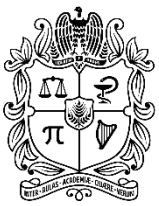


UNIVERSIDAD
NACIONAL
DE COLOMBIA

Identificación de patrones de diseño para software científico a partir de esquemas preconceptuales

Johnathan Mauricio Calle Gallego

Universidad Nacional de Colombia
Facultad de Minas
Medellín, Colombia
2016

Identificación de patrones de diseño para software científico a partir de esquemas preconceptuales

Johnathan Mauricio Calle Gallego

Tesis de investigación presentada como requisito parcial para optar al título de:

Magister en Ingeniería de Sistemas

Director:

Ph.D. Carlos Mario Zapata Jaramillo

Línea de Investigación:

Ingeniería de software

Universidad Nacional de Colombia

Facultad de Minas

Medellín, Colombia

2016

A mi madre, por su amor y apoyo incondicional.

A mis abuelos, por ser la muestra perfecta de lucha y constancia.

A Carolina Quiroga, por su amor incondicional.

Agradecimientos

Quiero extender un agradecimiento especial a mi director, el profesor Carlos Mario Zapata por su gran apoyo en los últimos años durante mi pregrado y posgrado, por las oportunidades y orientación constante.

Al profesor Juan Manuel Mejía por darme la oportunidad de ser parte de su grupo de investigación durante los dos últimos años.

Resumen

Los patrones de diseño son soluciones a problemas de diseño recurrentes en software científico. Estos patrones se usan para mitigar la ausencia de algunos aspectos de calidad inherentes al software. Los científicos, debido a su formación profesional, abordan diseños poco flexibles y difíciles de mantener en sus aplicaciones. Además, en ausencia de un lenguaje común con ingenieros de software se hace muy compleja la comunicación y validación del dominio de aplicación.

Normalmente, las representaciones de los patrones de diseño se basan en diagramas de UML u otro tipo de grafos. Estos diagramas son difíciles de entender para los científicos e ingenieros de software inexpertos debido a su nivel de formalismo y, además, porque sólo representan el patrón de diseño aplicado y no el problema genérico que resuelven. Por otro lado, estos diagramas como unidad no poseen los elementos necesarios para representar completamente un dominio de software científico y se deben valer de la combinación de varios de ellos para hacerlo.

Por ello, en esta Tesis de Maestría se propone una representación en esquemas preconceptuales de los patrones de diseño más usados en software científico y, además, la representación genérica del problema que resuelven. Adicionalmente, se presentan una serie de nuevos elementos para los esquemas preconceptuales que permiten la completa representación y validación de los dominios complejos presentes en el software científico.

Al usar esquemas preconceptuales se facilita el entendimiento de los patrones de diseño debido a su proximidad con el lenguaje natural y a los elementos disponibles para su representación. Además, se hace posible la comunicación y validación del dominio de aplicación entre científicos e ingenieros de software. Este trabajo ayuda a la comunidad científica a hacer un diseño más robusto, flexible y fácil de mantener de su aplicación, y además, abre las puertas a la automatización de la implementación de los patrones de diseño a partir de una representación del dominio en esquemas preconceptuales.

Palabras clave: Patrones de diseño; Software científico; Diseño de software; Ingeniería de software; Representación gráfica.

Abstract

Design patterns are solutions for recurrent design problems in scientific software. These patterns are used to mitigate the lack of several quality aspects inherent in the software. Scientists, due to their professional training, tackle little flexible and maintainable designs for their software applications. In addition, in the absence of a common vocabulary with software engineers, domain communication and validation becomes complex.

Normally, design patterns representation are based on UML class diagrams or other kind of graphs. These diagrams are difficult to understand for scientist and inexperienced software engineers due to their level of formalism and, also because of this diagrams only represents the implemented design pattern and not the generic problem the design patterns solves. Furthermore, these diagrams as unity do not have the necessary elements to represent scientific software domains completely, so they must combine to do it.

For this reason, in this Master's Thesis it is proposed a representation of design patterns for scientific software by using preconceptual schemes, and also, a generic representation of the problem that design patterns address. Additionally, it is proposed a number of new elements for preconceptual schemes that allows a complete representation and validation of complex domains in scientific software.

By using preconceptual schemes facilitates design patterns understanding due to their natural language proximity. In addition, it is made possible the validation and communication of application domain between scientists and software engineers. This work helps scientific community to make robust, flexible and maintainable software applications, and also, opens the doors to automated design pattern implementation from domain representation in preconceptual schemes.

Keywords: Scientific software; Software design; Software engineering; Graphic representation.

Contenido

Pág.

Lista de figuras

Pág.

Lista de tablas

Pág.

Lista de abreviaturas

Abreviaturas

Abreviatura	Término
At.	*Attribute*
EP	Esquema Preconceptual
DC	Diagrama de Clases
OOM	Object Oriented Modeling
Op.	*Operation*
PD	Patrón de Diseño
POO	Programación Orientada a Objetos

1. Introducción

1.1 Justificación

En la actualidad, el software se considera de uso general y comercial en lugar de ser sólo para una persona o un pequeño grupo de expertos. Debido a este cambio de alcance, surge la necesidad de construir software confiable, escalable, flexible y fácil de mantener (Hannay *et al*., 2009; Sanders y Kelly, 2008; Segal, 2007). Los patrones de diseño se usan en software científico para implementar adecuadamente métodos de diseño y programación bajo el paradigma de Programación Orientada a Objetos (POO) y, así, aportar aspectos de calidad ausentes en este tipo de aplicaciones de software (Barbieri *et al.*, 2012; Hannay *et al*., 2009).

Los científicos cada vez gastan más tiempo y recursos en el desarrollo de sus propios sistemas debido a su poca experiencia en ingeniería de software (Wilson *et al*., 2014). Al ser desarrolladores y usuarios finales de sus propias aplicaciones, dejan varias etapas de desarrollo por fuera del ciclo de vida del software. Estas decisiones se deben a la poca importancia que otorgan al desarrollo de sus aplicaciones (Hannay *et al*., 2009). Este tipo de faltas los lleva rápidamente a aplicaciones muy difíciles de mantener y extender en el tiempo. Además, debido a la complejidad del dominio de aplicación, usualmente un tema específico de estudio con sólo un par de expertos en el mundo o una ciencia emergente, la comunicación y validación de este tipo de dominio con ingenieros de software es una tarea muy difícil de completar (Sanders y Kelly, 2008).

Al implementar POO se afecta directamente la eficiencia computacional de este tipo de aplicaciones. Es allí cuando los patrones de diseño se hacen necesarios (Gardner, 2004; Rodriguez *et al.*, 2004). Estos patrones definen una arquitectura adecuada para contrarrestar los efectos de este paradigma en la eficiencia del software científico (Cickovski *et al.*, 2004; Panagiotis y Margaritis, 2014).

En la literatura se encuentran varias aproximaciones al uso de POO en desarrollo de software científico donde se critica la efectividad de UML y otras modelos gráficos para representar el dominio (Ackroyd *et al*., 2008; Hannay *et al*., 2009; Kelly, 2007; Morris y Segal, 2012; Segal y Morris, 2008, 2009) y, además, para la ejemplificación de los patrones de diseño como herramienta de implementación de dicho paradigma (Barbieri *et al*., 2012; Morris y Segal, 2012; Sanders y Kelly, 2008).

Los patrones de diseño, al ser modelos estándar de una estructura o proceso cuya arquitectura se puede representar gráficamente (Ahmed, 2015), se pueden representar en esquemas preconceptuales. Por ello, en esta Tesis de Maestría se propone una representación de los patrones de diseño y la representación genérica del problema que resuelven en esquemas preconceptuales. Esta representación se ubica en un nivel intermedio de formalismo entre las representaciones rígidas de UML (Ackroyd *et al*., 2008; Carver *et al.*, 2007) y las representaciones menos formales de otros tipos de grafos (Gamma *et al.*, 1994). Estos esquemas sirven, también, como herramienta de comunicación y validación de los patrones de diseño y del dominio de aplicación entre los científicos e ingenieros de software, debido a su proximidad con el lenguaje natural (Zapata y Arango, 2007). Finalmente, se propone una serie de nuevos elementos para los esquemas preconceptuales que facilitan la representación estructural de estos modelos en un único diagrama. Este trabajo resalta los beneficios de aplicar patrones de diseño al contexto científico y aporta una manera de construir software flexible, escalable y fácil de mantener a partir de reglas de diseño de POO.

1.2 Definición del problema

Normalmente, la representación del dominio de aplicación en la etapa de requisitos y diseño de software se hace mediante diagramas de UML (Zhu, 2005). Estos diagramas, debido a su nivel técnico, no son muy útiles para comunicar y validar el dominio en contextos científicos (Ahmed, 2015; Morris y Segal, 2012; Panagiotis y Margaritis, 2014; Wilson *et al*., 2014). Además, se deben emplear varios de estos diagramas para representar algunos aspectos como las interacciones y secuencias complejas que existen en software científico (Morris y Segal, 2012; Shalloway y Trott, 2004).

La representación de los patrones de diseño usualmente incluye uno o varios diagramas de UML y, en algunos casos, de otro tipo de grafos (Barbieri *et al*., 2012; Gardner, 2004; Panagiotis y Margaritis, 2014). Estas representaciones, debido a su nivel de formalismo, hacen muy complejo el proceso de entendimiento e implementación de dichos patrones para los científicos e ingenieros de software inexpertos. Por otro lado, estas representaciones sólo reflejan la implementación del patrón de diseño, dejando a un lado la situación genérica del problema que resuelven (Ahmed, 2015; Hannay *et al*., 2009; Panagiotis y Margaritis, 2014; Segal, 2009; Segal y Morris, 2009; Shalloway y Trott, 2004; Wilson *et al*., 2014).

1.3 Objetivo general

Definir la arquitectura de los patrones de diseño para software científico en esquemas preconceptuales usando elementos adicionales para permitir el completo modelado de dominios complejos en software científico.

1.4 Objetivos específicos

- Identificar los patrones de diseño para software científico en la literatura que sean candidatos para la representación en esquemas preconceptuales.
- Analizar las estructuras recurrentes en dominios de software científico para definir su representación en esquemas preconceptuales.
- Representar los patrones de diseño para software científico en esquemas preconceptuales.
- Definir las reglas que permitan obtener los conceptos adicionales correspondientes a cada uno de los patrones de diseño de estudio.
- Validar las representaciones de los patrones de diseño con otros esquemas disponibles en la literatura.

1.5 Metodología

La metodología que se utiliza para cumplir cada uno de los objetivos planteados en esta Tesis se divide en cuatro fases: exploración, análisis, desarrollo y validación.

1.5.1 Exploración

- **Identificar los patrones de diseño para software científico en la literatura que sean candidatos para la representación en esquemas preconceptuales.**
    1. Analizar artículos y libros sobre patrones de diseño en software científico, su uso e implementación.
    2. Definir los patrones de diseño para software científico que serán objeto de estudio.

1.5.2 Análisis

- **Analizar las estructuras recurrentes en dominios de software científico para definir su representación en los esquemas preconceptuales.**
    1. Revisar los componentes inherentes al proceso de desarrollo de software científico (Modelo matemático, condiciones iniciales, etc.) e identificar cuáles de estos son estrictamente necesarios para modelar completamente el dominio de un software científico.
    2. Identificar los procesos recurrentes y crear un elemento nuevo para representarlo en el esquema preconceptual.

1.5.3 Desarrollo

- **Definir y representar los patrones de diseño para software científico en esquemas preconceptuales.**
    1. Definir la arquitectura de los patrones de diseño para software científico encontrados en la literatura.
    2. Adaptar, si es necesario, los patrones de diseño para el correcto uso de éstos en el desarrollo de software científico.
    3. Definir, si es necesario, elementos adicionales que permitan el modelado completo de la arquitectura del patrón.

- **Definir las reglas que permitan obtener los conceptos adicionales correspondientes a cada uno de los patrones de diseño de estudio.**

1. Definir las condiciones y precondiciones necesarias para que un conjunto de elementos en el esquema preconceptual se pueda llevar a la implementación de uno de los patrones para software científico.

1.5.4 Validación

- **Validar las representaciones de los patrones de diseño con otros esquemas disponibles en la literatura.**
    1. Comparar la representación de los patrones de diseño para software científico en esquemas preconceptuales con otras representaciones validadas conceptualmente o en código encontradas en la literatura.

1.6 Estructura de la tesis

Esta Tesis de Maestría se organiza así: en el Capítulo 2 se presenta el marco teórico que abarca el software científico, el diseño de software, UML (*Unified Modeling Language*), los patrones de diseño y los esquemas preconceptuales; en el Capítulo 3 se exponen los acercamientos a las representaciones de los patrones de diseño que se proponen en la literatura; en el Capítulo 4 se proponen los nuevos elementos de los esquemas preconceptuales para el modelado de dominios complejos y la representación de los patrones de diseño en estos esquemas; en el Capítulo 5 se presenta la validación de las representaciones de los patrones de diseño a partir de las representaciones existentes en la literatura y, finalmente, en el Capítulo 6 se plantean las conclusiones y el trabajo futuro a partir de esta Tesis de Maestría.

2. Marco teórico

2.1 Software científico

El software científico comprende aplicaciones de software con un gran componente de conocimiento científico. Este software se usa, generalmente, para incrementar el conocimiento del dominio de estudio para resolver problemas del mundo real (Kelly, 2015).

En la literatura se definen tres tipos de software científico:

- Software científico industrial: software con un enfoque complejo que suple las necesidades de grandes procesos. Este tipo de software se orienta a un grupo de expertos del dominio (Kelly, 2007).
- Software científico de investigación o de academia: este tipo de software busca apoyar el proceso de una investigación y el alcance es de uno o un grupo pequeño de expertos en el dominio de aplicación (Ackroyd *et al*., 2008; Sanders y Kelly, 2008).
- Software científico a menor escala: este software lo desarrollan estudiantes de disciplinas de la ciencia y, eventualmente, se incluye en alguno de los dos grupos anteriores. Este software es de uso personal con base en un dominio de aplicación en ciencias (Kelly, 2007; Segal y Morris, 2008).

Este tipo de software tiene tres propósitos principales (Sanders y Kelly, 2008), a saber:

- Investigación: desarrollo de teorías y modelos para explorar y enriquecer sus propios dominios de aplicación.
- Entrenamiento: uso de este tipo de software en laboratorios y clases de investigación para formar a la nueva cohorte de investigadores.
- Apoyo a decisiones externas: recolección de datos para la toma de decisiones de agentes externos relacionados con el dominio de aplicación.

Finalmente, Kelly (2015) resume las características que definen a un software científico, que se listan a continuación:

- El software se basa en un modelo matemático, método numérico, modelo o fenómeno físico o un problema de la vida real que requiere intervención de alguno de estos elementos.
- Se requiere un experto o un especialista en el dominio de aplicación para su desarrollo.
- El usuario de este software tiene un conocimiento mínimo requerido para la interpretación de los datos de salida del software.
- El propósito del software no es controlar equipo especializado. El único receptor de los datos de salida, es el usuario final (el científico).
- El propósito principal del software es generar datos que ayuden a entender los problemas del mundo real.

2.1.1 Desarrollo de software científico

Sanders y Kelly (2008) definen el desarrollo de software científico como un ciclo de retroalimentación desde la perspectiva del científico. Una vez el software arroja un resultado, el científico lo analiza y, si es una salida inesperada, modifica el código y la teoría al unísono (Segal y Morris, 2009).

Los científicos tienen dificultades para definir un método de desarrollo claro, dado que, en algunos casos, los científicos no saben qué esperar de los resultados de su aplicación de software o no existen datos para comparar (Li *et al.*, 2015; Segal, 2009; Segal y Morris, 2009). Incluso, cuando la teoría está madura los científicos siguen usando esta aproximación iterativa, adaptando el código a los nuevos hallazgos de la investigación (Kelly, 2015).

2.2 Arquitectura de software

La Arquitectura de software es la línea de la Ingeniería de Software que define una guía de diseño para la construcción de los componentes, interfaces y subsistemas de un sistema computacional. Su objetivo es hacer el desarrollo y la evolución del software manejable mediante su ciclo de vida (De Oliveira y Soares, 2013; Streekmann, 2012).

