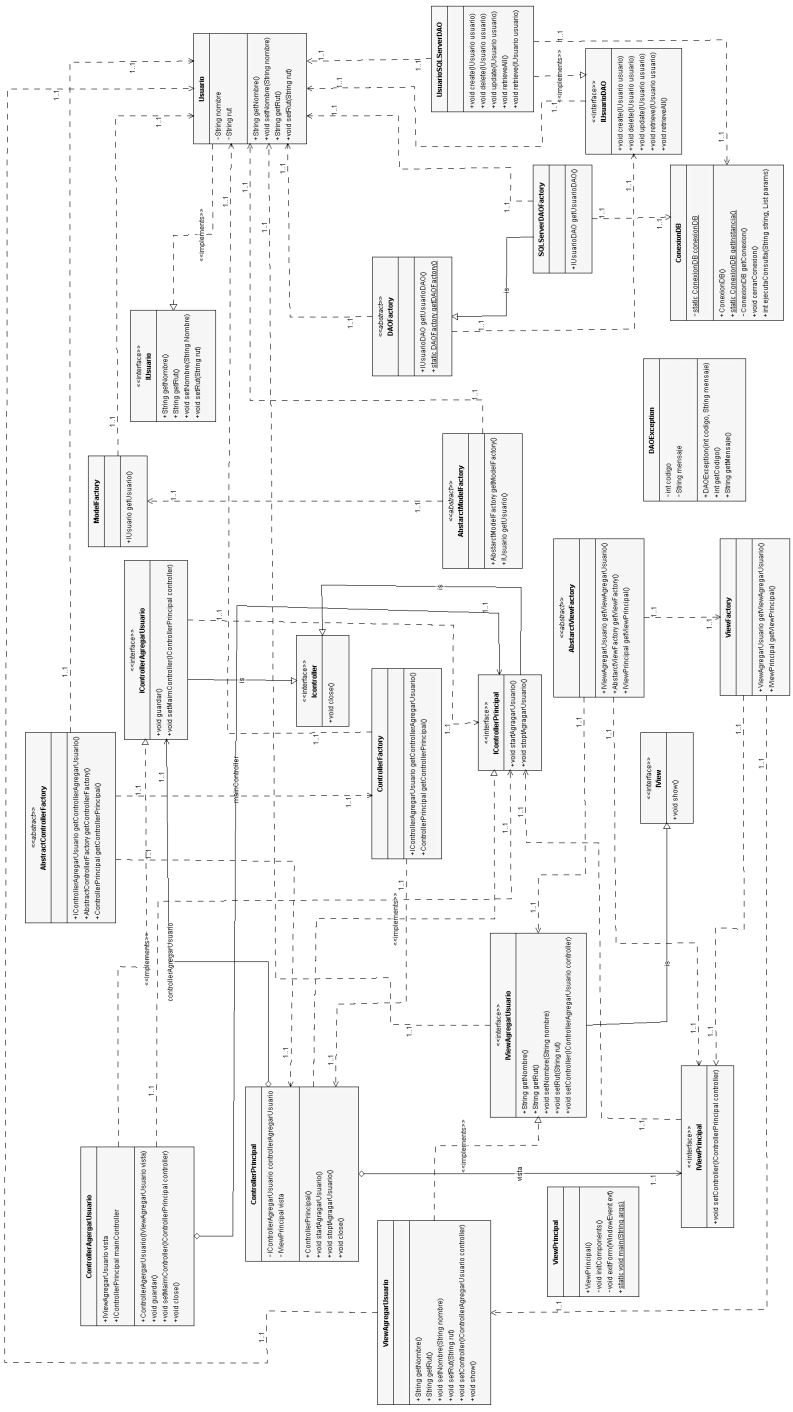


Figura 5.1: Diagrama de clases sistema analizado

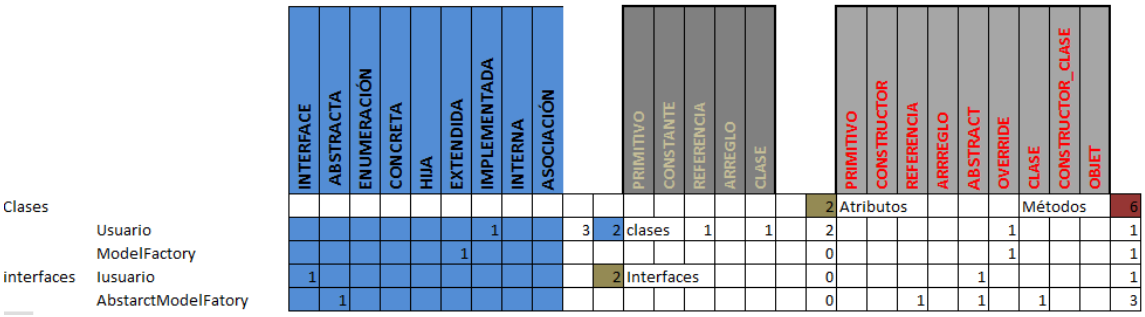


Figura 5.2: Matriz binaria relación sistema incógnita

5.2. Procedimiento difuso

5.2.1. Procedimiento de normalización

Representando estas características de diseño tanto de la base de conocimiento creacional como del sistema incógnita, sistema analizado en conjuntos difusos, Figura 5.3 y Figura 5.4 haciendo uso de la función característica denominada como Lambda o piramidal con la cual normalizamos o transformamos los números concretos a difusos.

Función Lambda:

$$\mu_A(x) = \begin{cases} 0 & si \quad x \leq a \\ \frac{(x-a)}{(m-a)} & si \quad a < x < m \\ \frac{(b-x)}{(b-m)} & si \quad m < x \leq b \\ 0 & si \quad x > b \end{cases}$$

Para esta demostación tenemos a la parte del sistema incógnita denominada Modelo se busca el patrón creacional Factory representados por sus características cuantitativas, así mismo se busca el patróon Factory, la meta capa Modelo del sistema incógnita a esta búsqueda se le denomina por derecha, y por izquierda. Para identificar la existencia del patrón creacional Factory sobre el sistema incógnita, meta capa Modelo, se representa de forma cuantitativa dichas características Cuadro. 5.1

Los resultados obtenidos en el proceso de normalización pertenencia del patrón Factory (x) en el modelo (y) son:

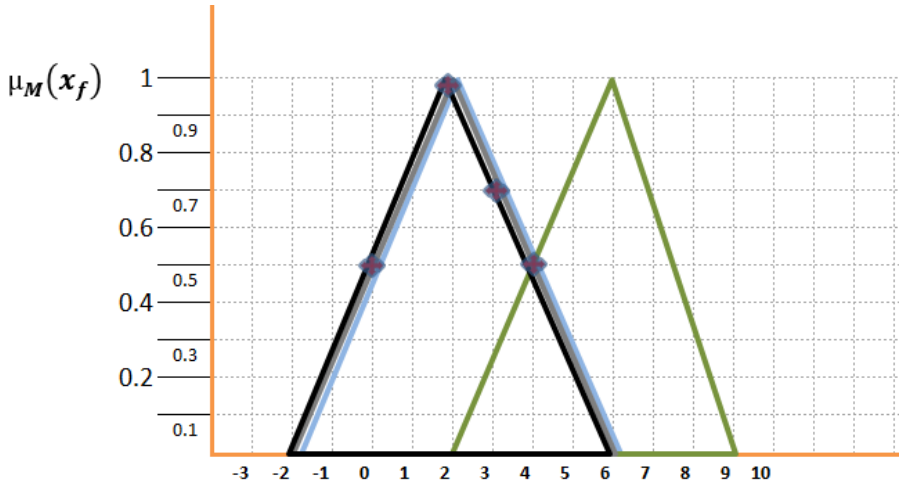


Figura 5.3: Conjunto Difuso centrado en el Modelo

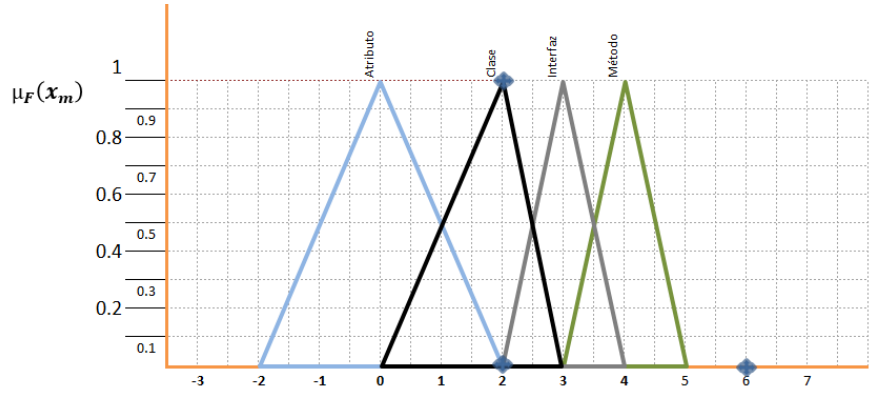


Figura 5.4: Conjunto difuso centrado en el patrón factory

Derecha

$$\mu_M(x_f) = \left\{ \frac{1}{2} + \frac{0{,}75}{3} + \frac{0{,}5}{4} + \frac{0{,}5}{0} \right\}$$

Los resultados obtenidos en el proceso de normalización pertenencia del modelo(x) sobre el patrón Factory (y) son:

Izquierda

$$\mu_F(x_m) = \left\{ \frac{1}{2} + \frac{0}{2} + \frac{0}{6} + \frac{0}{2} \right\}$$

De esta manera se puede observar el proceso de fusificación, para la identificación de los patrones creacionales sobre el sistema incógnita, según el esquema de control

	Clases	Interfaces	Métodos	Atributos
Modelo	2	2	6	2
Factory	2	3	4	0

Cuadro 5.1: Cuadro cuantitativo característico, Modelo Factory.

difuso mencionado anteriormente y sobre el cual se basa la presente investigación.