En Garlan *et al.* (2000) se resalta la relevancia de las arquitecturas de software para el desarrollo de software a gran escala. Asimismo, se introducen sus propósitos más importantes:

- Mejorar el entendimiento de la estructura abstracta de más alto nivel de un sistema computacional.
- Soportar la reutilización de estructuras y componentes de software.
- Definir la estructura abstracta y las restricciones para guiar la construcción del sistema.
- Separar la funcionalidad de los mecanismos de conexión para que puedan evolucionar por separado.
- Facilitar el análisis de las estructuras del sistema.
- Gestionar los componentes de una manera adecuada para explotar su uso.

2.2.1 Diseño de software

El diseño de software o diseño de la arquitectura de software según algunos autores, define la manera en que coexisten y se comunican los módulos y subsistemas de una aplicación de software (Peterson y Guthrie, 2013). El diseño de software se clasifica como uno de los factores clave para asegurar el éxito de un proyecto de software (De Oliveira y Soares, 2013; Garlan *et al.*, 1994).

Durante el diseño de software, normalmente se usan diagramas de UML para representar los componentes del sistema. Asimismo, estos diagramas muestran, a un nivel de abstracción más bajo, las clases y relaciones atómicas de la aplicación de software. A partir de estos diagramas, los analistas de software pueden definir requisitos de usabilidad y de interfaz para completar el diseño general del sistema. Por otro lado, desde esta etapa se puede definir el comportamiento de las clases y sus relaciones atómicas para aportar en el diseño detallado del software (Lee *et al.*, 2012). Una vez se complete el diseño detallado del software, se definen los algoritmos y la naturaleza de los datos que intervienen en el sistema (Peterson y Guthrie, 2013).

Algunos aspectos propios del software como la eficiencia, la robustez y la facilidad de mantenimiento se afectan en la etapa de diseño. En esta etapa se define la arquitectura completa de la aplicación de software que se construye. Uno de los aspectos más

importantes es la adecuada separación de intereses en el sistema, en la que cada componente y módulo se debe crear con un fin específico (De Oliveira y Soares, 2013; Dijkstra, 1972). Un diseño de software correcto debe tener en cuenta la separación de la abstracción de sus módulos y paquetes de su implementación, además de tener en cuenta principios de diseño primarios como la descomposición, el acoplamiento y la cohesión (Parnas, 1972; Streekmann, 2012).

2.2.2 Descomposición

La descomposición en el diseño de software incluye la partición de un sistema en módulos y sus respectivos submódulos (Bass *et al.*, 2003). Los propósitos principales de la descomposición en el diseño son: la asignación de responsabilidades a los módulos y componentes del sistema, la comunicación entre los artefactos que componen la arquitectura y la facilitación del proceso de modificación y adaptación del software durante el ciclo de vida del mismo (Streekmann, 2012).

2.2.3 Acoplamiento

El acoplamiento se define en IEEE (1990) como la manera y el grado de interdependencia de los módulos de un sistema. En el diseño de software se espera lograr un bajo acoplamiento y minimizar las conexiones entre los módulos (Stevens *et al.*, 1974). El efecto esperado de esta condición es la evolución independiente de cada uno de los módulos y submódulos del sistema. Esta aproximación permite mantener fácilmente un sistema computacional (Streekmann, 2012).

2.2.4 Cohesión

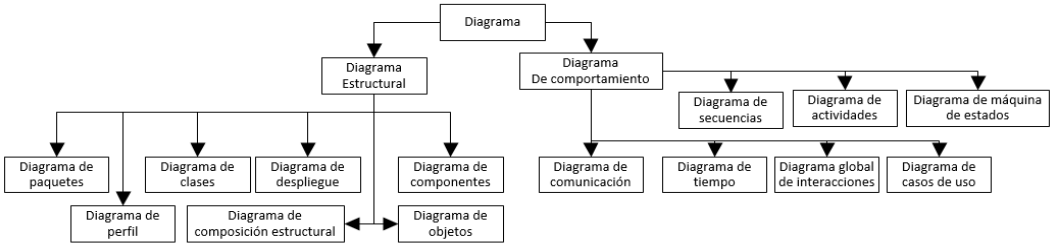
La cohesión se define como la vinculación de los elementos de un módulo o submódulo. Este concepto, a diferencia del acoplamiento, es altamente deseado en el diseño de software. La cohesión maximiza las relaciones dentro de un mismo módulo, submódulo o componente, permitiendo la correlación de la responsabilidad asignada desde el diseño de la arquitectura (Streekmann, 2012). Un alto nivel de cohesión indica un bajo acoplamiento (Stevens *et al*., 1974).

2.3 UML

El Lenguaje de Modelado Unificado (UML por sus siglas en inglés) es una familia de notaciones gráficas soportada en un metamodelo que ayuda a describir los artefactos y el diseño de sistemas de software (Fowler y Kendall, 1999). Este lenguaje nace de la unificación de muchos lenguajes gráficos de modelado orientado a objetos (Fowler, 2004).

UML incluye una notación estándar para expresar gráficamente los diseños y modelos en el software. Este lenguaje unificado abarca un largo y diverso conjunto de dominios de aplicación (The Object Management Group, 2005). Bashir *et al.* (2016) presenta una jerarquía de diagramas con base en las notaciones gráficas que componen UML en su versión 2.5 (Véase la Figura 2-1).

Figura 2-1: Jerarquía de diagramas que componen la familia de representaciones gráficas en UML 2.5.



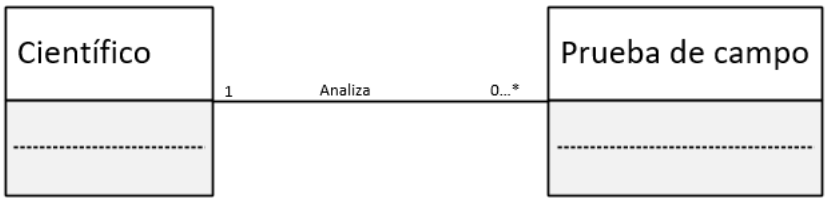
Fuente: Adaptado de Bashir *et al.* (2016).

2.3.1 Diagrama de Clases

El diagrama de clases hace parte de la familia de los diagramas estructurales de UML. Estos diagramas describen los tipos de objetos y conexiones existentes en el sistema. Además, permiten mostrar la estructura estática de las clases y entidades presentes en el dominio de aplicación, representando de una manera gráfica, la manera en que los usuarios piensan el mundo (Fowler, 2004). Existen dos tipos principales de relaciones:

- Asociación: esta relación describe la interacción entre dos clases o conceptos del dominio de aplicación (Fowler, 2004). Por ejemplo, un científico puede analizar una o varias pruebas de campo. Este ejemplo se representa como se muestra en la Figura 2-2.

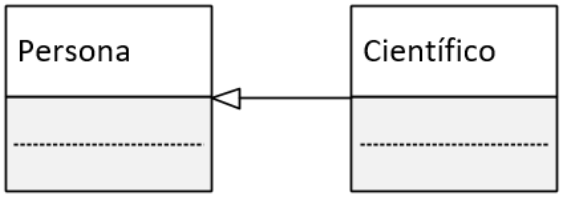
Figura 2-2: Representación de la asociación en un Diagrama de clases



Fuente: Elaboración propia.

- Subtipo o generalización: esta relación describe una jerarquía entre conceptos o clases del sistema (Fowler, 2004). Por ejemplo, un científico es un tipo de persona, como se muestra en la Figura 2-3.

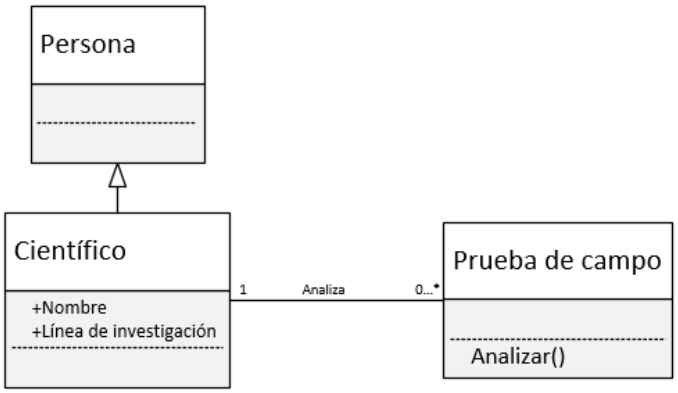
Figura 2-3: Representación de la generalización o Subtipo en un Diagrama de clases



Fuente: Elaboración propia.

Este tipo de diagramas tiene otro tipo de artefactos como los atributos y las operaciones. Los atributos describen una clase o concepto del dominio (Fowler, 2004). En la Figura 2-4 se expresan los atributos Nombre y Línea de investigación para definir las características de la clase Científico y, además, se representa la operación Analizar que se puede ejecutar sobre la clase Prueba de Campo y define su comportamiento.

Figura 2-4: Representación de los atributos y operaciones en Diagrama de clases



Fuente: Elaboración propia.

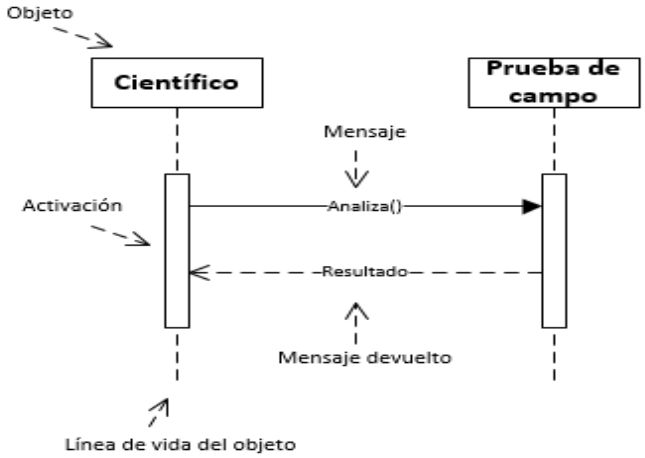
2.3.2 Diagrama de Secuencias

Los Diagramas de Secuencias hacen parte de los Diagramas de Interacción de UML. Su objetivo es describir el comportamiento y las colaboraciones de un grupo de objetos. En el caso particular de los Diagramas de Secuencias, se describe el comportamiento y la comunicación de un escenario específico que conforman varios objetos. Este diagrama muestra un conjunto de representaciones de los objetos y los mensajes que se transmiten (Fowler y Kendall, 1999).

La notación de los Diagramas de Secuencias se compone de: elementos rectangulares que representan a los objetos de las clases presentes en el dominio de aplicación; una línea vertical punteada que representa la línea de vida de cada objeto; un rectángulo vertical que representa la activación de un objeto; una línea horizontal que representa los mensajes que se transmiten entre estos objetos a lo largo de su línea de vida. Adicionalmente, estos mensajes pueden contener parámetros o instrucciones de control que caractericen el mensaje (Fowler, 2004).

En la Figura 2-5 se muestra la representación de los elementos del Diagrama de Secuencias anteriormente descritos. Además, se muestra la interacción que existe entre la clase Científico y la Clase Prueba de campo descrito en el ejemplo de la sección 2.3.1.

Figura 2-5: Representación de los elementos del Diagrama de Secuencias



Fuente: Elaboración propia.

2.4 Esquemas preconceptuales

Los Esquemas Preconceptuales (EP) se concibieron como una forma de representación del dominio de aplicación fácil de elaborar, fácil de entender y muy cercana al lenguaje natural (Zapata *et al*., 2006). Estos modelos representan el conocimiento del interesado y permiten la interacción con analistas de software (Zapata y Arango, 2007).

Para facilitar el entendimiento de ambas partes, los esquemas preconceptuales cuentan con una serie de elementos gráficos (nodos, relaciones, agrupadores y enlaces; Zapata (2012)) y una serie de reglas heurísticas para transformar el EP a algunos diagramas de UML 2.0 como el diagrama de clases, secuencias, máquina de estados y casos de uso. Asimismo, existen reglas que permiten su conversión a diagramas de *OO-Method* y el diagrama de KAOS (*Knowledge Acquisition in Automated Specification;* (Zapata y Arango, 2007; Zapata *et al*., 2006).

2.4.1 Nodos

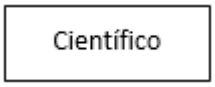
- Concepto: el concepto representa una entidad, un actor o categoría del dominio de aplicación. Lingüísticamente se define como un sustantivo o frase nominal (Zapata y Arango, 2007).

En la Figura 2-6 se presenta la representación gráfica de un Concepto y en la Figura 2-7 se ejemplifica su uso.

Figura 2-6: Representación gráfica de un Concepto en EP.

Fuente: Elaboración propia.

Figura 2-7: Ejemplo de un Concepto en EP.

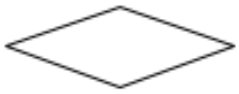


Fuente: Elaboración propia.

- Condicional: describe una expresión que condiciona la ejecución de una tríada dinámica. Esta condición puede contener conceptos y operadores (Zapata y Arango, 2007).

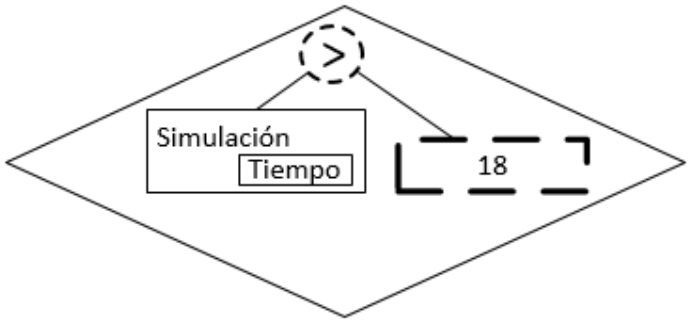
En la Figura 2-8 se puede apreciar la representación gráfica de un condicional y en la Figura 2-9 un ejemplo de su uso.

Figura 2-8: Representación gráfica de un condicional en EP.



Fuente: Elaboración propia.

Figura 2-9: Ejemplo de un condicional en EP.



Fuente: Elaboración propia.

- Operador: el operador en los EP compara o asigna una operación matemática entre dos conceptos (Zapata, 2012).

En la Figura 2-10 se presenta el elemento Operador y en la Figura 2-11 se representa el elemento Asignación. Finalmente en la Figura 2-12 se muestra un ejemplo de su uso en EP.

Figura 2-10: Representación gráfica de un operador.

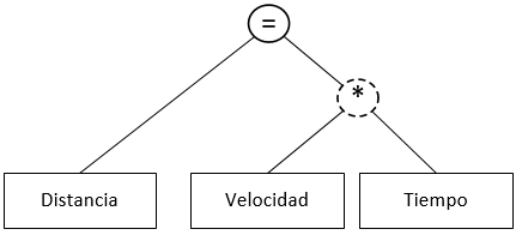
Fuente: Elaboración propia.

- Asignación**:** este elemento asigna un valor específico a un concepto (Zapata, 2012).

Figura 2-11: Representación gráfica del elemento asignación.

Fuente: Elaboración propia**.**

Figura 2-12: Ejemplo del elemento Operador y Asignación en EP.

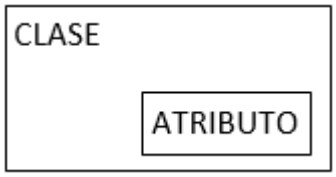


Fuente: Elaboración propia.

- Atributo compuesto: este elemento se refiere al atributo de un concepto específico (Zapata *et al*., 2010).

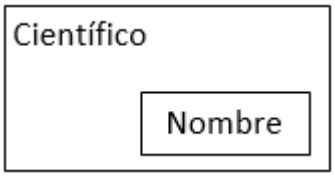
En la Figura 2-13 se representa un atributo compuesto en EP y en la Figura 2-14 se muestra un ejemplo de su uso.

Figura 2-13: Representación gráfica de un atributo compuesto.



Fuente: Elaboración propia.

Figura 2-14: Ejemplo de un atributo compuesto en EP.



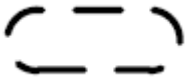
Fuente: Elaboración propia.

2.4.2 Relaciones

- Relación Dinámica: las relaciones dinámicas denotan acciones, operaciones y funciones del dominio de aplicación. Se representan lingüísticamente como verbos de acción o ejecución (Zapata, 2012).

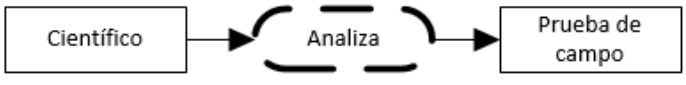
La representación y un ejemplo de la Relación Dinámica se puede ver en la Figura 2-15 y la Figura 2-16.

Figura 2-15: Representación de la Relación Dinámica en EP.



Fuente: Elaboración propia.

Figura 2-16: Ejemplo de la tríada Dinámica en EP.

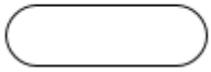


Fuente: Elaboración propia.

- Relación Estructural: esta relación describe los verbos “Tiene” y “Es” y generan una dependencia entre los conceptos que une (Zapata y Arango, 2007).

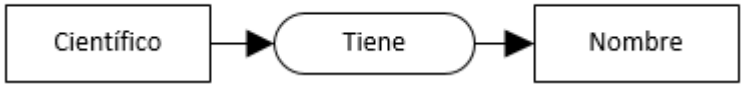
En la Figura 2-17 se presenta la representación gráfica de una Relación Estructural y en la Figura 2-18 se muestra un ejemplo del uso de la Relación Estructural en EP.

Figura 2-17: Representación gráfica de una Relación Estructural.



Fuente: Elaboración propia.

Figura 2-18: Ejemplo de una Relación Estructural en EP.



Fuente: Elaboración propia.

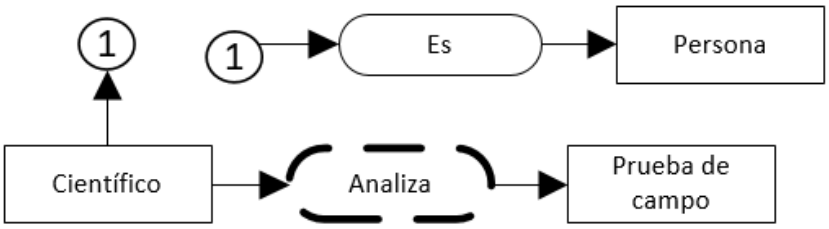
- Referencia: este elemento une conceptos y relaciones distantes dentro del EP (Zapata, 2012).

En la Figura 2-19 y la Figura 2-20 se muestran una representación gráfica y un ejemplo de la referencia en los EP respectivamente.

Figura 2-19: Representación gráfica de una referencia en EP.

Fuente: Elaboración propia.

Figura 2-20: Ejemplo de una referencia en EP.



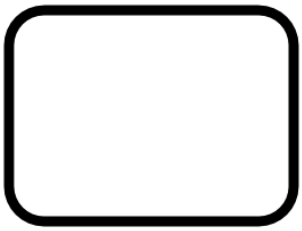
Fuente: Elaboración propia.

2.4.3 Agrupadores

- Marco: el marco permite agrupar relaciones dinámicas y asignarlas como responsabilidades (Zapata y Chaverra, 2012).

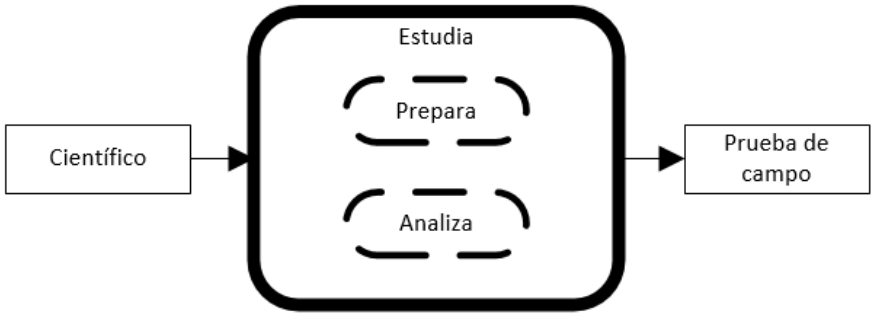
En la Figura 2-21 se muestra la representación gráfica del Marco y en la Figura 2-22 se presenta un ejemplo de su uso.

Figura 2-21: Representación gráfica del Marco.



Fuente: Elaboración propia.

Figura 2-22: Ejemplo del Marco en EP.

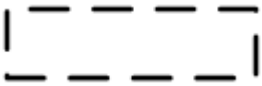


Fuente: Elaboración propia.

- Valor: este elemento le asigna los diferentes valores que puede tomar un concepto del dominio de aplicación (Zapata, 2012).

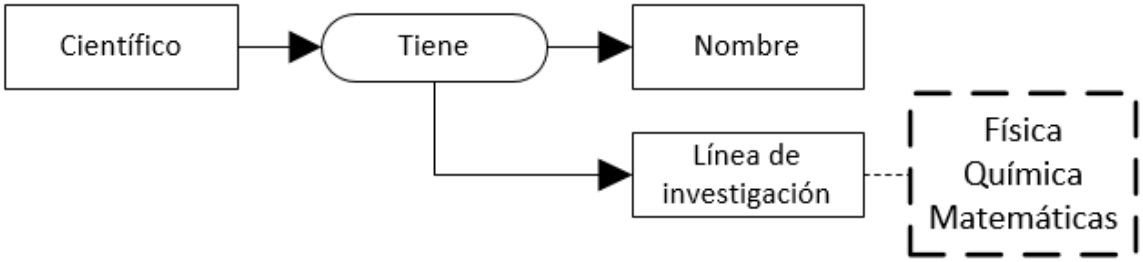
En la Figura 2-23 se presenta la representación gráfica del elemento Valor y en la Figura 2-24 se expresa un ejemplo de su uso en EP.

Figura 2-23: Representación gráfica del Valor en EP.



Fuente: Elaboración propia.

Figura 2-24: Ejemplo del elemento Valor en EP.



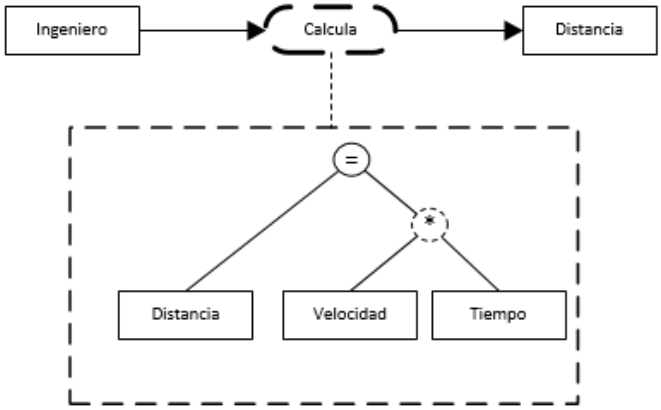
Fuente: Elaboración propia.

- Especificación: este elemento permite caracterizar las operaciones que tienen lugar en una tríada dinámica (Zapata, 2012). Su representación gráfica se expresa en Figura 2-25 y un ejemplo de su uso se presenta en la Figura 2-26.

Figura 2-25: Representación gráfica de una Especificación en EP.

Fuente: Elaboración propia.

Figura 2-26: Ejemplo del uso del elemento Especificación en EP.



Fuente: Elaboración propia.

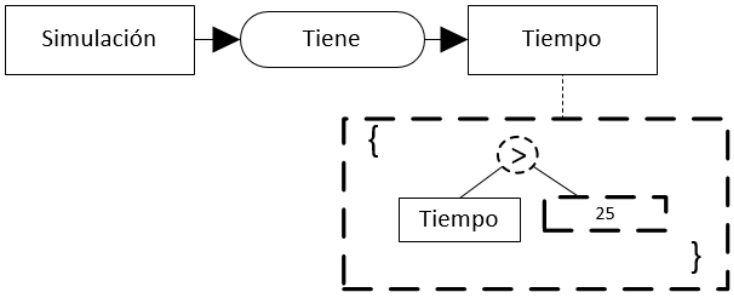
- Restricción: este elemento describe restricciones o características de la especificación de una tríada dinámica o de un concepto (Zapata y Chaverra, 2012).

En la Figura 2-27 se muestra la representación gráfica de la Restricción y en la Figura 2-28 se ejemplifica su uso en EP.

Figura 2-27: Representación gráfica de la Restricción en EP.

Fuente: Elaboración propia.

Figura 2-28: Ejemplo de una Restricción en EP.

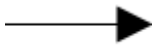


Fuente: Elaboración propia.

2.4.4 Enlaces

- Conexión: este elemento conecta conceptos a relaciones dinámicas y estructurales y permiten crear tríadas (Zapata, 2012). En la Figura 2-29 se muestra su representación gráfica.

Figura 2-29: Representación gráfica de una Conexión en EP.



Fuente: Elaboración propia**.**

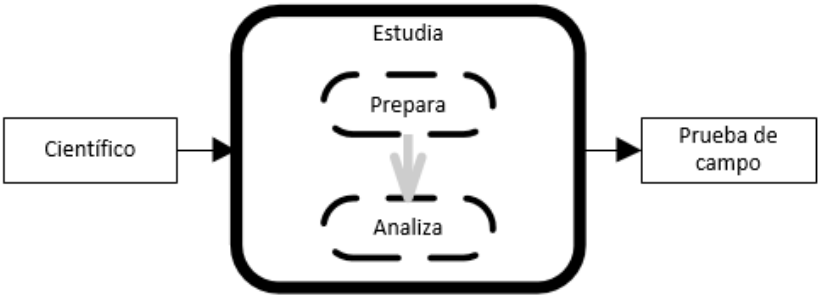
- Implicación: esta relación representa una relación causa-efecto entre tríadas dinámica o entre tríadas dinámicas y condicionales. La secuencia muestra el orden en que se deben ejecutar las operaciones (Zapata y Arango, 2007).

En la Figura 2-30 se puede observar la representación gráfica de la implicación y en la Figura 2-31 se puede ver un ejemplo de su uso en EP.

Figura 2-30: Representación gráfica de la Implicación en EP.

Fuente: Elaboración propia.

Figura 2-31: Ejemplo del uso de una Implicación en EP.



Fuente: Elaboración propia.

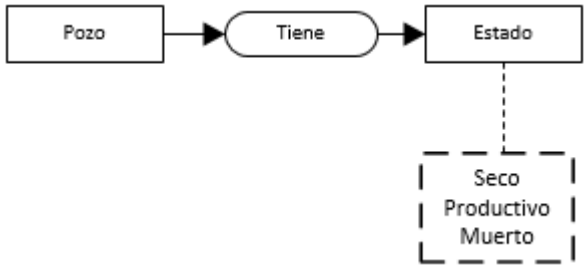
- Realización: este elemento permite conectar notas, especificaciones y restricciones a conceptos y relaciones dinámicas (Zapata, 2012).

En la Figura 2-32 se puede observar la representación gráfica de la relación Realización y en la Figura 2-33 un ejemplo de su uso.

Figura 2-32: Representación gráfica de la relación Realización en EP.

Fuente: Elaboración propia.

Figura 2-33: Ejemplo de la relación Realización en EP.



Fuente: Elaboración propia.

2.5 Patrones de diseño

La concepción de los patrones de diseño viene de la arquitectura y construcción de edificaciones, donde Alexander *et al.* (1977) definen que cada patrón describe un problema que ocurre una y otra vez en el tiempo y el elemento principal de una solución que se puede reutilizar sin tener que hacer todo de nuevo.

En software, los patrones de diseño nacen como una extensión de métodos de análisis y diseño orientados a objetos, para tratar de unificar las soluciones a problemas recurrentes en el desarrollo cotidiano de aplicaciones de software (Gardner, 2004). Estos patrones de diseño conservan la misma esencia que los de Alexander *et al.* (1977), sólo que, en vez de elementos de arquitectura y construcción de edificaciones, se tienen objetos e interfaces que representan el sistema que se construye (Gamma *et al*., 1994).

Gamme *et al.* (1994) introducen la primera formalización de los patrones de diseño, al definir los elementos principales de un patrón de diseño genérico:

- **El nombre del patrón:** define en una palabra o dos el problema de diseño, la intención de la solución y las consecuencias de su uso.

- **El problema:** define las condiciones del problema que soluciona, en algunos casos mediante listas de chequeo o la definición de un contexto.
- **La solución:** describe las entidades, relaciones y responsabilidades que componen la solución. Usualmente, se representa a partir de diagramas de UML e incluye una plantilla que se puede aplicar a varios contextos o dominios de aplicación.
- **Las consecuencias:** describen el resultado de un diseño basado en el patrón de diseño y cómo afecta al resto del diseño del sistema.

Los patrones de diseño se definieron y crearon especialmente para paradigmas orientados a objetos. El desempeño e implementación dependen directamente del lenguaje de programación y, en algunos casos, del contexto donde se usen (Shalloway y Trott, 2004).

La representación de estos patrones se basa, en la mayoría de los casos, en notaciones gráficas usando diagramas de UML y, en algunas ocasiones, en otro tipo de grafos. Además, la definición de cada patrón de diseño debe contener unas secciones de soporte para facilitar el entendimiento de dicho patrón: el nombre del patrón de diseño y su clasificación, motivación, aplicabilidad, estructura, participantes, colaboraciones y consecuencias. Finalmente, esta representación sólo expresa el estado final del patrón de diseño aplicado y sólo se enfoca en la estructura resultante de la implementación (Gamma *et al*., 1994).

Gamma *et al.*(1994) definen, en total, 23 patrones de diseño donde se resaltan las mayores premisas del diseño de software: composición sobre herencia, bajo acoplamiento, alta cohesión y correcta asignación de responsabilidades. Estos patrones de diseño se distribuyen en tres categorías: patrones de diseño estructurales, de comportamiento y de creación.

2.5.1 Patrones de diseño de creación

Los Patrones de Diseño de Creación se enfocan en hacer un sistema independiente de la manera en que sus objetos y clases se crean, componen y representan. Estos patrones resumen el proceso de creación o instanciación de estos elementos (Gamma *et al*., 1994). Existen dos aproximaciones: a nivel de clases y a nivel objetual. Los patrones de diseño de creación a nivel de clases usan la generalización para variar la clase que se está

instanciando. Por otro lado, a nivel objetual, se delega el proceso de instanciación de un objeto a otro (Shalloway y Trott, 2004).

Los patrones de diseño de creación se enfocan principalmente en el principio de diseño de composición sobre herencia, el cual toma relevancia a medida que el sistema evoluciona y crece (Gamma *et al*., 1994). Los patrones de diseño de creación se listan y definen (Pree y Sikora, 1995; Stelting y Maassen, 2002) a continuación:

- ***Abstract Factory*:** este patrón de diseño provee una interfaz o una clase abstracta, la cual define la creación de una familia de objetos relacionados o dependientes sin especificar su clase de origen.
- ***Builder*:** el *Builder* se enfoca en la creación de objetos complejos. Este patrón separa el proceso de construcción del objeto de su implementación, logrando así usar el mismo proceso de creación para otras representaciones del mismo objeto.
- ***Factory Method*:** el patrón *Factory Method* define una interfaz genérica para la creación de un objeto. Este patrón de diseño trabaja a nivel de clases y le permite a una subclase instanciar la clase adecuada para el contexto definido.
- ***Prototype*:** este patrón define una estructura deseada para el objeto y permite instanciar objetos haciendo una copia de éste.
- ***Singleton*:** el patrón *Singleton* restringe el número de instancias de una clase a sólo una en todo el sistema.