5.2.2. Proceso de relación y comparación

La siguiente tarea en el proceso de identificación será obtener la relación entre ambos conjuntos difusos realizando el producto cartesiano difuso o producto cruce R(AxB), con esto obtenemos todo el universo en el cual se presume una posible identificación del patrón creacional.

Considerando como A =$\mu_F(x_m)$ y B = $\mu_M(x_f)$.

Como en la teoría clásica, para relacionar dos conjuntos efectuaremos el producto cartesiano de sus elementos.

Como siempre, la diferencia residirá en que mientras en la teoría clásica, los elementos de los conjuntos están o no en relación, para el caso de una relación difusa cada par ordenado de los dos elementos puede gozar de una función de pertenencia que caracterice en qu grado dichos elementos están relacionados.

Para este caso se relaciona dos conjuntos difusos entre sí según una premisa, por ejemplo, la relación difusa R = "X aproximadamente igual que Y", y que tiene la estructura matricial: figura 5.5

Donde se expresa el grado en que los elementos de ambos conjuntos están relacionados entre sí.

El operador difuso que da resultado a esta operación es el operados mínimo difuso entre A=$\mu_F(x_m)$ y B = $\mu_M(x_f)$.

Como se puede observar, el pico de la función de pertenencia se encuentra centrado en aquellas posiciones tales, que los dos elementos son iguales, expresando así, que los elementos se encuentran relacionados al máximo (función de pertenencia = 1) cuando son iguales.

Producto cartesiano denotado po R=(AxB)

$$R=(AxB)\begin{vmatrix} \frac{1}{2,2}+\frac{0.75}{(2,3)}+\frac{0.5}{(2,4)}+\frac{0.5}{(2,0)} \\ \frac{0}{(2,2)}+\frac{0}{(2,3)}+\frac{0}{(2,4)}+\frac{0}{(2,0)} \\ \frac{0}{(2,2)}+\frac{0}{(2,3)}+\frac{0}{(2,4)}+\frac{0}{(2,0)} \\ \frac{0}{(2,2)}+\frac{0}{(2,3)}+\frac{0}{(2,4)}+\frac{0}{(2,0)} \end{vmatrix}$$

Figura 5.5: Relación difusa R = "X aproximadamente igual que Y"
R=(Sistema incógnita, Base de patrones)

Una ves obtenida la relación $\boldsymbol{R}$ del producto cartesiano, que contiene todas las características de ambos diseños, que dando la matriz R como sigue: figura 5.6

Para inferir resultados dentro del proceso difuso se generaliza el proceso en la identificación, usando los métodos y procedimientos difusos como la proyección, extensión cilíndrica y composición difusa, proponiendo para esta investigación, a la regla composicional de inferencia expresada como B=A$\circ R_{(AxB)}$, para comparar y comprobar de manera matemática los resultados de identificación de patrones de

Producto cartesiano difuso

R=(AxB)

Diseño Incógnita \ Patrón Factory	2	3	4	0	
2	1	0.75	0.5	0.5	C
2	0	0	0	0	I
6	0	0	0	0	M
2	0	0	0	0	A
	C	I	M	A	

Figura 5.6: Matriz resultado del producto (Modelo, Factory)

software creacionales.

La siguiente tarea es realizar el proceso de proyección que consigue reducir una relación (dos dimensiones) a un conjunto difuso (una sola dimensión).

En realidad lo que hace la proyección es, para cada valor de y, buscar la máxima en Mr(x,y) respecto de las x, donde Mr(x,y) es la función de pertenencia de la relación.

$$P_y = \mu_A(x) \cap \mu_R(AxB)$$

Matricialmente, se expresa Py(R) donde R es "$\boldsymbol{x}$ aproximadamente igual que $\boldsymbol{y}$".

La proyección de la relación R sobre Y se define como:

$$P_y(R) = \int sup(x) \left(\frac{\mu_R(x,y)}{y}\right)$$

Py(R)=	1, 0.75, 0.5, 0.5

Como podemos observar en la búsqueda por la derecha de la proyección entre ambos conjuntos se tienen características cualitativas, es decir características aproximadamente iguales del patrón factory a la meta capa modelo.

Para comprobar este proceso se realiza el cálculo de la extensión.

La extensión constituye el paso inverso a la proyección:

A partir de un conjunto difuso obtenemos una relación y se define por:

$$EC_{(A)} = \int_{x \cdot y}^{n} \frac{\mu_A(x)}{(x,y)}$$

La extensión consiste en asignar el valor de la función de pertenencia, de $\boldsymbol{x}$ a todas sus tuplas en $\boldsymbol{y}$. Es decir, dado el conjunto difuso discreto $\boldsymbol{x}$ definido por:

Izquierda

$$\mu_F(x_m) = \left\{\frac{1}{2} \quad + \quad \frac{0}{2} \quad + \quad \frac{0}{6} \quad + \frac{0}{2}\right\}$$

(Nótese que esta definición se corresponde a una función de pertenencia discreta del tipo LAMBDA) y el universo de discurso Y (2,2,6,2), al efectuar CE(X), tenemos a la extensión cilndrica $''x''$ sobre $''y''$ 5.7

Como puede verse, para extender el conjunto difuso $\boldsymbol{x}$ hacia el universo $\boldsymbol{y}$, basta con hacer el producto cartesiano de ambos universos y asignar, para cada tupla, un valor de $\boldsymbol{y}$ igual al valor de su $\boldsymbol{x}$ correspondiente en la tupla. Intersección entre R y EC 5.8

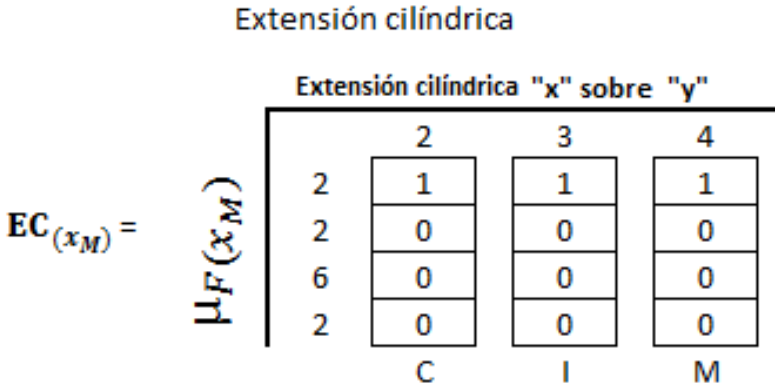


Figura 5.7: Extensión cilíndrica (Modelo, Factory)

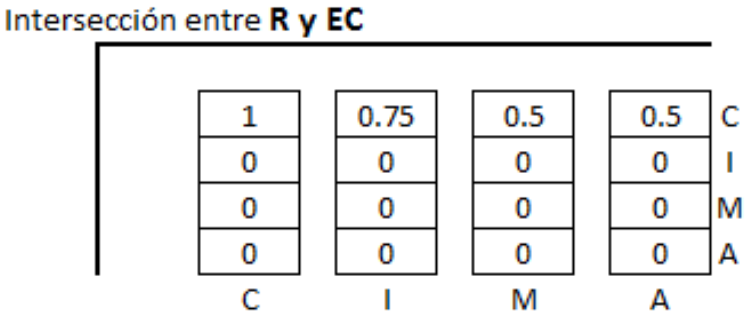


Figura 5.8: Intersección entre Producto cartesiano y Extensión cilíndrica (R, EC)

5.3. Comparación y resultados

5.3.1. Conclusión por el mínimo y el máximo difuso

El siguiente paso es efectuar la composición, para esta investigación proponemos la ejecución de la regla composicional de inferencia para deducir los resultados de identificación.

Esta operación se efectúa entre un conjunto difuso y una relación, dando como resultado un conjunto difuso nuevo.

Por ejemplo, dados el conjunto difuso A y la relación R definidas por:

$$A = \int \frac{\mu(x)}{x}$$

$$R = \int \frac{\mu_R(x,y)}{(x,y)}$$

La composición se efectúa por:

$$B = A \circ R = P_y \left(EC \left(A \right) \cap R \right)$$

Lo que expresa la composición, son todas las relaciones posibles entre los elementos de A y B (es decir, la relación R), la extensión de A (intersección por el mínimo), se proyecta hacia B., R representa todas las asociaciones posibles de elementos de A con los de B, concluyendo en una relación del grupo de elementos de B, con el conjunto A, expresada (según la relación definida) en grado de pertenencia de cada elemento de esta relación, (por eso nos queda un conjunto difuso como resultante).

La capacidad de la composición para concluir nos será muy útil cuando tratemos de realizar inferencias con proposiciones condicionales., para efectuar la composición de A con R, el primer paso es extender cilíndricamente A.

Seguidamente, en la fórmula de la composición leemos que debemos hacer el mínimo de las dos relaciones obtenidas (la que relaciona un universo con otro (extensión cilíndrica que acabamos de efectuar).

Intersectando A con R por la T-norm del mnimo (efectuamos el mínimo de ambas tuplas) obteniendo lo siguiente: Intersección entre R y EC figura 5.9

Proyectando ahora sobre el universo de patrón Factory, es decir, con el máximo de cada columna según la definición de proyección, tenemos: Proyección sobre el universo Factory figura 5.10

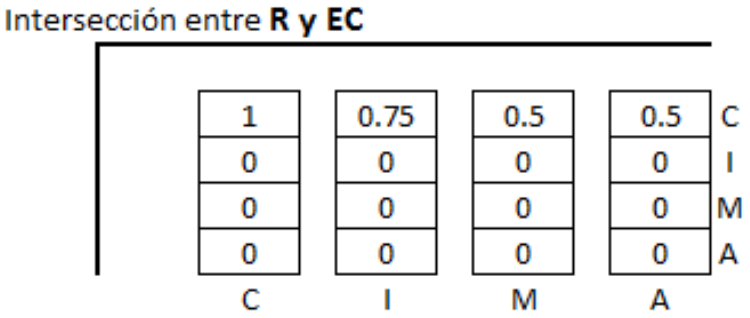


Figura 5.9: Intersectando A con R por la T-norm del mínimo

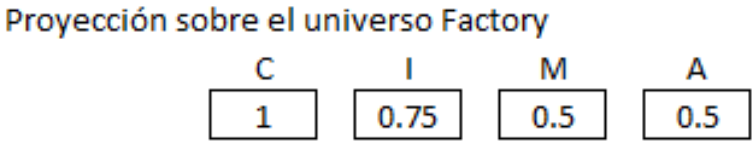


Figura 5.10: Proyectando el máximo sobre el universo Factory

Posteriormente se realizan operaciones de números difusos uniendo y cruzando las características del patrón resultante obteniendo el grado real de pertenencia este proceso obteniendo el siguiente resultado figura 5.11

5.3.2. Comparación de resultados

Identificando así el grado de pertenecía del la meta capa modelo al patrón creacional Factory, procediendo a comparar resultados y de deducir con el experto didseñador así como utilizando la Distancia Euclídea como medida de similitud.6.6

Metodología propuesta	Distancia Euclídea	Diseñador
0.5	0.52	Poco parecido

Cuadro 5.2: Comparación de resultados.

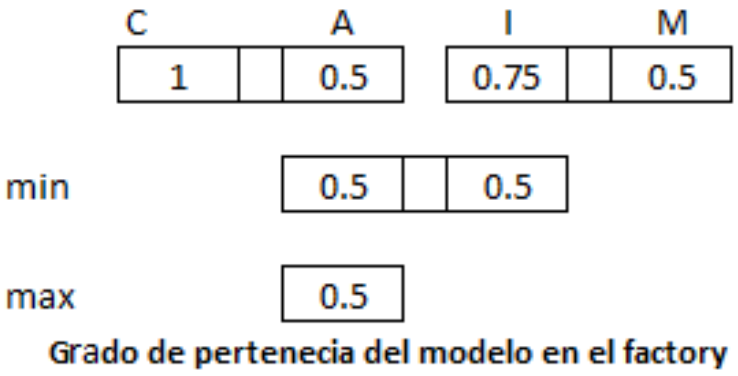


Figura 5.11: Unión, intersección de números difusos
Resultado características del patrón

Capítulo 6

Resultados

6.1. Introducción

En la presente sección se realizó la comparación entre la técnica propuesta, con la técnica de similitud, denomina medida de similitud geométrica, donde se hace uso de la propuesta de distancia de **Minkowski** expuesta en el trabajo de Berzal (Galiano, 2002), mencionada en capítulos anteriores, comparando el resultado entre ambas técnicas obteniendo resultados satisfactorios o similares para el mismo caso de identificación del capítulo anterior.

6.1.1. Medidas de comparación

Zadeh (1971), define la noción de similitud (similarity) como la generalización de la noción de equivalencia. Los conjuntos difusos están relacionados con las mediciones de similitudes y disimilitudes entre objetos por su capacidad de representar información subjetiva resultante de la complejidad del mundo real. tversky1982similarity presentó una definición formal de medidas de similitud como una combinación lineal de las medidas de sus características comunes y distintivas. Bouchon-Meunier, Rifqi y Bothorel (Bouchon-Meunier *et al.*, 1996, pág 123), proponen una definición general de medidas de comparación que involucran descripciones difusas de características, con la intención de enfocar la flexibilidad del concepto de similitud en el marco de conjuntos difusos, y extender la definición original de Zadeh (Zadeh, 1971) sobre

similitud. Rifqi, Berger y Bouchon-Meunier (Bustince, 2000) presentaron un método para escoger medidas de comparación entre dos objetos, basado en el poder de discriminación de una medida. De esta forma, las medidas pueden ser comparadas entre ellas según su comportamiento. Las medidas de similitud se distinguen en dos clases: la geométrica y la teoría de conjuntos. Los modelos de distancia geométrica es el enfoque más comúnmente utilizado. Objetos que deben compararse son considerados como puntos en un espacio métrico. Estos modelos están limitados por 4 propiedades que la distancia tiene que satisfacer, la positividad, la simetría, minimalidad y la desigualdad triangular (Tversky & Gati, 1982).

6.1.2. Comprobación de grado de similitud geométrica.

A continuación se aplicará el modelo de inferencia para establecer el grado de similitud entre los diseños, Modelo y Factory los cuales tienen atributos, representados mediante conjuntos difusos triangulares, cuyos parámetros se indican en la tabla 6.1.

	C	I	M	A
Modelo	2	2	6	2
Factory	2	3	4	0

Cuadro 6.1: Parámetros de entrada.

Para ello, es necesario determinar algunas funciones o tareas:

Para evaluar la propuesta, se probará que al aplicar el modelo diseñado para obtener el grado de similitud que existe entre dos objetos A y B, cuyos atributos son difusos, se cumplen ciertas condiciones que garanticen soluciones apropiadas. Entre estas condiciones están las siguientes:

1. ***Determinación de etiquetas lingüísticas***

   a) Sean A y B dos objetos con atributos ***muy diferentes*** (con atributos muy distanciados en el universo de discurso) y $S_{gp}(A, B)$ el grado de

similitud que hay entre ellos. Debe producir un grado de similitud global igual a cero (0) o muy cercano a cero, es decir:

$$S_{gp}(A, B) = 0 \text{ muy cercano a } 0.$$

b) Sean A y B dos objetos con atributos ***poco similares*** (con atributos bastante distanciados en el universo de discurso) y $S_{gp}(A, B)$ el grado de similitud que hay entre ellos. Debe cumplirse que:

$$0 < S_{gp}(A, B) < 0{,}5$$

c) Sean A y B dos objetos con atributos ***bastante similares*** (con atributos poco distanciados en el universo de discurso) y $S_{gp}(A, B)$ el grado de similitud que hay entre ellos. Debe cumplirse que:

$$0 < S_{gp}(A, B) < 1$$

d) Sean A y B dos objetos con atributos ***exactamente iguales*** y $S_{gp}(A, B)$ el grado de similitud que hay entre ellos, debe producir un grado de similitud global igual a 1, es decir:

$$0 < S_{gp}(A, B) = 1$$

e) Sean A y B dos objetos con atributos ***diferentes*** y $S_{gp}(A, B)$ el grado de similitud que hay entre ellos. Debe cumplirse que $S_{gp}(A, B) = S_{gp}(B, A)$. Esto lo podemos denominar condición de simetría.

Para realizar la evaluación de la propuesta se tomaron en cuenta los casos de estudio, utilizados por (Sequera, 2010) en su investigación con (Bashon *et al.*, 2011, pág 13).

2. ***Determinación del universo del discurso.***

Este caso trata con descripciones (más o menos aproximadas) de sistemas de software y patrones creacionales a detectar. Los atributos involucrados son la Clases, Interfaces, Métodos y Atributos. Los datos disponibles pueden ser representados en el cuadro 6.2 (el cual se reproduce a continuación, para facilitar su visualización)

	Cases	Interfaces	Métodos	Atributos
Modelo	muy diferentes	muy diferentes	muy diferentes	muy diferentes
Vista	poco similares	poco similares	poco similares	poco similares
Control	bastante similares	bastante similares	bastante similares	bastante similares
Dao	exactamente iguales	exactamente iguales	exactamente iguales	exactamente iguales
Persistencia	diferentes	diferentes	diferentes	diferentes

Cuadro 6.2: Etiquetas lingüísticas para determinar similitud de atributos.

Un dominio está conectado a cada atributo; este dominio es el conjunto de todos los valores que pueden ser tomados por el atributo. Por lo tanto, el dominio de cada uno de los atributos presentados por (Sequera, 2010) se determinó con los siguientes valores: Clase C, U= [0,2], Interfaz I, U= [2,3], Método M, U= [3,4], Atributo A, U = [-2,0].