2.5.2 Patrones de diseño estructurales

Los Patrones de Diseño Estructurales describen maneras de componer o crear objetos para realizar nuevas funcionalidades (Stelting y Maassen, 2002). Este tipo de patrones se enfoca en la manera en que las clases y objetos se unen para formar estructuras más complejas. La principal característica de los patrones de diseño estructurales es que le permiten a un objeto variar su comportamiento y composición en tiempo de ejecución, aportando gran flexibilidad al sistema (Gamma *et al*., 1994). Los patrones de diseño estructurales se definen según Gamma *et al.* (1994) a continuación:

- ***Adapter*:** el patrón *Adapter* le permite a un cliente trabajar con clases cuya interfaz no posee una interfaz permitida en un sistema específico. Este patrón adapta la interfaz del objeto externo al de las clases del sistema al que se integra.
- ***Bridge*:** este patrón desacopla la abstracción de una clase de su implementación, permitiendo así, que varíen independientemente y se entregue flexibilidad y facilidad de mantenimiento al software.
- ***Composite:*** el patrón *Composite* compone objetos relacionados en una jerarquía de tipo árbol. Este patrón de diseño le permite al cliente tratar de la misma manera un objeto y un grupo de objetos.
- ***Decorator:*** el patrón de diseño *Decorator* le añade responsabilidades adicionales a un objeto en tiempo de ejecución.
- ***Flyweight:*** este patrón de diseño permite apoyarse en la copia de objetos complejos en un sistema que requiere la instanciación de un gran número de objetos con una estructura similar.
- ***Facade:*** este patrón provee una interfaz que representa las interfaces de los subsistemas del software. Además, facilita el uso de un sistema complejo con muchas relaciones directas al cliente.
- ***Proxy:*** el patrón *Proxy* restringe el acceso a un objeto para evitar su creación o participación precoz en el sistema. Usualmente, trata a objetos complejos o de gran tamaño.

2.5.3 Patrones de Diseño de Comportamiento

Los patrones de Diseño de Comportamiento se enfocan en la asignación de responsabilidades entre clases y objetos (Shalloway y Trott, 2004). Estos patrones determinan flujos de control y comunicación complejos entre objetos que de otra forma, sería complejo seguir en tiempo de ejecución (Stelting y Maassen, 2002).

Estos patrones de diseño distribuyen, a nivel de clase, el comportamiento y las responsabilidades. Por otro lado, a nivel objetual, se describe la colaboración entre objetos para cumplir con una tarea común (Gamma *et al*., 1994). La descripción de estos patrones (Gamma *et al*., 1994; Stelting y Maassen, 2002) se lista a continuación:

- ***Chain of responsibility:*** el propósito de este patrón de diseño es evitar el acoplamiento entre el objeto receptor y emisor de un mensaje específico. El patrón transfiere la responsabilidad a un objeto disponible.

- ***Command:*** este patrón encapsula una petición como un objeto para permitir su parametrización.
- ***Interpreter:*** el patrón *Interpreter*, dado un lenguaje, permite definir una representación para su gramática y solucionar el mismo problema recurrente desde el contexto al que pertenece.
- ***Iterator*****:** este patrón de diseño proporciona una manera de acceder a un objeto agregado sin exponer su estructura interna.
- ***Mediator:*** el patrón de diseño *Mediator* permite el bajo acople entre un conjunto de clases asignándole la implementación de los métodos a una sola clase.
- ***Memento*****:** este patrón de diseño captura y externaliza el estado interno de un objeto para que éste pueda variar y volver al estado inicial en cualquier tiempo.
- ***State:*** el patrón de diseño *State* permite a un objeto cambiar su comportamiento con base en los cambios de su estado interno.
- ***Strategy:*** este patrón de diseño permite encapsular algoritmos o funcionalidades específicas y hacerlas intercambiables. Este patrón de diseño le permite al algoritmo variar sin depender del cliente.
- ***Template Method:*** el patrón de diseño *Template Method* define el esqueleto de un algoritmo y permite cambiar ciertos pasos del mismo en las subclases. Estos cambios no afectan al algoritmo original.
- ***Visitor:*** este patrón de diseño permite adicionar funcionalidades a una clase existente sin afectar su estructura.
- ***Observer:*** el patrón de diseño *Observer* define una dependencia de uno a muchos entre un conjunto de objetos para actualizarlos automáticamente una vez el objeto principal cambie su estado interno.

3. Antecedentes

3.1 Patrones de Diseño en Software Científico

El software científico es, usualmente, complejo y demasiado específico para la reutilización de sus componentes (Kelly *et al.*, 2011). Por otro lado, los científicos desarrollan este tipo de software y sólo se preocupan por la eficiencia y la precisión de sus resultados (Li, *et al.*, 2011).

La inclusión de la POO (programación orientada a objetos) en el software científico se hace con el fin de aportarle robustez, facilidad de mantenimiento y flexibilidad al sistema (Gardner, 2004; Segal y Morris, 2008). En la literatura se encuentran propuestas que muestran las primeras aproximaciones de este paradigma al software científico como poco convenientes, debido al gran impacto que genera sobre la eficiencia de aplicaciones de este tipo (Charles, 2002; Cickovski *et al.*, 2004; Rodriguez *et al.*, 2004; Segal, 2005). Por ello, se introducen los patrones de diseño en el desarrollo de software científico como una aproximación más completa a la POO. Con la implementación de estos patrones se mitiga este impacto y se mejoran algunos aspectos de calidad del software, como la flexibilidad, la facilidad de mantenimiento y la escalabilidad (Gardner, 2004; Gardner y Manduchi, 2007; Segal, 2005).

Gama *et al.* (1994) presentan una aproximación convencional a los patrones de diseño en áreas de la física, la medicina, la simulación de flujo, las matemáticas, la industria militar y la industria nuclear, entre otros. Además, se proponen algunos patrones de análisis para áreas más específicas de la ciencia (Hemert y Barker, 2008; Karastoyanova y Andrikopoulos, 2013; Kelly *et al*., 2011; Li *et al.*, 2015; Panagiotis y Margaritis, 2014; Rasmussen *et al*., 2014; Sabatucci *et al.*, 2015; Segal, 2008; Wagner, 2007). Los patrones de análisis expresan el problema a resolver y los patrones de diseño revelan la manera de resolverlo (Gardner y Manduchi, 2007). En las Tablas 3-1 a 3-3 se puede observar un

resumen de la revisión de la literatura sobre los patrones de diseño que utilizan diferentes autores en software científico.

Tabla 3-1 Resumen de la revisión de la literatura en patrones de diseño de creación en software científico.

Autor(es)	***Abstract Factory***	***Builder***	***Factory Method***	***Prototype***	***Singleton***
Gardner (2004)	x	X	x		
Cickovski *et al.* (2004)	x		x	x	x
Decyk *et al.* (2006)	x		x		
Gardner *et al.* (2007)	x	X	x		
Aras *et al.* (2004)			x		
Hartmanis *et al.* (2001)	x	X			
Barbieri *et al.* (2012)		X			

Tabla 3-2 Resumen de la revisión de la literatura en patrones de diseño estructural en software científico

Autor(es)	***Adapter***	***Bridge***	***Composite***	***Decorator***	***Facade***	***Flyweight***	***Proxy***
Charles (2002)		x					
Gardner (2004)	x		x		x		x
Cickovsky *et al.* (2004)		x					x
Decyk *et al.* (2006)	x				x		
Pérez *et al.* (2011)					x		
Aras *et al.* (2004)	x				x		
Rodriguez *et al.* (2004)					x		
Cuéllar *et al.* (2005)		x			x		
Wagner (2007)							x
Dodig-crnkovic(2002)			x	x			
Panagiotis y Margaritis (2014)				x			
Gardner *et al.* (2007)	x			x	x		x
Decyk *et al.* (2007)			x				

Tabla 3-3 Resumen de la revisión de la literatura en patrones de diseño de comportamiento en software científico

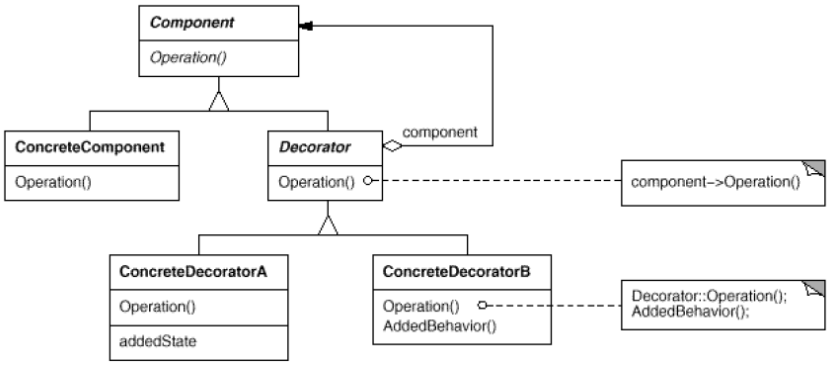
Autor(es)	***Chain of responsibility***	***Command***	***Interpreter***	***Iterator***	***Mediator***	***Memento***	***Observer***	***State***	***Strategy***
Gardner (2004)				x	x			x	
Cickovsky *et al.* (2004)								x	x
Decyk *et al.* (2006)									x
Decyk *et al.* (2007)									x
Aras *et al.* (2004)									x
Pérez *et al.* (2011)									x
Hartmanis *et al.* (2001)									x
Gardner *et al.* (2007)							x	x	
Davis (2012)								x	
Panagiotis *et al.* (2014)								x	x

3.2 Representación de los Patrones de Diseño

Gamma *et al.* (1994) proponen la primera representación de los 23 Patrones de Diseño en su libro introductorio. Esta representación se basa en técnicas de modelado objetual (OMT por su sigla en inglés) que, posteriormente, se llevaron a la notación convencional de diagramas de clases y, en algunos casos, de otros diagramas de UML que apoyan la representación principal, mostrando la dinámica del mismo.

En la Figura 3-1 se muestra la representación de uno de los patrones de diseño estructurales en diagrama de clases.

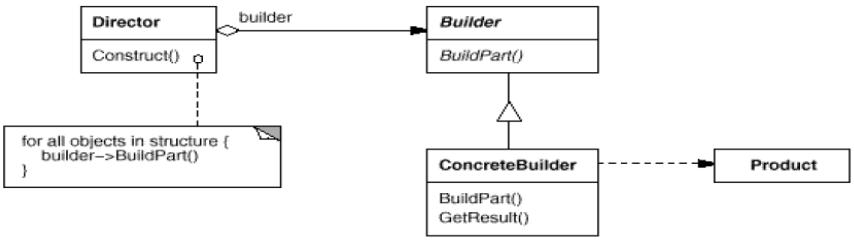
Figura 3-1: Representación en diagrama de clases del patrón de diseño *Decorator.*



Fuente: Tomada de Gamma *et al.* (1994).

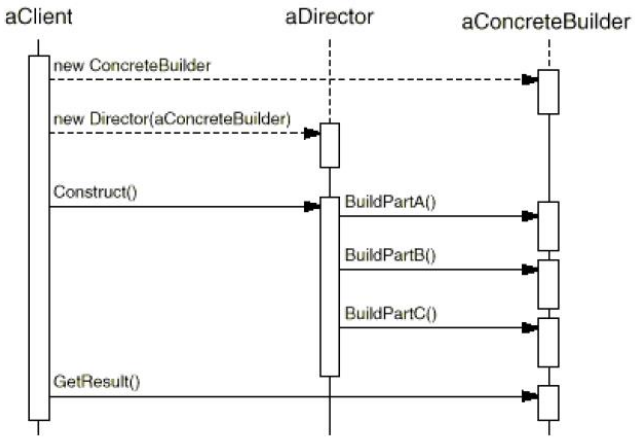
En algunos casos, las representaciones contienen diagramas de UML de apoyo que ejemplifican la comunicación entre los componentes del patrón de diseño. En la Figura 3-2 se muestra la representación común del patrón de diseño *Builder* y en la Figura 3-3 se muestra un diagrama de secuencias que muestra cómo se comunican los elementos de este patrón de diseño según Gamma *et al.* (1994).

Figura 3-2: Representación en diagrama de clases del patrón de diseño *Builder.*



Fuente: Tomada de Gamma *et al.* (1994).

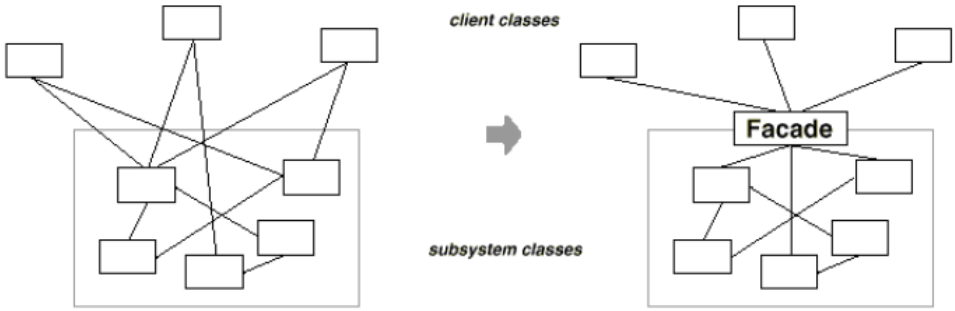
Figura 3-3: Representación de apoyo en diagrama de secuencias del patrón de diseño *Builder.*



Fuente: Tomada de Gamma *et al.* (1994).

Algunas representaciones se basan en otro tipo de grafos que intentan explicar de forma genérica la arquitectura del patrón. En la Figura 3-4 se expresa una representación alternativa del patrón de diseño *Facade* que proponen Gamma *et al.* (1994).

Figura 3-4: Representación tipo grafo del patrón de diseño *Facade*



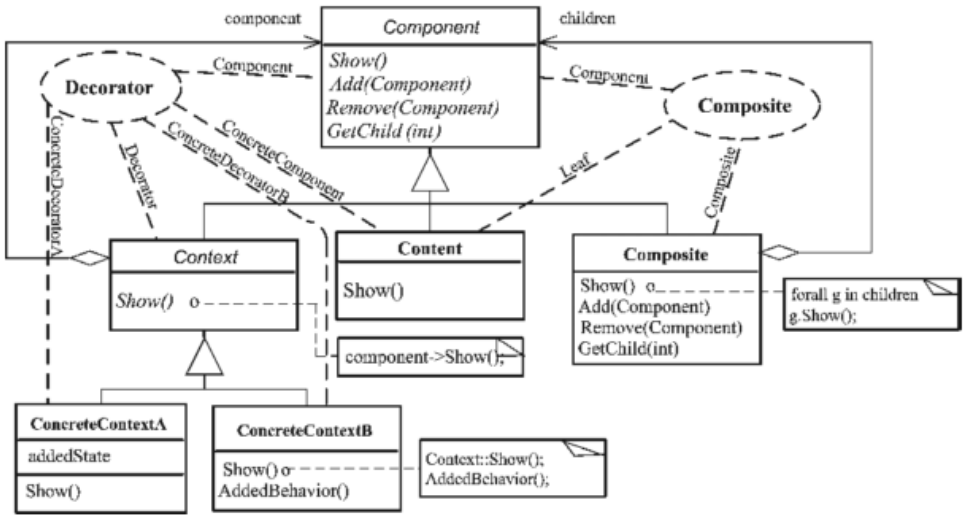
Fuente: Tomada de Gamma *et al.* (1994).

Las representaciones que proponen Gamma *et al.* (1994) se acompañan con una descripción textual del propósito general del patrón, un ejemplo de implementación, los patrones relacionados, los elementos que intervienen en el patrón de diseño, su papel en el diseño y, en algunas ocasiones, la manera de interacción y sus componentes.

Stelting *et al.* (2002) proponen una representación basada en los mismos diagramas de UML pero varían los elementos de la descripción textual. Estos autores expresan ventajas y desventajas de usar cada patrón de diseño y variaciones de algunos patrones. Por otro lado, Shalloway *et al.* (2004) proponen algunas claves de diseño y análisis que pretenden identificar cuándo se usa el patrón, una de las críticas más fuertes que exhibe la representación convencional de dichos patrones.

Dong (2002) propone una representación con notaciones de colaboración en UML. En la Figura 3-5 se ejemplifica el uso de esta notación. Asimismo, otros autores proponen algunas notaciones basadas en otros componentes de UML (Dong, 2003; Dong y Yang, 2003; Dong, Yang, y Zhang, 2007). En la Figura 3-6 se presenta la notación de colaboración de UML y en la Figura 3-7 se expresa la notación de roles de UML para los patrones de diseño *Composite* y *Decorator*.