3. ***Determinación de Funciones de Pertenencia.*** Se utilizará para esta evaluación funciones de pertenencia triangulares o Lambda que podrán representar todos los atributos difusos mostrados en el cuadro 6.3. Partiendo del universo del discurso señalado arriba, se estiman los valores para las diferentes etiquetas lingüísticas presentadas.

	C	I	M	A
	2	2	6	2
Modelo	Parecida	Poco parecida	Relativamente parecida	Poco parecida
	(1.8, 2, 2.2, 2.4, 2.8)	(0, 1, 2, 2.2)	(3.6, 4, 4.2, 4.4)	(0.5,0.8,1,2)
	2	3	4	0
Factory	Muy parecida	Muy parecida	Parecida	Relativamente parecida
	(1.4, 1.6, 1.8, 2)	(2.6, 2.8, 3, 3.2)	(3.4, 3.6, 3.8, 4)	(0,0,0.5,0.8)

Cuadro 6.3: Estimación de valores de etiquetas lingüísticas.

Por ejemplo, para el atributo Clase se tiene U = [0,3] entonces, para la etiqueta clase muy parecida se dice que la función de pertenencia asociada incluye el valor más alto dentro del universo del discurso quedando: Muy parecida (1.4,1.6,1.8,2), parecida con valores un poco menores a Muy parecida, Parecida = (0.8,1,1.2,1.4), más bien parecida = (1.8,2,2.2,2.4), entre otros.

Para el atributo Interfaz del diseño con U= [2,4] se tienen algunas etiquetas lingüísticas como muy Parecida, parecida y relativamente parecida. Para decir que una interfaz es parecida puede estimarse un máximo cuantitativo de 3 relaciones en ese caso muy Parecida = (2.6, 2.8, 3, 3.2), para parecida = (2.2, 2.4, 2.6, 2.8) poco parecida que incluye al mínimo valor del universo del discurso poco parecida = (0,1,2,2.2). En el cuadro 6.1 se muestran los valores asignados a los distintos atributos de Modelo y Factory.

Partiendo del cuadro 6.3, un diseñador puede opinar que las etiquetas lingüísticas parecida y Muy parecida de la variable Clase del diseño son bastante similares; las etiquetas lingüísticas poco parecida y muy Parecida de la variable interfaz son poco similares, las etiquetas poco parecida y relativamente parecida son muy diferentes y las etiquetas relativamente parecida y parecida de la variable Método son bastante similares.

De lo anterior, el observador pudiera establecer a priori que para este caso, se considera que los objetos son poco similares, es decir que $0 <$ Sg(Modelo, Factory) < 0.5. Como una primera parte de la comprobación, en el presente trabajo se utilizarán los datos del cuadro 6.3 determinado las funciones o tareas siguientes:

a) Universo de discurso normalizado de los datos, en caso de ser necesario.

b) Comparación entre valores lingüísticos de los 4 atributos, utilizando medidas de similitud.

c) Ponderación global de comparaciones entre atributos entre Modelo y Factory.

	C	I	M	A
Modelo	2	2	6	2
Variable L	Más bien parecida	Poco parecida	Relativamente parecida	Poco parecida
Normalizado	(0.6, 0.6, 0.7, 0.8)	(0, 0, 0, 0.1)	(0.3, 0.5, 0.6, 0.7)	(0.62, 0.7, 0.75, 1)
Factory	2	3	4	0
Variable L	Muy parecida	Muy Parecida	Parecida	Relativamente parecida
Normalizado	(0.46, 0.53, 0.6, 0.66)	(0.3, 0.4, 0.5, 0.6)	(0.2, 0.3, 0.4, 0.5)	(0.5, 0.5, 0.62, 0.7)

Cuadro 6.4: Atributos Normalizados de los conjuntos Modelo y Factory.

4. ***Normalización de los atributos.*** El cuadro 6.4 muestra los atributos para Modelo y Factory Normalizados.

   En el caso de las medidas basadas en distancia, es necesario previo a aplicar la medida de similitud, el proceso de normalización de los datos, por consiguiente se tomarán los datos mostrados en el cuadro 6.4. Utilizando la Distancia Euclídea como medida de similitud aplicamos la siguiente ecuación:

$$d\left(a,b\right)=1-\sqrt{\sum_{j=1}^{J}\left(a_j-b_j\right)^2}$$

Atributos	C	I	M	A	Similitud
Objeto	*s(a1,b1)*	*s(a2,b2)*	*s(a3,b3)*	*s(a4,b4)*	S(Modelo, Factory)
d=(a,b)	0.77	0.13	0.63	0.59	0.53

Cuadro 6.5: Distancia Euclídea como medida de similitud entre Modelo y Factory.

6.2. Comparación de resultados.

El grado de similitud global utilizando la distancia Euclídea es de 0.53.

Si el diseñador estableció a priori que para este caso, los objetos Modelo y Factory son poco similares, es decir que ***0 <Sg(Modelo, Factory)*** $<$ ***0.5***, implica que la medida de distancia Euclídea no logra resultados satisfactorios, más si es lo contrario hay una cierta similitud entre ambos componentes. Al realizar las operaciones con números difusos uniendo y intersectando las características del patrón resultante obteniendo el grado real de pertenencia con la metodología propuesta con resultado de 0.5 como se puede observar en el cuadro 6.6

Metodología propuesta	Distancia Euclídea	Diseñador
0.5	0.52	Poco parecido

Cuadro 6.6: Comparación de resultados.

De igual manera se pondrá calcular la similitud del resto de los atributos.

Capítulo 7

Conclusión

En la presente tesis se identificaron patrones de diseño de software clasificados como creacionales, con técnicas de inteligencia artificial (IA), en sistemas y aplicaciones de software orientadas a objetos concretamente bajo el lenguaje de programación java.

Centrados en el uso de catálogo GoF, existen catálogos de patrones para resolver problemas en distintos niveles de abstracción.

La idea es, que los diseñadores de software usen y conozcan patrones para entender mejor la abstracción en el desarrollo de un sistema de software.

La identificación de patrones de diseño es a menudo un factor de tiempo en el desarrollo del sistema de software, para arquitectos y desarrolladores sin o poca experiencia.

Con la ayuda de procedimientos y métodos de la lógica difusa, para identificar patrones presentes en sistemas o aplicaciones de software, se ayuda al desarrollador sin experiencia o al desarrollador experimentado a comprender de mejor manera la solución de problemas durante el desarrollo de sistemas, reducir la curva de aprendizaje en el desarrollo, se aporta experiencia, calidad y buenas prácticas durante el proceso de programación del sistema de software mediante un buen diseño bajo patrones probados que permiten la estandarización entre analistas, diseñadores y programadores, en cada uno de los niveles que conforma la arquitectura de software, es decir, comprender y reconocer soluciones a problemas similares que se presentan en contextos diferentes.

Por otra parte se comprueba que la lógica difusa permite la representación matemática, modelando funciones no lineales, que usa el razonamiento aproximado, similar al del ser humano. Razonamiento que haría el experto diseñador guiado por la experiencia de tiempo, planteado en el procedimiento matemático de la lógica difusa.

Con los planteamientos lógicos difusos se identifica la estructura y las relaciones entre los elementos de un sistema de software y los modela cuantitativa y cualitativamente, interpretando la estructura del sistema de software de bajo nivel matemáticamente, mejorando la estructura, calidad, buenas prácticas promoviendo la estandarización en el desarrollo orientado a objetos reconociendo a los patrones creacionales como la mejor herramienta para la solución a problemas recurrentes en el desarrollo de sistemas.

La Lógica difusa se basa en reglas que no tienen límites discretos, esto permite manejar mejor la ambigüedad, esto es muy útil para reflejar el cómo tiende a pensar el ser humano, en términos relativos, más no absolutos.

Cuando la lógica difusa se incorpora a un proceso ambiguo donde la semántica es similar en diferentes contextos, como la interpretación de un sistema de software modelado matemáticamente, el resultado es un sistema que interpreta mejor la manera natural en que un experto humano resolvería un problema.

7.1. Trabajos futuros.

Existen sistemas de software que por la forma en que están diseñados, si es que hay algún precedente de la fase de diseño en su desarrollo, ofrecen resultados forzados, es decir, que son sistemas no funcionales, ya que al ser puestos a prueba o exigidos en su ejecución tienden a colapsar, aun y cuando los resultados son los esperados por parte del sistema.

Con la presente propuesta de identificación de patrones creacionales clasificados en el catálogo del grupo de los cuatro, no solo se pretende ofrecer una alternativa estática para sistemas que se encuentren en la fase de prueba, sino también para sistemas que se encuentren actualmente en funcionamiento, es decir, de forma dinámica, donde con el análisis invasivo, con herramientas, como la api reflexión de java, el

paradigma orientado a aspectos y el análisis distribuido entre otros, así como el uso del modelado matemático con métodos y procedimientos de inteligencia artificial concretamente sobre métodos de la lógica difusa, que de lugar a el desarrollo futuro de la herramienta de software que automatice el proceso de identificación estático y dinámico sobre sistemas de software desarrollados en los conceptos de orientación a objetos.

Conceptos que se interpretan subjetivamente por el equipo de desarrollo y en los que se pretende, especialmente los diseñadores sean estandarizados, formando así, la comunidad estandarizada en el diseño y desarrollo de software orientado a objetos, no solo en el lenguaje de programación java, sino también en otros lenguajes que presuman soporten el paradigma o que sean multiparadigma, incluso en lenguajes de nueva generación.