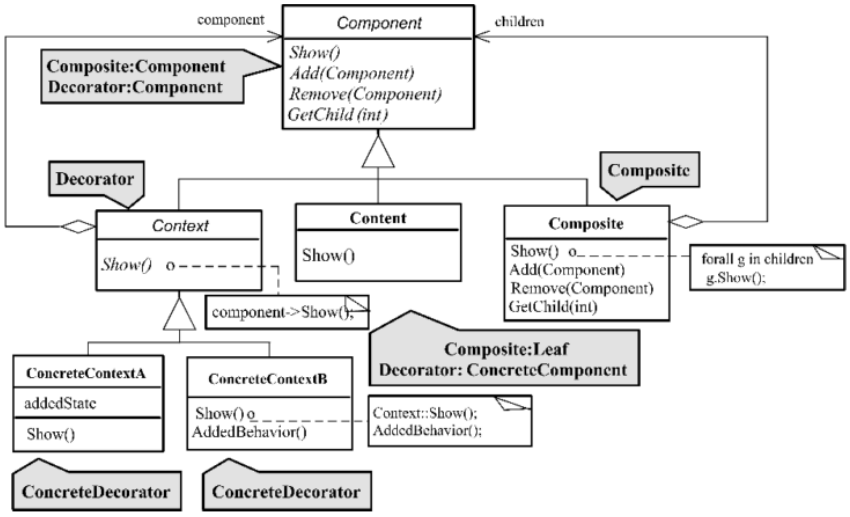
Figura 3-5: Representación de los patrones de diseño *Composite* y *Decorator* en notación de colaboración de UML.



Fuente: Tomada de Dong (2002).

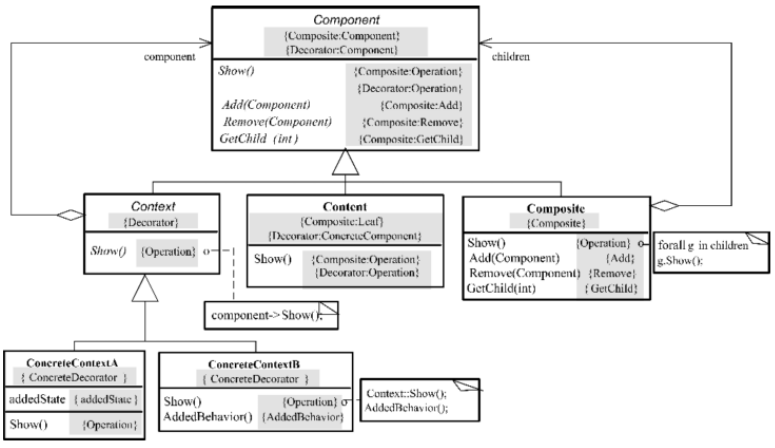
Otros autores proponen variaciones de los patrones de diseño y ejemplifican la aplicabilidad de estos patrones en contextos científicos (Decyk y Gardner, 2007; Gardner, 2004; Hemert y Barker, 2008; Kelly y Sanders, 2008; Roure y Goble, 2009; Segal, 2008). Todas las representaciones presentadas se basan en las representaciones convencionales con diagramas de UML.

Figura 3-6: Representación de los patrones de diseño *Composite* y *Decorator* en notación de roles de UML.



Fuente: Tomada de Dong (2003).

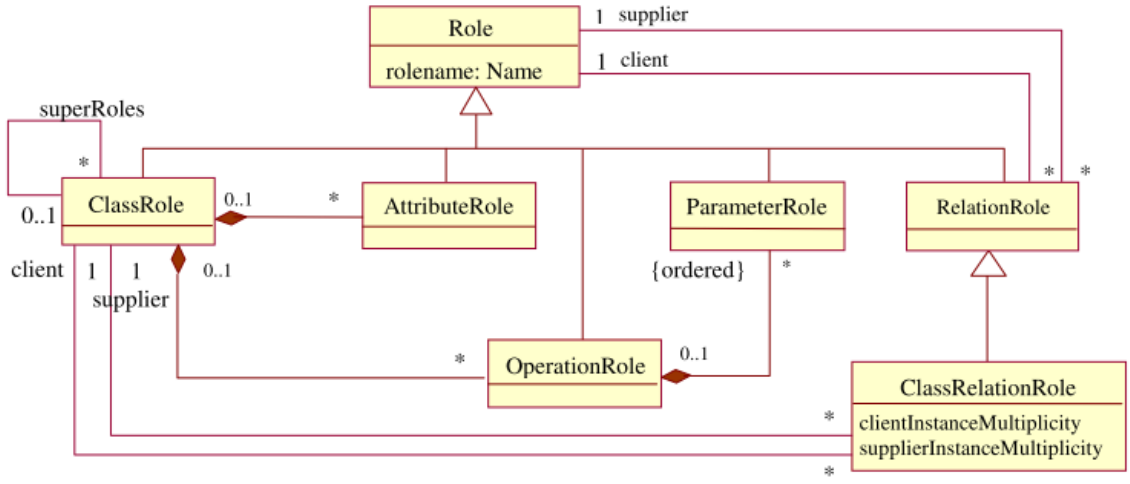
Figura 3-7: Representación de los patrones de diseño *Composite* y *Decorator* en notación de etiquetado de UML.



Fuente: Tomada de Dong *et al.* (2007).

Algunos autores proponen metamodelos para incentivar la implementación y combinación de los patrones de diseño (Kim *et al.*, 2003; Smith y Stotts, 2002). En la Figura 3-8 se presenta un ejemplo del concepto de rol en los patrones de diseño, cuyo objetivo es definir la participación de un concepto en el patrón (Kim y Carrington, 2004).

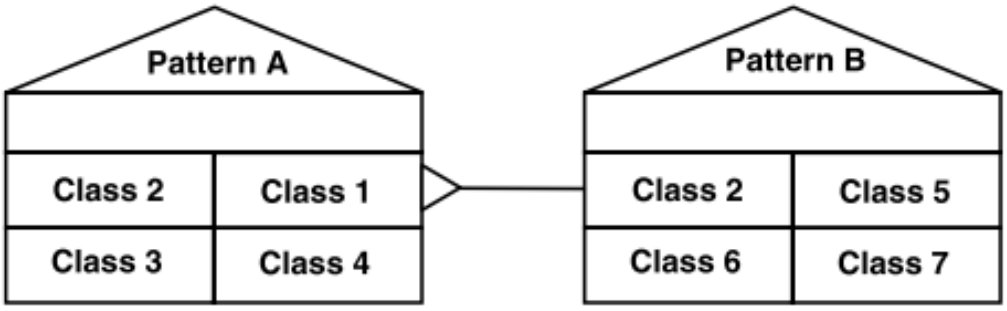
Figura 3-8: Representación de un metamodelo de rol para definir los patrones de diseño.



Fuente: Tomada de Kim *et al.* (2004).

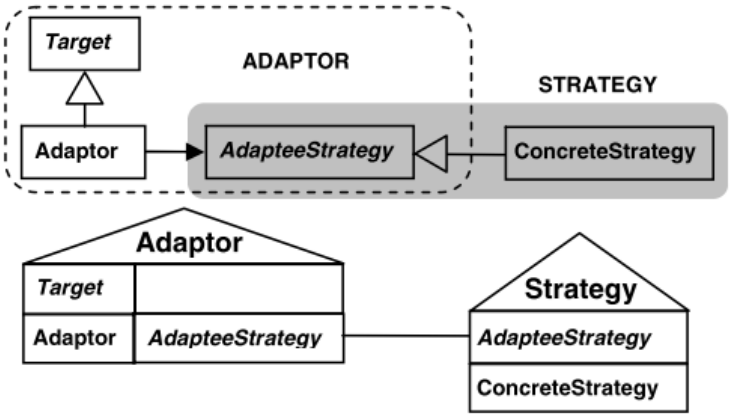
Mustapha *et al.* (2011) introduce un lenguaje bajo UML dedicado a los patrones de diseño, con el objetivo de aportar una manera más legible y completa de representar a los patrones de diseño. En la Figura 3-9 se muestran los elementos gráficos del lenguaje dedicado y en la Figura 3-10 se muestra un ejemplo de conversión de los patrones *Adapter* y *Strategy*.

Figura 3-9: Elementos gráficos del lenguaje dedicado a patrones de diseño.



Fuente: Tomado de Mustapha *et al.* (2011).

Figura 3-10: Ejemplo de lenguaje dedicado a patrones de diseño con los patrones *Adapter* y *Strategy*.

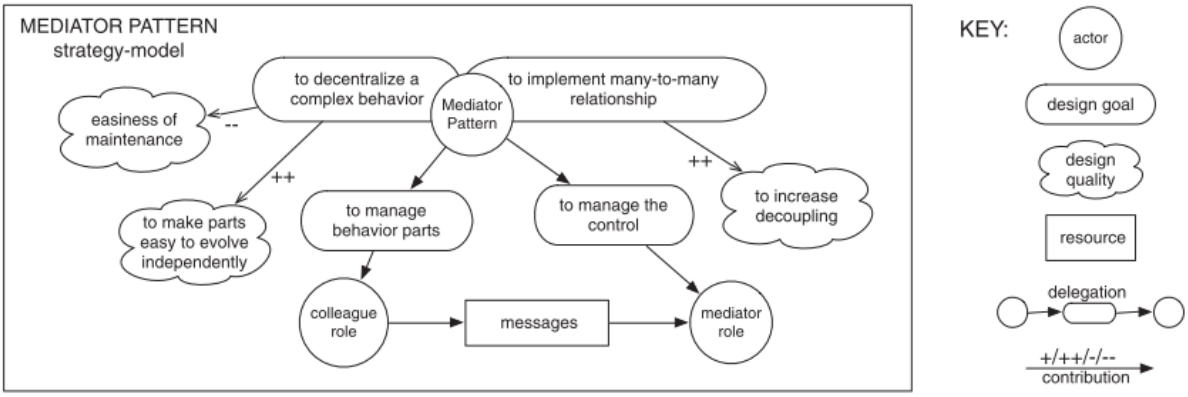


Fuente: Tomado de Mustapha *et al.* (2011).

Algunos autores proponen una especificación sólo textual de los patrones de diseño. Estos enfoques se centran en la especificación del problema que resuelven los patrones de diseño y responden a cuál es el contexto y qué indicadores muestran un posible uso de los patrones (Bonfè *et al.*, 2013; Clune *et al.*, 2012; Hasheminejad y Jalili, 2012; Seffah, 2015).

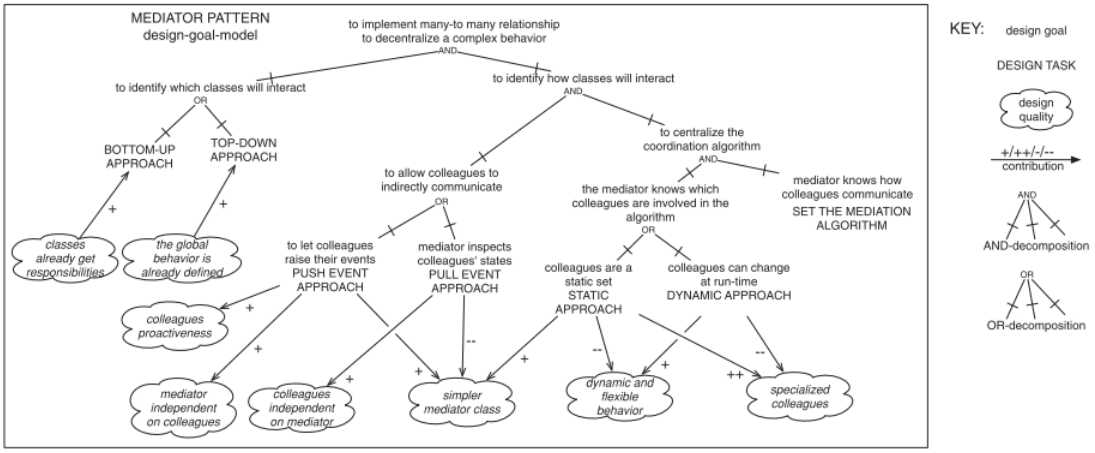
Sabatucci *et al.* (2015) proponen representaciones basadas en estrategias y objetivos. En la Figura 3-11 se muestra un ejemplo de la representación basada en estrategia y los elementos que la componen. Por otro lado, en la Figura 3-12 se ejemplifica el patrón *Mediator* con la aproximación por objetivos y los elementos que la componen.

Figura 3-11: Ejemplo de la representación del patrón *Mediator* basado en estrategia.



Fuente: Tomado de Sabatucci *et al.* (2015).

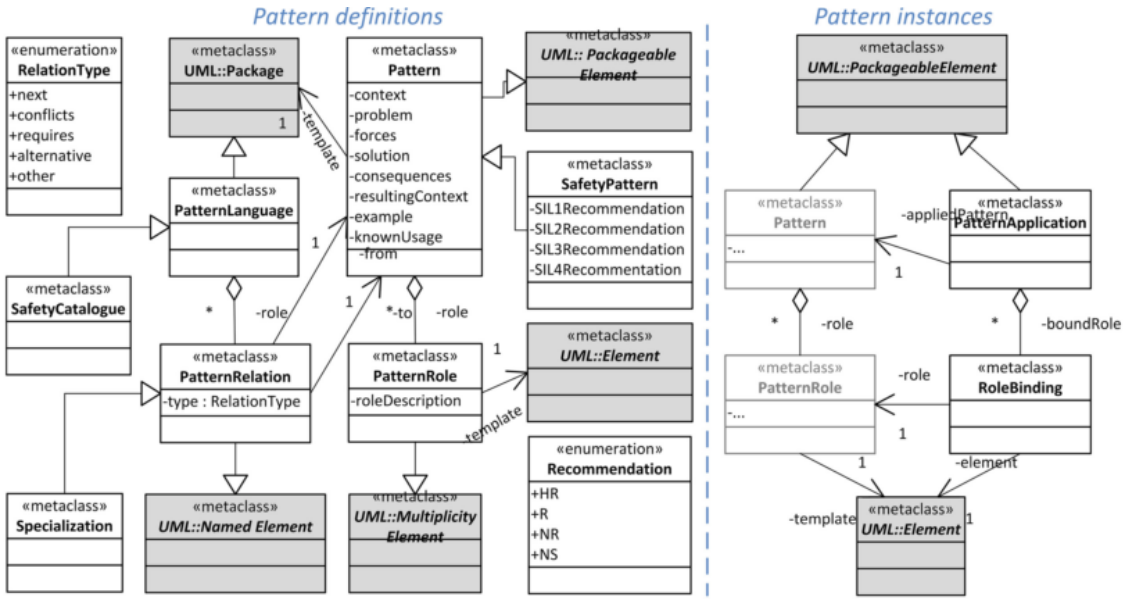
Figura 3-12: Ejemplo de la representación del patrón *Mediator* basada en objetivos.



Fuente: Tomado de Sabatucci *et al.* (2015).

Winetzhammer *et al.* (2015) proponen la representación de los patrones de diseño con base en un metamodelo que especifica la definición y la instanciación de los patrones. En la Figura 3-13 se representan los elementos UML y objetuales para la representación genérica del metamodelo.

Figura 3-13: Representación genérica de las definiciones e instancias de los patrones de diseño a partir de un metamodelo.



Fuente: Tomada de Winetzhammer *et al.* (2015).

En resumen, las representaciones para los patrones de diseño que se encuentran en la literatura, siguen los mismos lineamientos desde su definición en Gamma *et al.* (1994). Estas representaciones sólo se enfocan en la representación del patrón de diseño implementado y además, dejan a un lado una serie de operaciones internas que son necesarias para caracterizar completamente el patrón.

En las representaciones y ejemplos de la literatura, se encuentran inconsistencias cuando se llevan las representaciones a código. En la mayoría de los casos, hay código que complementa la implementación pero no se presentó en el diseño inicial. Estas aproximaciones, confunden el propósito y la arquitectura del patrón de diseño.

Una de las mayores dificultades encontradas, es la poca información que se presenta para definir el problema que intenta resolver el patrón de diseño. Usualmente, esta información es textual y, en algunas ocasiones, no representa una situación genérica. Al no tener bien caracterizado el problema que se resuelve, la implementación de los patrones de diseño, pueden llevar a implementaciones peores o nocivas para el sistema (Shalloway y Trott, 2004).

4. Propuesta de solución

En esta Sección se propone un conjunto de nuevos elementos para los EP que facilitan la representación y comunicación de un dominio de aplicación complejo y, además, una nueva aproximación a la representación de los patrones de diseño y su uso en contextos científicos.

Esta Sección se estructura así: en la Sección 4.1 se proponen nuevos elementos para los EP para representar dominios complejos y las reglas para la obtención de código a partir de ellos. En la Sección 4.2 se proponen las representaciones de los patrones de diseño que más se usan en software científico en esquemas preconceptuales y, adicionalmente, las representaciones genéricas de los problemas que soluciona cada patrón.

4.1 Nuevos elementos para los Esquemas Preconceptuales

En esta Sección se presentan algunos elementos para los EP que complementan los elementos recopilados en Zapata *et al.* (2012) para la completa representación y comunicación de dominios científicos. Estos nuevos elementos aportan mayor flexibilidad y características del paradigma POO a los EP.

4.1.1 Concepto tipo arreglo

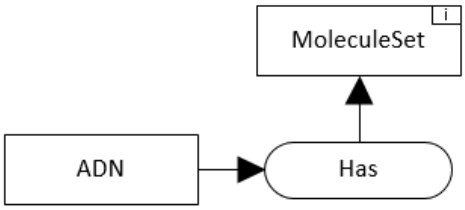
Los conceptos en los EP se limitan a un valor individual. En contextos científicos se hace necesario el uso de estructuras que permitan almacenar uno o varios elementos. Para la representación de los conceptos tipo arreglo se propone la representación del concepto tradicional de Zapata *et al.* (2007), más un pequeño rectángulo en su esquina superior derecha con un índice “i” que indique la posibilidad de iteración sobre éste.

En la Figura 4-1 se muestra el símbolo del concepto tipo arreglo y en la Figura 4-2 se ejemplifica su uso.

Figura 4-1: Representación gráfica de un concepto tipo arreglo en EP.

Fuente: Elaboración propia.

Figura 4-2: Ejemplo de un concepto tipo arreglo en EP.



Fuente: Elaboración propia.

En algunas ocasiones, estos arreglos pueden almacenar algunos datos predefinidos. Para representar un conjunto de datos predefinidos se debe usar el elemento nota de Zapata (2012) para encapsular una tabla con el mismo nombre del concepto, una columna con los índices “i” y otra con los valores que están almacenados en el arreglo.

En la Figura 4-3 se presenta un conjunto de datos predefinidos para el concepto tipo arreglo “*MoleculeSet*” del ejemplo de la Figura 4-2. Cabe resaltar que los elementos de dicho arreglo se pueden repetir como se observa en los índices 2 y 4.

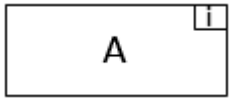
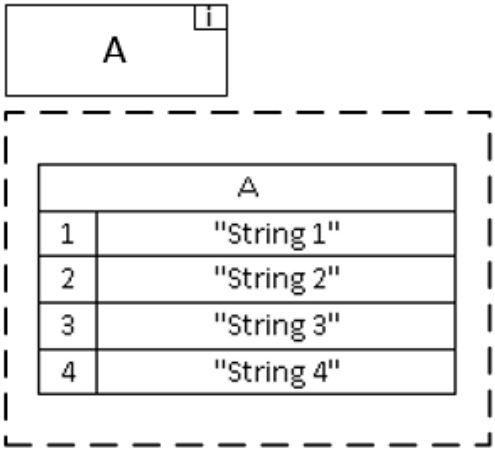
Figura 4-3: Representación de valores predefinidos de un concepto tipo arreglo.

MoleculeSet	
1	Phosphoric Acid
2	Deoxyribose
3	Nitrogen base
4	Deoxyribose

Fuente: Elaboración propia.

En la Tabla 4-1 se presentan las reglas para obtener el código en Java a partir del Concepto tipo arreglo y su variación con datos predefinidos.

Tabla 4-1: Reglas para la obtención de código a partir del concepto tipo arreglo.

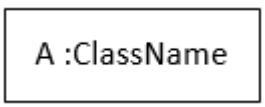
Concepto tipo arreglo	Código Java
	ArrayList A = new ArrayList();
	ArrayList A = new ArrayList(); A.add(“String 1”); A.add(“String 2”); A.add(“String 3”); A.add(“String 4”);

4.1.2 Concepto tipo clase

Zapata (2012) introduce tipos de datos para los conceptos. Usualmente, en los PD y en los paradigmas orientados a objetos es necesario definir un concepto de tipo clase para explotar algunas características de la herencia de POO. El tipo de los conceptos tipo clase sólo puede tomar valores de los conceptos clase modelados en el dominio. El nombre del concepto se compone, siguiendo los lineamientos de los tipos de datos presentados en Zapata (2012), como sigue: “NombreConcepto :NombreConceptoClase”.

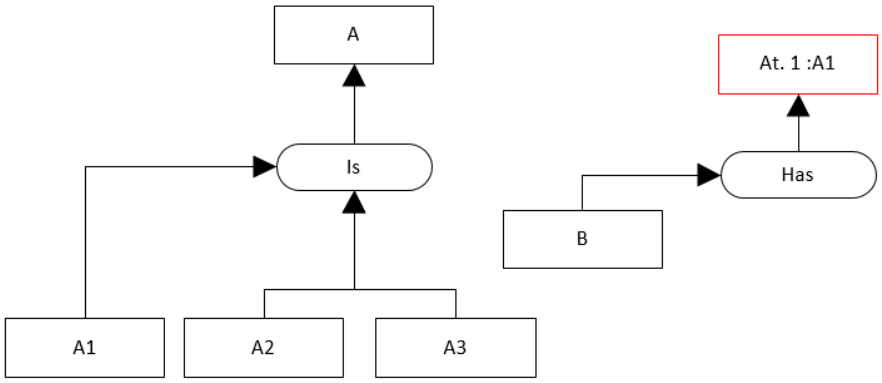
En la Figura 4-4 se muestra la representación del concepto tipo clase (“Nombre concepto”:”Nombre clase”) y en la Figura 4-5 se resalta el ejemplo de su uso.

Figura 4-4: Representación gráfica de un concepto tipo clase en EP.



Fuente: Elaboración propia.

Figura 4-5: Ejemplo de concepto tipo clase en EP.



Fuente: Elaboración propia.

En la Tabla 4-2 se presenta la regla para obtener el código en Java a partir de un concepto tipo clase.

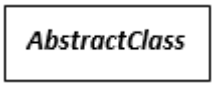
Tabla 4-2: Regla para la obtención de código a partir de un concepto tipo clase.

Concepto tipo clase	Código en Java
X :ClassA	ClassA X = new ClassA();

4.1.3 Concepto abstracto

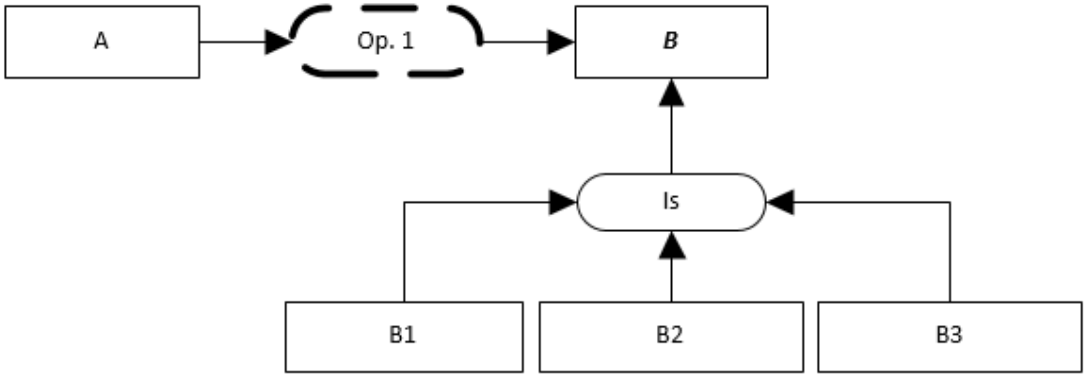
Este concepto se introduce con el fin de recrear las clases abstractas de POO, en las que se define una interacción o relación genérica a una clase y su implementación se define en las clases hijas (Shalloway y Trott, 2004). Este tipo de relaciones están presentes en gran parte de los patrones de diseño. Para la representación de este tipo de conceptos se usa la representación convencional con el nombre del concepto en negrita y formato cursiva. En la **Figura 4-6** se propone la representación del concepto abstracto en EP y en la **Figura 4-7** se expresa el ejemplo de un concepto abstracto.

Figura 4-6: Representación gráfica de un concepto abstracto en EP.



Fuente: Elaboración propia.

Figura 4-7: Ejemplo del uso de un concepto abstracto en EP.



Fuente: Elaboración propia.

Los conceptos abstractos tienen restricciones de uso: debe ser una clase padre y, además, en la representación del dominio sólo se pueden especificar sus relaciones dinámicas que no sean abstractas. En la Tabla 4-3 se presenta el código en Java equivalente al concepto abstracto.

Tabla 4-3: Regla para la obtención de código a partir de un concepto abstracto.

Concepto abstracto	Código en Java

A	public abstract class A {
	
	}

4.1.4 Relación dinámica abstracta

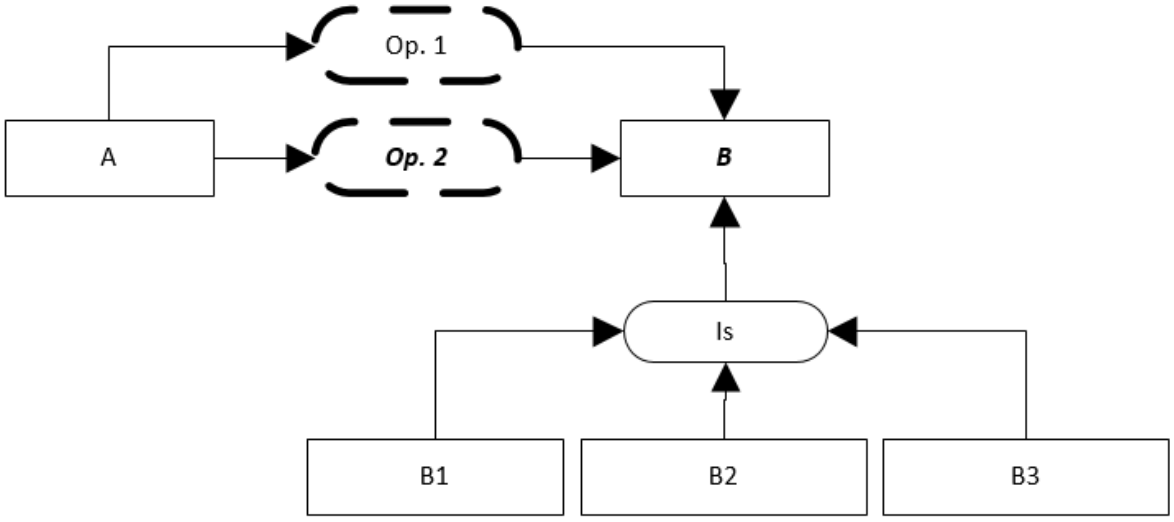
Cuando aparecen conceptos abstractos en el dominio, existe la posibilidad de que se conecte a relaciones dinámicas abstractas y no abstractas. Las relaciones dinámicas abstractas permiten que las subclases de un concepto abstracto especifiquen el comportamiento de la misma relación dinámica de manera independiente. Se propone una representación bajo el mismo símbolo gráfico presentado en Zapata *et al.* (2007) para las relaciones dinámicas, pero con el nombre en negrita y formato cursiva. En la **Figura 4-8:** Representación de la relación dinámica abstracta en EP.**Figura 4-8** se propone la

representación de la relación dinámica abstracta. En la **Figura 4-9** se propone un ejemplo donde se representan dos tríadas dinámicas, una abstracta y la otra convencional.

Figura 4-8: Representación de la relación dinámica abstracta en EP.

Fuente: Elaboración propia.

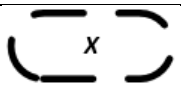
Figura 4-9: Ejemplo de una relación dinámica abstracta en EP.

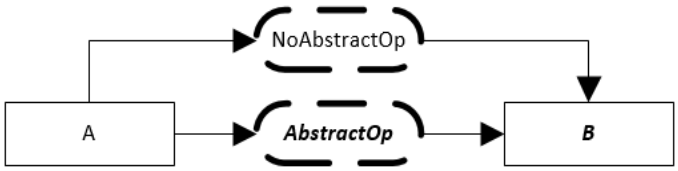


Fuente: Elaboración propia.

En la representación del dominio deberá existir sólo una especificación para la relación dinámica "*Op. 1*" y especificaciones para la relación dinámica "*Op. 2*" para cada uno de los conceptos hijos del concepto abstracto "B". Las tríadas dinámicas que contengan relaciones dinámicas abstractas siempre deben finalizar en un concepto abstracto. En la Tabla 4-4 se presentan las reglas para la obtención de código a partir de este tipo de relaciones dinámicas.

Tabla 4-4: Reglas para la obtención de código a partir de relaciones dinámicas abstractas.

Relación dinámica abstracta	Código en Java
	abstract void X();

	public abstract class B {
	public void NoAbstractOp()
	{
	//Code...
	}
	abstract void AbstractOp();
	}

4.1.5 Parámetro

Los EP carecen de un elemento que parametrice las relaciones dinámicas o funciones del sistema. Zapata (2007) propone la parametrización de las relaciones dinámicas sólo si éstas se conectan al atributo de un concepto. Esta aproximación no permite relaciones dinámicas más complejas como las presentes en contextos científicos y, además, limita el número de parámetros a uno. Se propone la forma de hexágono para la representación de los parámetros en especificaciones de relaciones dinámicas atómicas, especificaciones de tipo marco y funciones en EP. En la Figura 4-10 se propone su representación.

Figura 4-10: Representación del elemento parámetro en EP.

Fuente: Elaboración propia.

Este elemento puede tener un valor genérico o se puede restringir a un tipo de clase específico usando la notación que se presenta en la Sección 4.1.2. La ejemplificación de este elemento se presenta en las Secciones 4.1.6 y 4.1.7.

4.1.6 Operadores predefinidos

Zapata (2012) incluye el elemento operador como comparador de conceptos mediante signos matemáticos básicos. En esta Sección se proponen varios operadores adicionales que representan funciones matemáticas y trigonométricas. Siguiendo los lineamientos de este elemento, todos los llamados a los operadores predefinidos se deben realizar con la notación tipo árbol. Por otro lado, estos operadores sólo se pueden llamar dentro de un elemento especificación para caracterizar una relación dinámica o un concepto (Zapata, 2012). A estos operadores se conectan conceptos, valores o parámetros.

- **Operadores matemáticos**

  En la Tabla 4-5 se resumen los Operadores matemáticos predefinidos. Además, en la Tabla 4-6 se presentan las reglas para la obtención del código en Java a partir de los operadores matemáticos.

Tabla 4-5: Operadores matemáticos predefinidos en EP.

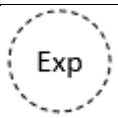
Operador	Significado matemático	Operador	Significado matemático
	Logaritmo natural		Módulo
	Raíz cuadrada		Euler
	Potencia		Valor absoluto

Tabla 4-6: Reglas para la obtención de código de los operadores matemáticos.