Apéndice A

A.1. Código fuente en java, patrones creacionales.

Steps 1 Use the following steps to implement the above mentioned design pattern.

Patrón Factory

```
Step 1
Create an interface.
Shape.java

public interface Shape {
        void draw();
}

Step 2
Create concrete classes implementing the same interface.
Rectangle.java

public class Rectangle implements Shape {
@Override
    public void draw() {
    System.out.println("Inside Rectangle::draw() method.");
    }
}
```

Square.java

```
public class Square implements Shape {
@Override
        public void draw() {
                System.out.println("Inside Square::draw() method.");
}
```

Circle.java

```
public class Circle implements Shape {
@Override
        public void draw() {
        System.out.println("Inside Circle::draw() method.");
        }
}
```

Step 3
Create a Factory to generate object of concrete
class based on given information.

ShapeFactory.java

```
public class ShapeFactory {
//use getShape method to get object of type shape
        public Shape getShape(String shapeType){
                if(shapeType == null){
                return null;
        }
        if(shapeType.equalsIgnoreCase("CIRCLE")){
                return new Circle();
        } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
```

```
            return new Rectangle();
        } else if(shapeType.equalsIgnoreCase("SQUARE")){
            return new Square();
        }
            return null;
        }
}
```

Step 4
Use the Factory to get object of concrete
class by passing an information such as type.

FactoryPatternDemo.java

```
public class FactoryPatternDemo {
        public static void main(String[] args) {
        ShapeFactory shapeFactory = new ShapeFactory();
//get an object of Circle and call its draw method.
        Shape shape1 = shapeFactory.getShape("CIRCLE");
//call draw method of Circle
        shape1.draw();
//get an object of Rectangle and call its draw method.
        Shape shape2 = shapeFactory.getShape("RECTANGLE");
//call draw method of Rectangle
        shape2.draw();
//get an object of Square and call its draw method.
        Shape shape3 = shapeFactory.getShape("SQUARE");
//call draw method of circle
            shape3.draw();
        }
}
```

```
/*
        Step 5
                Verify the output.
                        Inside Circle::draw() method.
                        Inside Rectangle::draw() method.
                        Inside Square::draw() method.
*/
```

Patrón Abstract Factory

```
Steps
Use the following steps to implement the
above mentioned design pattern.

Step 1
Create an interface for Shapes.

Shape.java

public interface Shape {
        void draw();
}

Step 2
Create concrete classes implementing the same interface.

Rectangle.java

public class Rectangle implements Shape {
@Override
     public void draw() {
     System.out.println("Inside Rectangle::draw() method.");
```

```
    }
}
```

Square.java

```
public class Square implements Shape {
@Override
     public void draw() {
     System.out.println("Inside Square::draw() method.");
     }
}
```

Circle.java

```
public class Circle implements Shape {
@Override
     public void draw() {
     System.out.println("Inside Circle::draw() method.");
     }
}
```

Step 3
Create an interface for Colors.

Color.java

```
public interface Color {
void fill();
}
```

Step4
Create concrete classes implementing the same interface.

Red.java

```
public class Red implements Color {
@Override
     public void fill() {
     System.out.println("Inside Red::fill() method.");
     }
}
```

Green.java

```
public class Green implements Color {
@Override
     public void fill() {
     System.out.println("Inside Green::fill() method.");
     }
}
```

Blue.java

```
public class Blue implements Color {
@Override
     public void fill() {
     System.out.println("Inside Blue::fill() method.");
        }
}
```

```
Step 5
Create an Abstract class to get factories for
Color and Shape Objects.

AbstractFactory.java

public abstract class AbstractFactory {
        abstract Color getColor(String color);
        abstract Shape getShape(String shape) ;
}

Step 6
Create Factory classes extending AbstractFactory to
generate object of concrete class based on
given information.

ShapeFactory.java

public class ShapeFactory extends AbstractFactory {
@Override
        public Shape getShape(String shapeType){
                if(shapeType == null){
                return null;
        }
                if(shapeType.equalsIgnoreCase("CIRCLE")){
                return new Circle();
        } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
                return new Rectangle();
        } else if(shapeType.equalsIgnoreCase("SQUARE")){
                return new Square();
        }
        return null;
```

```
}
@Override
        Color getColor(String color) {
        return null;
        }
}
```

ColorFactory.java

```
public class ColorFactory extends AbstractFactory {
@Override
        public Shape getShape(String shapeType){
        return null;
        }
@Override
        Color getColor(String color) {
                if(color == null){
                return null;
                }
                if(color.equalsIgnoreCase("RED")){
                        return new Red();
                } else if(color.equalsIgnoreCase("GREEN")){
                        return new Green();
                } else if(color.equalsIgnoreCase("BLUE")){
                        return new Blue();
                }
                return null;
                }
        }
```

Step 7
Create a Factory generator/producer class to get factories by passing an information such as Shape or Color.

FactoryProducer.java

```
public class FactoryProducer {
    public static AbstractFactory getFactory(String choice){
        if(choice.equalsIgnoreCase("SHAPE")){
           return new ShapeFactory();
        } else if(choice.equalsIgnoreCase("COLOR")){
                 return new ColorFactory();
                }
                return null;
                }
}
```

Step 8
Use the FactoryProducer to get AbstractFactory in order to get factories of concrete classes by passing information such as type.

AbstractFactoryPatternDemo.java

```
public class AbstractFactoryPatternDemo {
        public static void main(String[] args) {
//get shape factory
        AbstractFactory shapeFactory =
                FactoryProducer.getFactory("SHAPE");
//get an object of Shape Circle
        Shape shape1 = shapeFactory.getShape("CIRCLE");
//call draw method of Shape Circle
```

```
        shape1.draw();
//get an object of Shape Rectangle
        Shape shape2 = shapeFactory.getShape("RECTANGLE");
//call draw method of Shape Rectangle
        shape2.draw();
//get an object of Shape Square
        Shape shape3 = shapeFactory.getShape("SQUARE");
//call draw method of Shape Square
        shape3.draw();
//get color factory
        AbstractFactory colorFactory =
                FactoryProducer.getFactory("COLOR");
//get an object of Color Red
        Color color1 = colorFactory.getColor("RED");
//call fill method of Red
        color1.fill();
//get an object of Color Green
        Color color2 = colorFactory.getColor("Green");
//call fill method of Green
        color2.fill();
//get an object of Color Blue
        Color color3 = colorFactory.getColor("BLUE");
//call fill method of Color Blue
        color3.fill();
        }
}
```

Step 9

```
Verify the output.

Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
Inside Red::fill() method.
Inside Green::fill() method.
Inside Blue::fill() method.
```

Patrón Singleton

```
Steps
Use the following steps to implement the above mentioned
design pattern.

Step 1
Create a Singleton Class.

SingleObject.java

public class SingleObject {
//create an object of SingleObject
  private static SingleObject instance = new SingleObject();
//make the constructor private so that this class cannot be
//instantiated
        private SingleObject(){}
//Get the only object available
        public static SingleObject getInstance(){
        return instance;
        }
        public void showMessage(){
                System.out.println("Hello World!");
        }
}
```

```
Step 2
Get the only object from the singleton class.

SingletonPatternDemo.java

public class SingletonPatternDemo {
        public static void main(String[] args) {
//illegal construct
//Compile Time Error: The constructor SingleObject() is not
                visible
//SingleObject object = new SingleObject();
//Get the only object available
        SingleObject object = SingleObject.getInstance();
//show the message
                object.showMessage();
        }
}

Step 3
Verify the output.

        Hello World!
```

Patrón Buider

```
Steps
Use the following steps to implement the above
mentioned design pattern.

Step 1
Create an interface Item
representing food item and packing.
```

Item.java

```
public interface Item {

        public String name();

        public Packing packing();

        public float price();
    }
```

Packing.java

```
    public interface Packing {

        public String pack();
    }
```

Step 2
Create concreate classes implementing the Packing interface .
Wrapper.java

```
public class Wrapper implements Packing {

            @Override
            public String pack() {
                return "Wrapper";
            }
        }
```

Bottle.java

```
public class Bottle implements Packing {

    @Override
    public String pack() {
        return "Bottle";
    }
}
```

Step 3
Create abstract classes implementing the item interface providing default functionalities.

Burger.java

```
public abstract class Burger implements Item {

        @Override
        public Packing packing() {
            return new Wrapper();
        }

        @Override
        public abstract float price();
    }
```

ColdDrink.java

```
public abstract class ColdDrink implements Item {
    @Override
    public Packing packing() {
        return new Bottle();
    }
    @Override
    public abstract float price();
}
```

Step 4
Create concrete classes extending
Burger and ColdDrink classes.

VegBurger.java

```
public class VegBurger extends Burger {
    @Override
    public float price() {
        return 25.0f;
    }
    @Override
    public String name() {
        return "Veg Burger";
    }
}
```

ChickenBurger.java

```
public class ChickenBurger extends Burger {
    @Override
    public float price() {
        return 50.5f;
    }
    @Override
    public String name() {
        return "Chicken Burger";
    }
}
```

Coke.java

```
public class Coke extends ColdDrink {

    @Override
    public float price() {
        return 30.0f;
    }
    @Override
    public String name() {
        return "Coke";
    }
}
```

Pepsi.java

```
public class Pepsi extends ColdDrink {

    @Override
    public float price() {
        return 35.0f;
    }

    @Override
    public String name() {
        return "Pepsi";
    }
}
```

Step 5
Create a Meal class having
Item objects defined above.