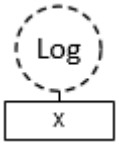
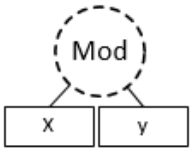
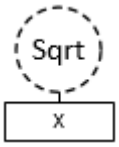
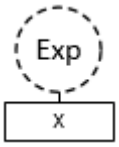
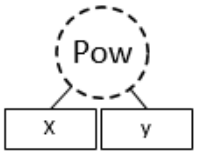
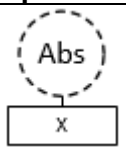
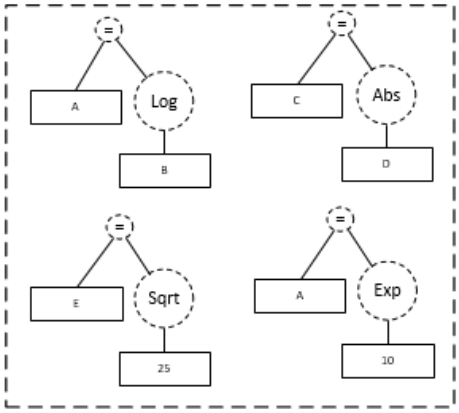
Operador	Código en Java	Operador	Código en Java
	Math.log(x);		x%y;
	Math.sqrt(x);		Math.exp(x);

Tabla 4-6: (Continuación) Reglas para la obtención de código de los operadores matemáticos.

Operador	Código en Java	Operador	Código en Java
	Math.pow(x,y);		Math.abs(x);

En la Figura 4-11 se muestran los ejemplos de los operadores que se conectan a sólo un elemento: *Log*, *Abs*, *Sqrt* y *Exp*. Por otro lado, en la Figura 4-12 se muestran los ejemplos de los operadores que se conectan a dos elementos: *Mod* y *Pow*.

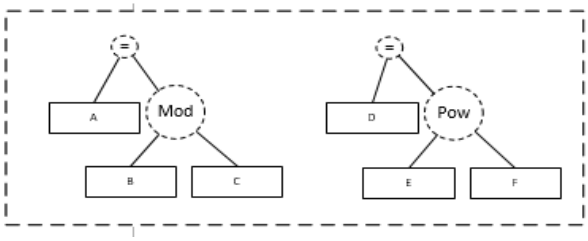
Figura 4-11: Ejemplo de los operadores de matemáticos *Log*, *Abs*, *Sqrt* y *Exp* en EP.



Fuente: Elaboración propia.

Se debe tener en cuenta el orden de los elementos que se conectan a los operadores. Los ejemplos de la Figura 4-12 representan “B Módulo C” y “E elevado a la F”.

Figura 4-12: Ejemplo de los operadores matemáticos *Mod* y *Pow* en EP.



Fuente: Elaboración propia.

- **Operadores trigonométricos**

  En la Tabla 4-7 se presenta un resumen de los operadores trigonométricos y en la Tabla 4-8 se presentan las reglas para la obtención de código a partir de estos operadores.

Tabla 4-7: Operadores trigonométricos predefinidos en EP.

Operador	Significado matemático	Operador	Significado matemático
Sin	Seno	Csc	Cosecante
Cos	Coseno	Ctg	Cotangente
Tan	Tangente	Sec	Secante

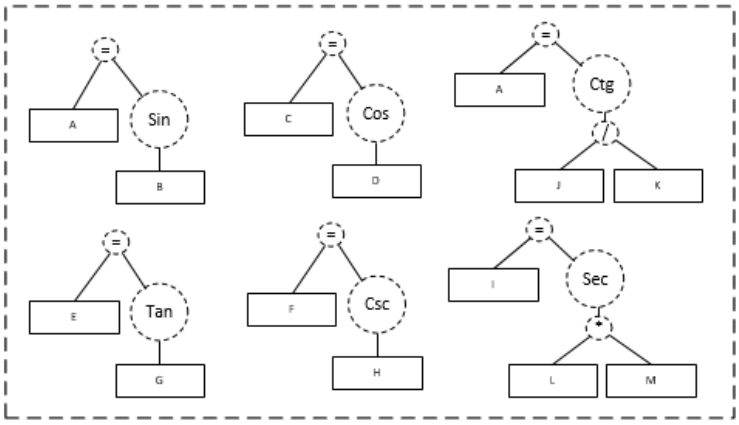
Tabla 4-8: Reglas para la obtención de código de operadores trigonométricos.

Operador	Código en Java	Operador	Código en Java
Sin	Math.sin(x);	Csc	1/Math.sin(x);
Cos	Math.cos(x);	Ctg	1/Math.tan(x);
Tan	Math.tan(x);	Sec	1/Math.cos(x);

En la Figura 4-13 se ejemplifican los operadores trigonométricos en EP.

Estos operadores matemáticos y trigonométricos permiten tener un acercamiento más preciso a los modelos y métodos matemáticos que están presentes en los contextos científicos para su comunicación y validación.

Figura 4-13: Ejemplo de los operadores trigonométricos en EP.



Fuente: Elaboración propia.

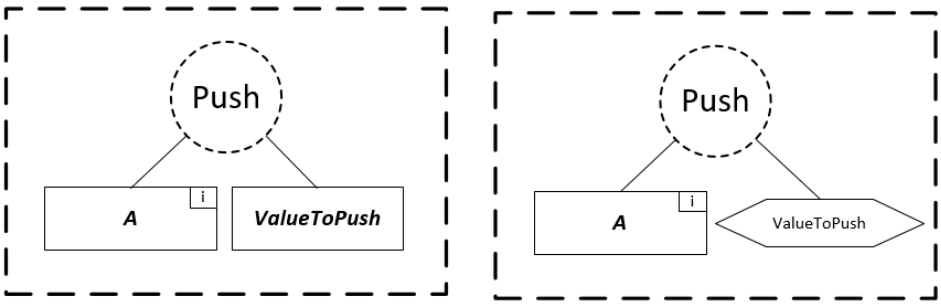
- **Operador *Push***

El Operador *Push* inserta un concepto, valor o parámetro dentro de un concepto tipo arreglo. El elemento insertado, siempre se almacena en la última posición del Concepto tipo arreglo. En la Figura 4-14 se propone la representación en EP del Operador *Push* y en la Figura 4-15 se muestra un ejemplo de este operador usando como elemento insertado un concepto y un parámetro.

Figura 4-14: Representación gráfica del Operador *Push* en EP.

Fuente: Elaboración propia.

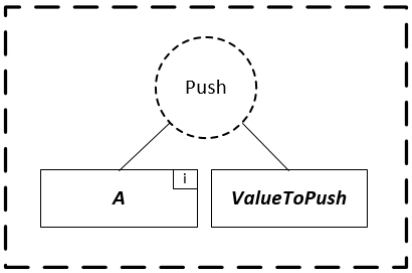
Figura 4-15: Ejemplo del Operador *Push*.



Fuente: Elaboración propia.

En la Tabla 4-9 se expresa la regla para obtener el código Java desde el operador *Push*.

Tabla 4-9: Regla para la obtención de código a partir del operador Push.

Operador *Push*	Código en Java
	A.add(ValueToPush);

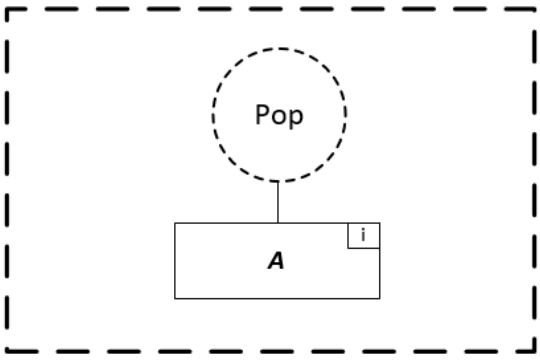
- **Operador Pop**

Este operador elimina el elemento almacenado en la última posición de un concepto tipo arreglo. En la Figura 4-16 se presenta el símbolo de este operador. Por otro lado, en la **Figura 4-17** se ejemplifica su uso en EP.

Figura 4-16: Representación gráfica del operador pop en EP.

Fuente: Elaboración propia.

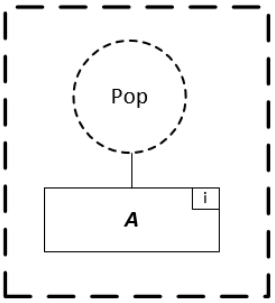
Figura 4-17: Ejemplo del operador pop.



Fuente: Elaboración propia.

En la Tabla 4-10 se muestra la regla para obtener código a partir del operador pop en EP.

Tabla 4-10: Regla para la obtención de código a partir del operador pop.

Operador pop	Código en Java
	A.remove(A.size()-1);

- **Operador *Contains***

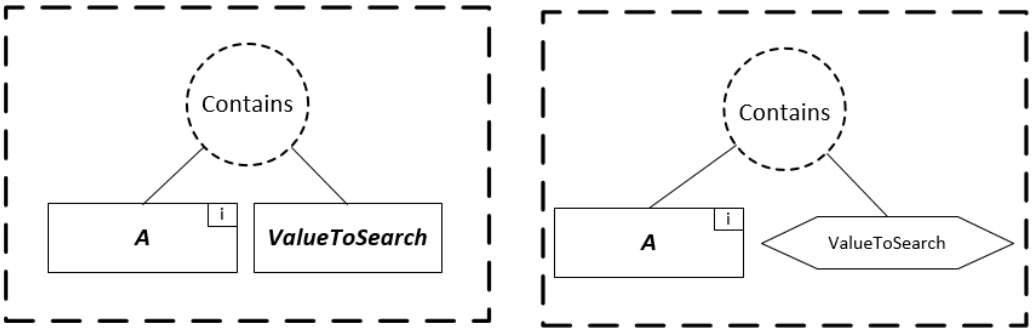
El operador *contains* verifica si un elemento (concepto, parámetro o valor) se encuentra almacenado en un concepto tipo arreglo.

En la Figura 4-18 se propone la representación del operador *contains*. Asimismo, en la Figura 4-19 se muestra un ejemplo de su uso.

Figura 4-18: Representación gráfica del operador *contains* en EP.

Fuente: Elaboración propia.

Figura 4-19: Ejemplo del operador *contains*.

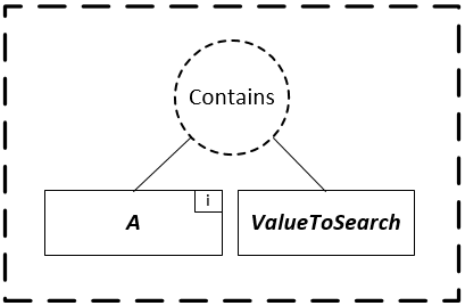


Fuente: Elaboración propia.

El ejemplo de la Figura 4-19 se lee así: "¿El concepto tipo arreglo A contiene a *ValueToSearch*?".

En la Tabla 4-11 se presenta la regla para obtener el código en Java a partir del operador *contains*.

Tabla 4-11: Regla para la obtención de código a partir del operador *contains*.

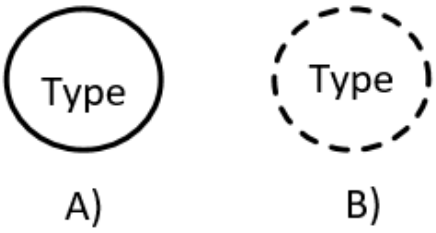
Operador *contains*	**Código en Java**
	A.contains(ValueToSearch);

- **Operador *Type***

Este operador se propone con dos variaciones: asignación e información. El operador *Type* de asignación, le define un tipo de dato a un concepto. Por otro lado, el operador *Type* de información, describe el tipo de dato de un concepto. Los tipos de datos pueden ser: entero, booleano, *string*, doble o de tipo clase.

En la Figura 4-20 se representa el operador *type* de asignación (A) y de información (B). Este operador sigue la misma sintaxis tipo árbol que se presenta en Zapata (2012).

Figura 4-20: Representación gráfica del operador *type* en EP.

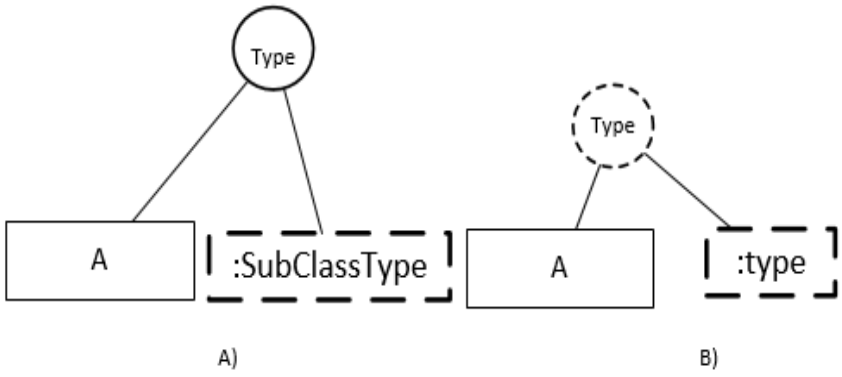


Fuente: Elaboración propia.

Este operador tiene algunas restricciones de uso: el operador *type* de asignación sólo le puede asignar valores tipo clase (":*className*") a conceptos clase del dominio; estos

conceptos deben tener subclases y los valores deben representar sólo estas subclases. Por otro lado, el operador *type* de información consulta si el tipo de un concepto es el del valor con el que se compara. En la Figura 4-21 se ejemplifica su uso.

Figura 4-21: Ejemplo del operador *type*.



Fuente: Elaboración propia.

En la Tabla 4-12 se expresa la regla para obtener el código en Java de la implementación del operador *type*.

Los operadores presentados en esta Sección facilitan la asignación de tipos de clases a conceptos clase del dominio, lo cual aporta una mayor flexibilidad al modelo.

Tabla 4-12: Regla para la obtención de código a partir del operador *type*.

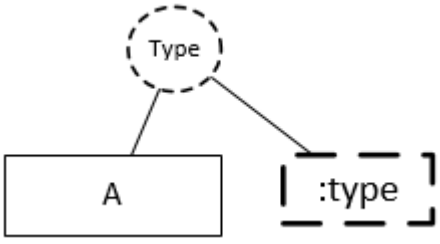
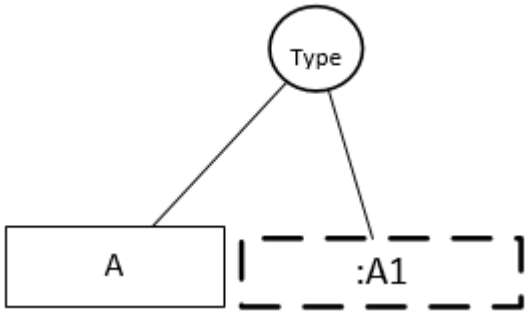
Operador *type*	**Código en Java**
	**if(A.getClass().getName().equals("type"))** **{** **return true;** **}** **else** **{** **return false;** **}**

Tabla 4-12: (Continuación) Regla para la obtención de código a partir del operador *type*.

Operador *type*	Código en Java

	**Precondición:**
	public class A
	{
	
	}
	public class A1 extends A
	{
	
	}
	Código del operador:
	A A = new A1();

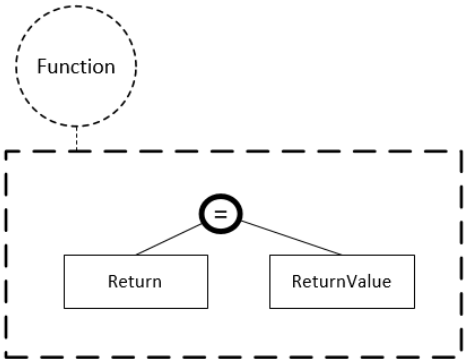
4.1.7 Funciones que define el analista

En contextos científicos se crean usualmente funciones específicas cuya reutilización es deseable debido a su complejidad (Rasmussen *et al.*, 2014). Los EP no brindan la posibilidad de reutilizar el comportamiento de una relación dinámica. Se propone ampliar el alcance del elemento operador una vez más para permitir al analista definir un operador personalizado cuya especificación defina el comportamiento de una función que se pueda reutilizar varias veces en todo el EP. El símbolo gráfico para las funciones definidas por el analista es el mismo del operador que presentan Zapata *et al.* (2012) y, además, tiene un nombre personalizado que representará el nombre de la función. Este nombre de función no podrá ser ninguno de los nombres de las funciones matemáticas y trigonométricas presentadas en la Sección 4.1.6.

El símbolo de operador se acompaña con una especificación que caracteriza el comportamiento de la función que se quiere reutilizar. Por otro lado, en su especificación siempre tendrá un concepto llamado retorno al cual se asigna el valor que devuelve la función con base en su comportamiento interno. Finalmente, en esta especificación

pueden existir parámetros que ayuden a definir la estructura de la función. En la Figura 4-22 se representa gráficamente una función que define el analista.