Meal.java

```
import java.util.ArrayList;
import java.util.List ;
     public class Meal {
         private List<Item> items = new ArrayList<Item>();
                  public void addItem(Item item) {
                    items.add(item);
                  }



                public float getCost() {
```

```
                        float cost = 0.0f;
                        for (Item item : items) {
                            cost += item.price();
                        }
                        return cost;
                    }

  public void showItems() {
      for (Item item : items) {
        System.out.print("Item : " + item.name());
  System.out.print(", Packing : " + item.packing().pack());
       System.out.println(", Price : " + item.price());
        }
     }
  }
```

Step 6
Create a MealBuilder class, the actual
builder class responsible to create Meal objects.

MealBuilder.java

```
public class MealBuilder {

                        public Meal prepareVegMeal() {
                            Meal meal = new Meal();
                            meal.addItem(new VegBurger());
                            meal.addItem(new Coke());
                            return meal;
                           }

                    public Meal prepareNonVegMeal() {
```

```
        Meal meal = new Meal();
        meal.addItem(new ChickenBurger());
        meal.addItem(new Pepsi());
        return meal;
      }
    }
```

Step 7
BuiderPatternDemo uses MealBuider to demonstrate
builder pattern.

BuilderPatternDemo.java

```
public class BuilderPatternDemo {
   public static void main(String[] args) {
        MealBuilder mealBuilder = new MealBuilder();
        Meal vegMeal = mealBuilder.prepareVegMeal();
        System.out.println("Veg Meal");
        vegMeal.showItems();
        System.out.println("Total Cost: "
          + vegMeal.getCost());
   Meal nonVegMeal = mealBuilder.prepareNonVegMeal();
        System.out.println("\n\nNon-Veg Meal");
        nonVegMeal.showItems();
        System.out.println("Total Cost: "
          + nonVegMeal.getCost());
                }
              }
```

```
Step 8
Verify the output.Veg Meal


Item          : Veg Burger, Packing : Wrapper , Price : 25.0
Item          : Coke , Packing : Bottle , Price : 30.0
Total Cost    : 55.0

Non -Veg Meal

Item      : Chicken Burger, Packing : Wrapper , Price : 50.5
Item      : Pepsi , Packing : Bottle , Price : 35.0
Total Cost    : 85.5
```

Patrón Prototype

```
Steps
Use the following steps to implement the above mentioned
design pattern.

Step 1
Create an abstract class implementing Clonable interface

        .
Shape.java

public abstract class Shape implements Cloneable {

            private String id;
            protected String type;

            abstract void draw();

            public String getType() {
```

```
            return type;
        }

        public String getId() {
            return id;
        }

        public void setId(String id) {
            this.id = id;
        }

        public Object clone() {
            Object clone = null;
            try {
                clone = super.clone();
            } catch (CloneNotSupportedException e) {
                e.printStackTrace();
            }
            return clone;
        }
    }
```

Step 2

Create concrete classes extending the above class .

Rectangle.java

```
public class Rectangle extends Shape {

            public Rectangle() {
                type = "Rectangle";
            }
            @Override
```

```
        public void draw() {
            System.out.println("Inside Rectangle:"
                    + ":draw() method.");
        }
    }
```

Square.java

```
    public class Square extends Shape {

        public Square() {
            type = "Square";
        }

        @Override
        public void draw() {
            System.out.println("Inside Square:"
                    + ":draw() method.");
        }
    }
```

Circle.java

```
    public class Circle extends Shape {

        public Circle() {
            type = "Circle";
        }

        @Override
        public void draw() {
            System.out.println("Inside Circle:"
```

```
                                + ":draw() method.");
                    }
                }
Step 3
Create a class to get concrete classes from database
and store them in a Hashtable.

ShapeCache.java

import java.util.Hashtable;
public class ShapeCache {
    private static Hashtable<String, Shape>
            shapeMap = new Hashtable<String, Shape>();
        public static Shape getShape(String shapeId) {
            Shape cachedShape = shapeMap.get(shapeId);
            return (Shape) cachedShape.clone();
        }
// for each shape run database query and create shape
// shapeMap.put(shapeKey, shape);
// for example, we are adding three shapes

            public static void loadCache() {
                Circle circle = new Circle();
                circle.setId("1");
                shapeMap.put(circle.getId(), circle);
                Square square = new Square();
                square.setId("2");
                shapeMap.put(square.getId(), square);
                Rectangle rectangle = new Rectangle();
                rectangle.setId("3");
                shapeMap.put(rectangle.getId(), rectangle);
            }
```

```
        }
```

Step 4
PrototypePatternDemo uses ShapeCache class to
get clones of shapes stored in a Hashtable.

PrototypePatternDemo.java

```
public class PrototypePatternDemo {
  public static void main(String[] args) {
   ShapeCache.loadCache();
   Shape clonedShape = (Shape) ShapeCache.getShape("1");
   System.out.println("Shape : " + clonedShape.getType());
   Shape clonedShape2 = (Shape) ShapeCache.getShape("2");
   System.out.println("Shape : " + clonedShape2.getType());
   Shape clonedShape3 = (Shape) ShapeCache.getShape("3");
   System.out.println("Shape : " + clonedShape3.getType());
      }
   }
```

Step 5
Verify the output.Shape.

```
        Shape : Circle
        Shape : Square
        Shape : Rectangle
```

Apéndice B

B.1. Base del conocimiento patrones creacionales.

FACTORY

DISEÑO / PATRÓN	PRIMITIVO	CONSTANTE	REFERENCIA	ARREGLO	CLASE	PRIMITIVO	CONSTRUCTOR	REFERENCIA	ARRREGLO	ABSTRACT	OVERRIDE	CLASE	CONSTRUCTOR_CLASE	OBJET		
INTERFACE						1									1	
ABSTRACTA															0	
ENUMERACIÓN															0	2
CONCRETA								1							1	
HIJA															0	
EXTENDIDA															0	
IMPLEMENTADA						1					1				2	3
ASOCIACIÓN															0	
	0	0	0	0	0	2	0	1	0	0	1	0	0		4	
		Variables			0						Métodos			4	4	

Figura B.1: ***(Factory)***

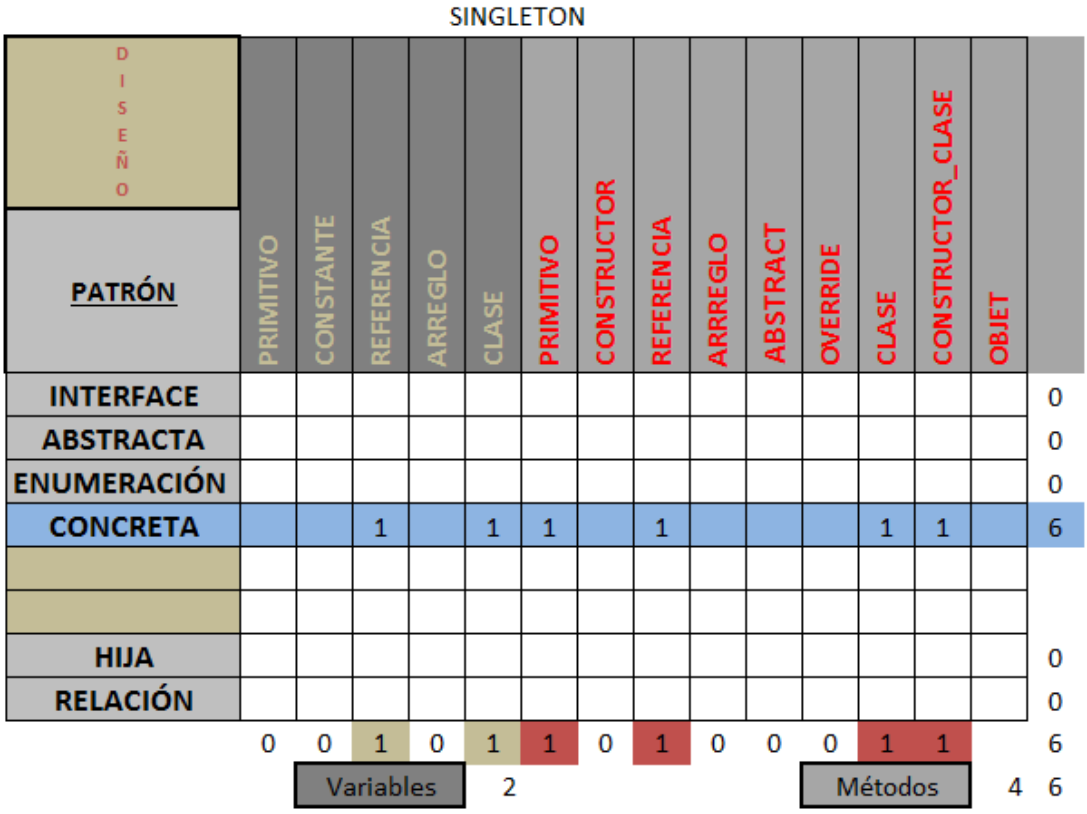


Figura B.2: ***(Singleton)***

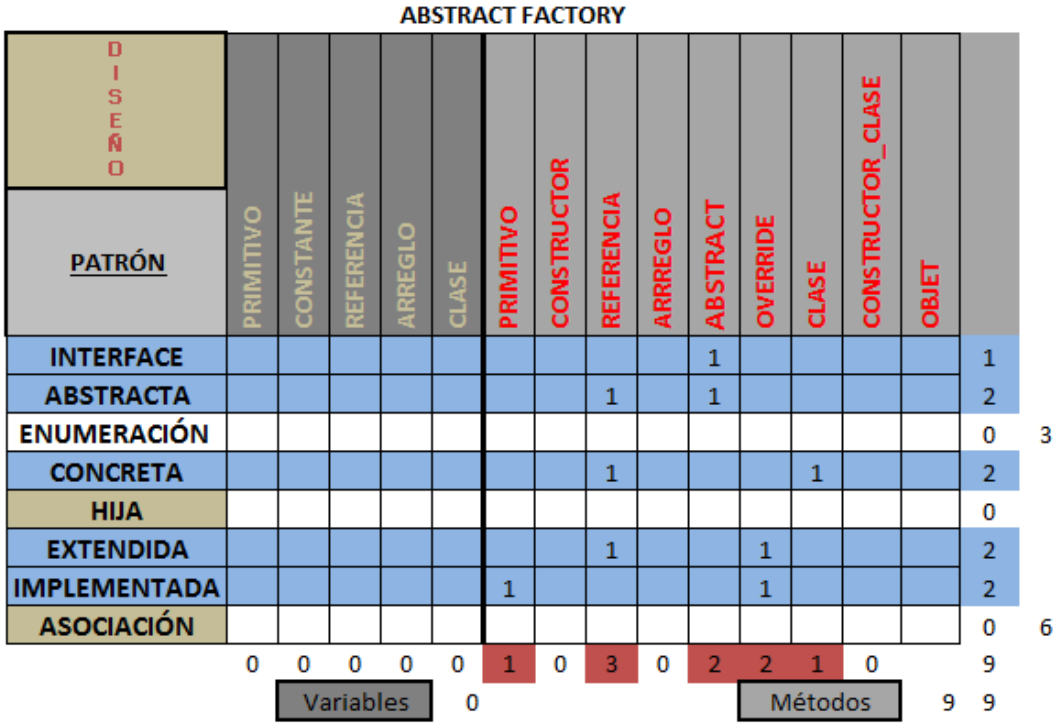


Figura B.3: ***(Abstarct Factory)***

PROTOTYPE

DISEÑO / PATRÓN	PRIMITIVO	CONSTANTE	REFERENCIA	ARREGLO	CLASE	PRIMITIVO	CONSTRUCTOR	REFERENCIA	ARRREGLO	ABSTRACT	OVERRIDE	CLASE	CONSTRUCTOR_CLASE	OBJET		
INTERFACE								1				1			2	
ABSTRACTA						1		1		1	1			1	5	
ENUMERACIÓN															0	7
CONCRETA															0	
HIJA															0	
EXTENDIDA							1				1				2	
IMPLEMENTADA															0	
INTERNA															0	
ASOCIACIÓN			1	1				1							3	5
	0	0	1	1	0	1	1	3	0	1	2	1	0	1	12	
		Variables			2						Métodos			10	12	

Figura B.4: ***(Prototype)***

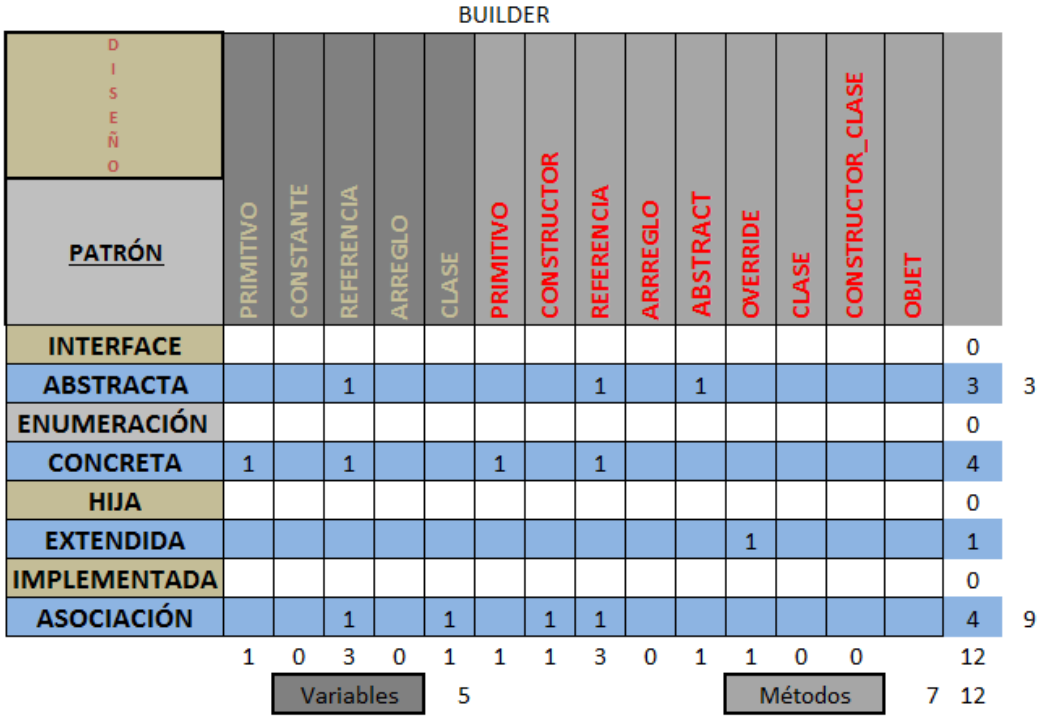
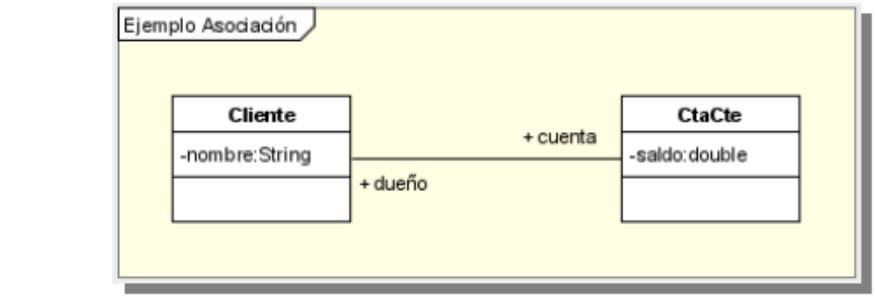


Figura B.5: ***(Builder)***

Apéndice C

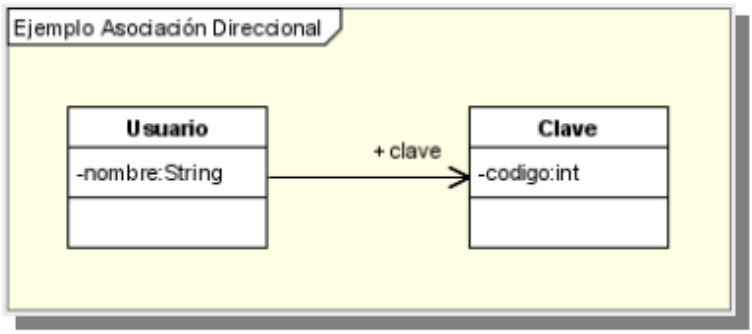
C.1. Analisis de código piezas de diseño y piezas de patrón.




```
public class Cliente {

    private String nombre;

    public CtaCte cuenta;

}
```

```
public class CtaCte {

    private double saldo;

    public Cliente dueño;

}
```

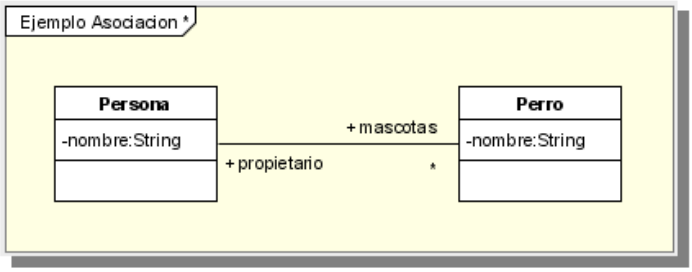
Figura C.1: Asociación con multiplicidad.




```
public class Usuario {

    private String nombre;

    public Clave clave;

}
```

```
public class Clave {

    private int codigo;

}
```

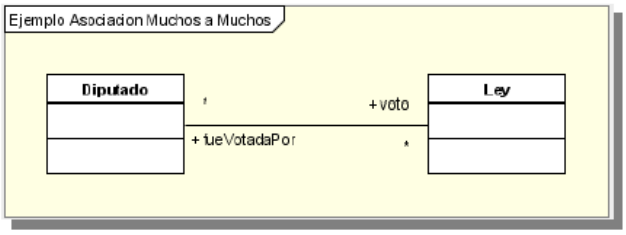
Figura C.2: Asociación Direccional con multiplicidad.




```
public class Persona {

    private String nombre;

    public java.util.Collection mascotas = new java.util.TreeSet();

}
```

```
public class Perro {

    private String nombre;

    public Persona propietario;

}
```

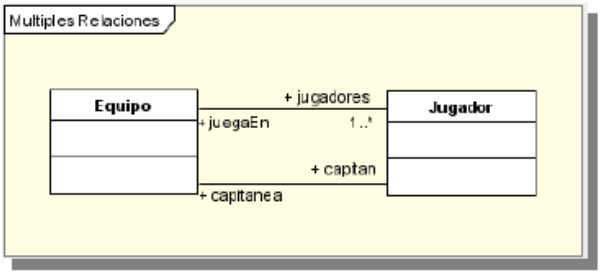
Figura C.3: Asociación Bidireccional con multiplicidad.




```
public class Diputado {

    public java.util.Collection voto = new java.util.TreeSet();

}
```

```
public class Ley {

    public java.util.Collection fueVotadaPor = new java.util.TreeSet();

}
```

Figura C.4: Asociación con multiplicidad Muchos a Muchos.




```
public class Equipo {

    public Jugador capitan;

    public java.util.Collection jugadores = new java.util.TreeSet();

}
```

```
public class Jugador {

    public Equipo capitanea;

    public Equipo juegaEn;

}
```

Figura C.5: Asociación con más de una relación.

Bibliografía

(). *clasificación de patrones*. John Wiley Sons,.

(). Conjuntos difusos y lógica difusa, teoría y aplicaciones .

(). *Una teoría probabilística de reconocimiento de patrones*. springer.

Alexander, C. (1979). *The Timeless Way of Building*. Oxford University Press.

Bansiya, J. (1998). Automating design-pattern identification. *Dr. Dobb's journal* **23**(6).

Bashon, Y., Neagu, D. & Ridley, M. J. (2011). Fuzzy set-theoretical approach for comparing objects with fuzzy attributes. In: *Intelligent Systems Design and Applications (ISDA), 2011 11th International Conference on*. IEEE.

Bosch, J., Bachatene, H., Hedin, G. & Koskimies, K. (1998). Oosa98: Ecoop'98 workshop on object-oriented software architectures .

Bouchon-Meunier, B., Rifqi, M. & Bothorel, S. (1996). Towards general measures of comparison of objects. *Fuzzy sets and systems* **84**(2), 143–153.

Budinsky, F. J., Finnie, M. A., Vlissides, J. M. & Yu, P. S. (1996). Automatic code generation from design patterns. *IBM systems Journal* **35**(2), 151–171.

Bunke, H. & Allermann, G. (1983). Inexact graph matching for structural pattern recognition. *Pattern Recognition Letters* **1**(4), 245–253.

Buschmann, F. & Henney, K. (2003). Explicit interface and object manager. In: *EuroPLoP*.

Bustince, H. (2000). Indicator of inclusion grade for interval-valued fuzzy sets. application to approximate reasoning based on interval-valued fuzzy sets. *International Journal of Approximate Reasoning* **23**(3), 137–209.

Chambers, C., Harrison, B. & Vlissides, J. (2000). A debate on language and tool support for design patterns. In: *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM.

Chatzigeorgiou, A., Tsantalis, N. & Stephanides, G. (2006). Application of graph theory to oo software engineering. In: *Proceedings of the 2006 international workshop on Workshop on interdisciplinary software engineering research*. ACM.

Cooper, J. W. (2000). *Java design patterns: a tutorial*. Addison-Wesley Professional.

Cruz, P. P. (2011). *Inteligencia artificial con aplicaciones a la ingeniería*. Alfaomega.

DeMarco, T. (1979). *Structured analysis and system specification*. Yourdon Press.

Díaz, M. P., Montero, S. & Aedo, I. (2005). *Ingeniería de la web y patrones de diseño*. Pearson Prentice Hall.

Dodge, S., Weibel, R. & Lautenschütz, A.-K. (2008). Towards a taxonomy of movement patterns. *Information visualization* **7**(3-4), 240–252.

Foster, T. & Zhao, L. (1998). Modeling transport objects with patterns. *JOURNAL OF OBJECT-ORIENTED PROGRAMMING* **10**(8), 26–32.

Fowler, M. (). Analysis patterns .

Galiano, F. B. (2002). *ART: un método alternativo para la construcción de árboles de decisión*. Ph.D. thesis, Universidad de Granada.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Pearson Education.

Gottwald, S. (2001). *A treatise on many-valued logics*, vol. 3. Research Studies Press Baldock.

Guerrero, C. A., Suárez, J. M. & Gutiérrez, L. E. (2013). Patrones de diseño gof (the gang of four) en el contexto de procesos de desarrollo de aplicaciones orientadas a la web. *Información tecnológica* **24**(3), 103–114.

Han, J. & Kamber, M. (2006). Data mining: Concepts and techniques, 2nd editionmorgan kaufmann publishers. *San Francisco, CA, USA* .

Hayashi, S., Katada, J., Sakamoto, R., Kobayashi, T. & Saeki, M. (2008). Design pattern detection by using meta patterns. *IEICE TRANSACTIONS on Information and Systems* **91**(4), 933–944.

Hericko, M. & Beloglavec, S. (2005). A composite design-pattern identification technique. *Informatica* **29**(4).

Heričko, M. & Beloglavec, S. (2005). A composite design-pattern identification technique. *Special Issue: Hot Topics in European Agent Research I Guest Editors: Andrea Omicini* , 469.

Jensen, R. & Lopes, M. H. B. d. M. (2011). Nursing and fuzzy logic: an integrative review. *Revista latino-americana de enfermagem* **19**(1), 195–202.

Johnson, R. E. (1992). Documenting frameworks using patterns. In: *ACM Sigplan Notices*, vol. 27. ACM.

Kosko, B. (1992). *Neural Networks and Fuzzy Systems: A Dynamical Systems Approach to Machine Intelligence/Book and Disk*, vol. 1. Prentice hall.

Krämer, C. & Prechelt, L. (1996). Design recovery by automated search for structural design patterns in object-oriented software. In: *Reverse Engineering, 1996., Proceedings of the Third Working Conference on*. IEEE.

Lado., R. T. (2010). *TALLER DE PATRONES DE DISEO*, vol. 1. Universidad de Zaragoza.

Larman, C. (1999). *UML y Patrones*. Pearson.

Larman, C. & Applying, U. (2001). Patterns: An introduction to object-oriented analysis and design and the unified process.

Laube, P., Imfeld, S. & Weibel, R. (2005). Discovering relative motion patterns in groups of moving point objects. *International Journal of Geographical Information Science* **19**(6), 639–668.

Medina, J. I. & Luque, C. J. (2008). La geometría de minkowski a partir del grupo de lorentz .

Reina, D. & Moscovitz, L. J. (2008). *Fundamentos de Matemática Difusa*. Ph.D. thesis, Tesis de Grado. Fundación Universitaria Konrad Lorenz. Facultad de Matemáticas. Búsqueda realizada 18-19-10. Dirección electrónica http://www. konradlorenz. edu. co/images/stories/suma_digital_matematicas/EDICION_09_0I/trabajo_de_grado_daniel_reina. pdf.

Ross, T. J. (2009). *Fuzzy logic with engineering applications*. John Wiley & Sons.

Schmidt, D. C., ORyan, C., Kircher, M., Pyarali, I. & Buschmann, F. (1998). Leader/followers. In: *PLoP conference. http://hillside. net/plop/-plop2k/proceedings/ORyan/ORyan. pdf*.

SCHMULLER, J. (2000). *APRENDIENDO UML EN 24 HORAS*. S.A. ALHAMBRA MEXICANA.

Sequera, D. (2010). Estudio sobre la aplicación de operaciones de intersección y diferencia en medidas de comparación entre conjuntos difusos. *Trabajo Especial de Grado, Licenciatura en Ciencias Matemáticas, UCLA* .

Shull, F., Melo, W. L. & Basili, V. R. (1998). An inductive method for discovering design patterns from object-oriented software systems .

Stytz, M. R. & Block, E. (1993). A fuzzy logic assistant for virtual environment operators immersed in a battlespace. In: *AI, Simulation, and Planning in High Autonomy Systems, 1993. Integrating Virtual Reality and Model-Based Environments. Proceedings. Fourth Annual Conference*. IEEE.

Taibi, T. (2007). *Design patterns formalization techniques*. Igi Global.

Tanaka, K. (1997). An introduction to fuzzy logic for practical applications .

Tversky, A. & Gati, I. (1982). Similarity, separability, and the triangle inequality. *Psychological review* **89**(2), 123.

Velasco-Elizondo, P. (2013). *2nd Summer School in Software Engineering*.

Villa, M. C. (2011). Modelo para la ayuda a la toma de decisiones en la selección de patrones de desarrollo de software. *Serie Científica* **4**(2).

Welicki, L. E. & Aguilar, L. J. (2014). Meta-especificación y catalogación de patrones de software con lenguajes de dominio específico y modelos de objetos adaptativos: una vía para la gestión del conocimiento en la ingeniería del software .

Yen, J. & Langari, R. (1998). *Fuzzy logic: intelligence, control, and information*. Prentice-Hall, Inc.

Zadeh, L. A. (1971). Similarity relations and fuzzy orderings. *Information sciences* **3**(2), 177–200.