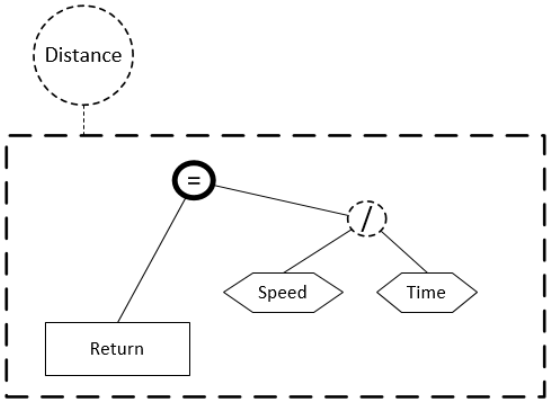
Figura 4-22: Representación gráfica de una función que define el analista en EP.



Fuente: Elaboración propia.

El valor del concepto "*Return*" se puede definir a partir de un concepto, un valor, un parámetro o una operación matemática. Este valor puede ser de varios tipos de datos: *string*, entero, doble, booleano, etc. En la Figura 4-23 se muestra el ejemplo de una función que define el analista que calcula la distancia con base en dos parámetros.

Figura 4-23: Ejemplo de una función que define el analista en EP.

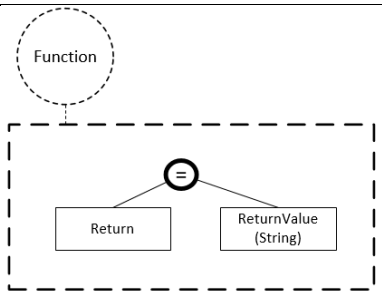


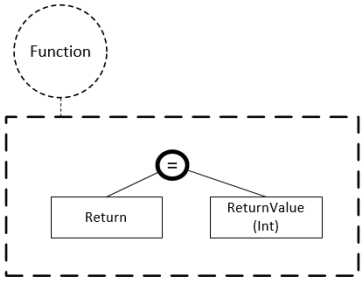
Fuente: Elaboración propia.

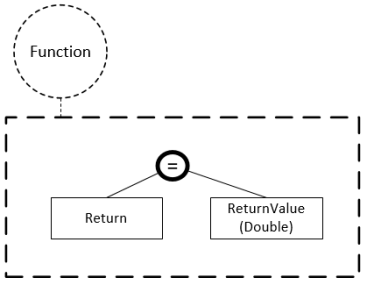
En la Tabla 4-13 se presentan las reglas para obtener el código en Java a partir de funciones que crea el usuario de tipos de retorno *string*, entero, doble, booleano y el caso de una función con parámetros.

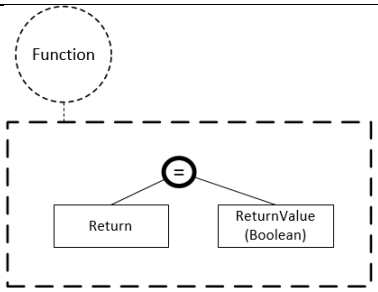
Tabla 4-13: Reglas para la obtención de código a partir de funciones que crea el analista en EP.

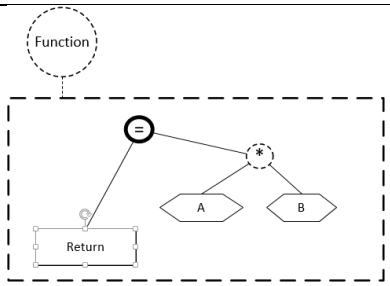
Función definida por el analista	Código en Java

	public String Function()
	{
	String returnValue = /*Code...*/;
	return returnValue;
	}

	public int Function()
	{
	int returnValue = /*Code...*/;
	return returnValue;
	}

	public Double Function()
	{
	Double returnValue = /*Code...*/;
	return returnValue;
	}

	public Boolean Function()
	{
	Boolean returnValue = /*Code...*/;
	return returnValue;
	}

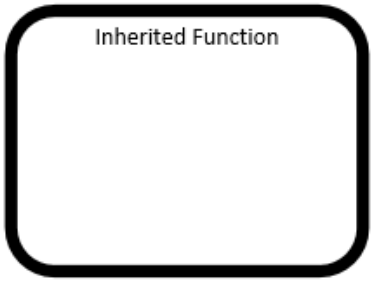
	public int Function(A,B)
	{
	int returnValue = A*B;
	return returnValue;
	}

Estos nuevos elementos le darán al sistema mayor flexibilidad y facilidad de mantenimiento.

4.1.8 Especificación tipo Marco

En esta Sección se propone el marco que presentan Zapata *et al.* (2012) para caracterizar relaciones dinámicas abstractas o relaciones dinámicas complejas cuya especificación contenga el llamado a otras relaciones dinámicas. Adicionalmente, se define una restricción para el nombre del marco: este nombre deberá ser el mismo de la relación dinámica (abstracta o no) que caracteriza. En la Figura 4-24 se propone la especificación tipo marco.

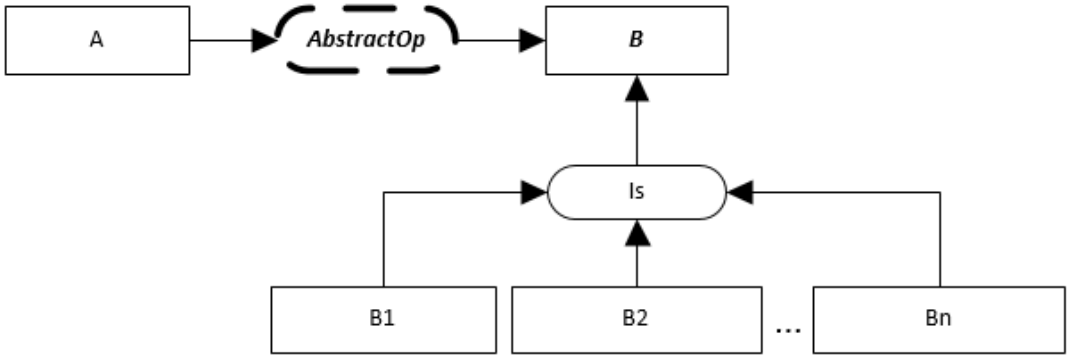
Figura 4-24: Representación gráfica de la especificación tipo marco en EP.



Fuente: Adaptado de Zapata (2012).

Por cada relación dinámica abstracta que se conecte a un concepto abstracto, las clases hijas de este concepto deberán tener una especificación tipo marco para definir la implementación individual de dicha relación. La implementación de cada relación dinámica abstracta puede incluir relaciones dinámicas que se conectan a otros conceptos del dominio e, incluso, tener especificaciones combinadas con otras relaciones dinámicas. En la Figura 4-25 se muestra la primera parte del ejemplo de la especificación tipo marco, donde se observan dos tríadas dinámicas, una abstracta y otra no abstracta. Se representa, además, un concepto abstracto “*B*” y los conceptos “B1”, “B2”…”Bn” como sus conceptos hijos.

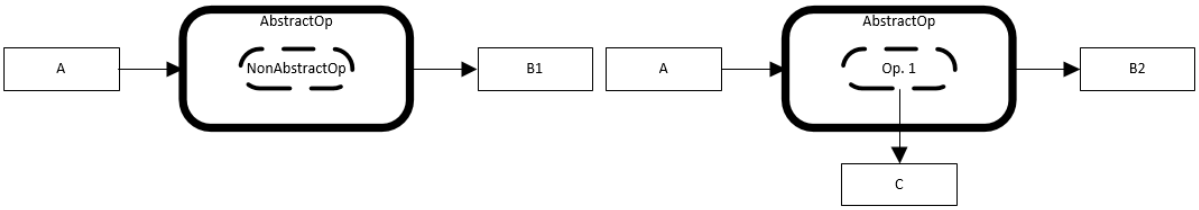
Figura 4-25: Ejemplo de la implementación de Especificación tipo Marco (1/3)



Fuente: Elaboración propia

En la Figura 4-26 se presenta la especificación tipo marco para la relación dinámica abstracta "*AbstractOp*" de los conceptos hijos "B1" y "B2". Estas especificaciones pueden variar tanto como sea necesario entre cada uno de los conceptos hijos.

Figura 4-26: Ejemplo de la implementación de Especificación tipo Marco (2/3).

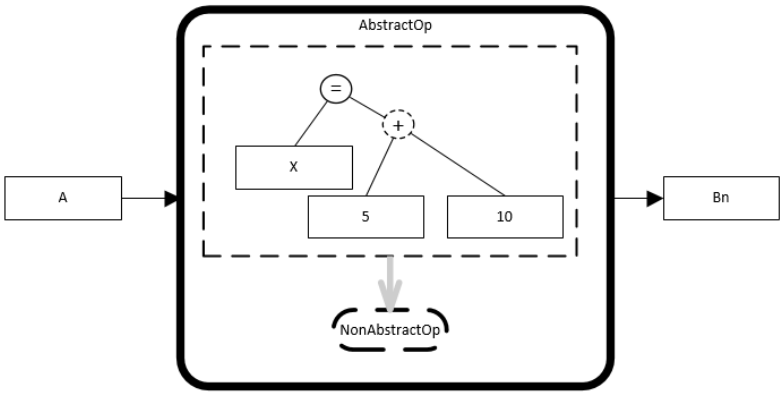


Fuente: Elaboración propia.

Por otra parte, en la Figura 4-27 se presenta la especificación tipo marco de la relación abstracta "*AbstractOp*" del Concepto hijo "Bn" donde se puede observar la combinación de una especificación y una relación dinámica.

Adicionalmente, en la Tabla 4-14 se presentan las reglas para la obtención del código a partir de la especificación tipo marco en EP.

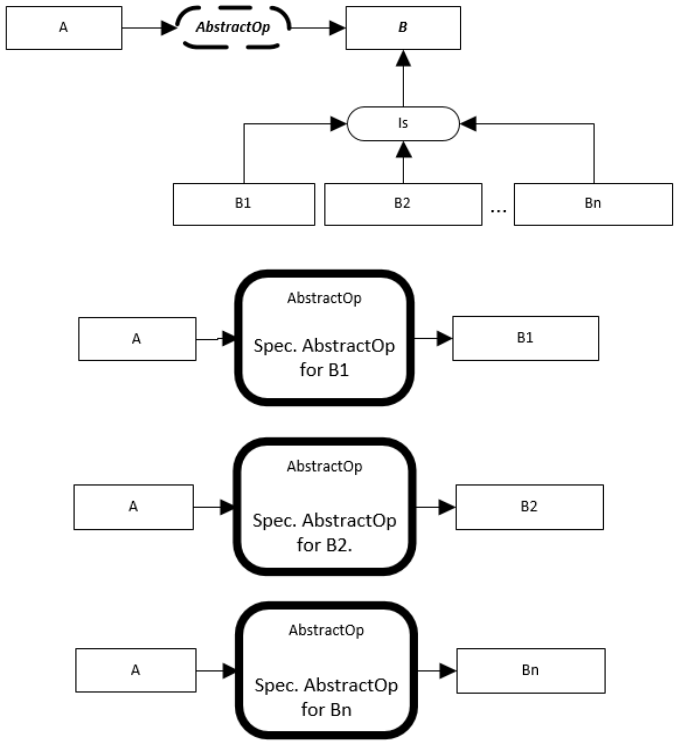
Figura 4-27: Ejemplo de la implementación de especificación tipo marco (3/3).



Fuente: Elaboración propia.

Tabla 4-14: Reglas para la obtención de código a partir de la especificación tipo marco

Especificación tipo marco	**Código Java**

	public abstract class B {
	  public abstract void AbstractOp();
	}
	public class B1 extends B{
	  @Override
	  public void AbstractOp() {
	  //Spec. AbstractOp for B1
	  }
	}
	public class B2 extends B{
	  @Override
	  public void AbstractOp() {
	  //Spec. AbstractOp for B2
	  }
	}
	public class Bn extends B{
	  @Override
	  public void AbstractOp() {
	  //Spec. AbstractOp for Bn
	  }}

4.1.9 Conexión de responsabilidad

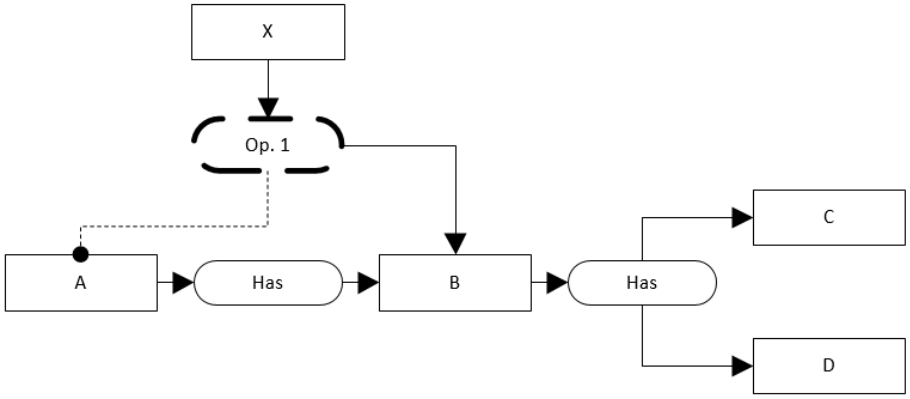
Uno de los principales objetivos de los PD es delegar la responsabilidad de ejecutar y gestionar las funciones a objetos específicos que, en ocasiones, no son los actores directos de dichas relaciones, sino intermediarios, con el fin de promover el desacople en el sistema (Shalloway y Trott, 2004). En los EP, las tríadas dinámicas se componen de un concepto, una relación dinámica y otro concepto. Estas tríadas, según las reglas de conversión a diagrama de clases, incluyen la relación dinámica como operación del concepto que finaliza la tríada (Zapata y Arango, 2007). Esta aproximación obliga a que el concepto que finaliza la tríada sea un atributo para, así, asignar la función al concepto que lo posee y mantener consistentes las reglas de conversión. En algunos casos, es necesario componer tríadas dinámicas con conceptos clase como atributos de otro concepto y, además, asignarle la responsabilidad al concepto que lo posee. Por ello, se propone la conexión de responsabilidad como una asociación al concepto que tiene la responsabilidad de ejecutar la relación dinámica. Esta conexión sólo es válida en el caso de una tríada dinámica que finalice con atributo que sea concepto clase en el dominio. Su representación es una línea punteada con un conector circular y relleno, el cual se conectará en el concepto que posee la responsabilidad. En la Figura 4-28 se propone su representación.

Figura 4-28: Representación gráfica de la conexión de responsabilidad.

Fuente: Elaboración propia.

En la Figura 4-29 se presenta un ejemplo de la conexión de responsabilidad, donde se muestra un concepto “A” que tiene como atributo a un concepto “B” y, además, este concepto es un concepto clase. Aunque la relación dinámica “*Op. 1*” recae sobre el concepto “B”, la responsabilidad se asociará con el concepto “A”.

Figura 4-29: Ejemplo de una conexión de responsabilidad en EP.

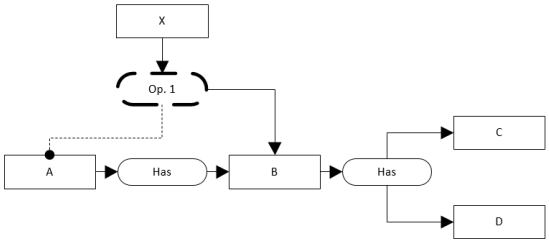


Fuente: Elaboración propia.

En la Tabla 4-15 se muestra una regla para la obtención de código en Java de la conexión de responsabilidad.

Tabla 4-15: Reglas para la obtención de código de la conexión de responsabilidad.

Conexión de responsabilidad	Código Java

	public class A {
	B B = new B();
	public void Op1()
	{
	//Code…
	}
	}
	public class B {
	String C;
	String D;
	}

4.2 Representación de los Patrones de Diseño de Creación para Software Científico en Esquemas Preconceptuales

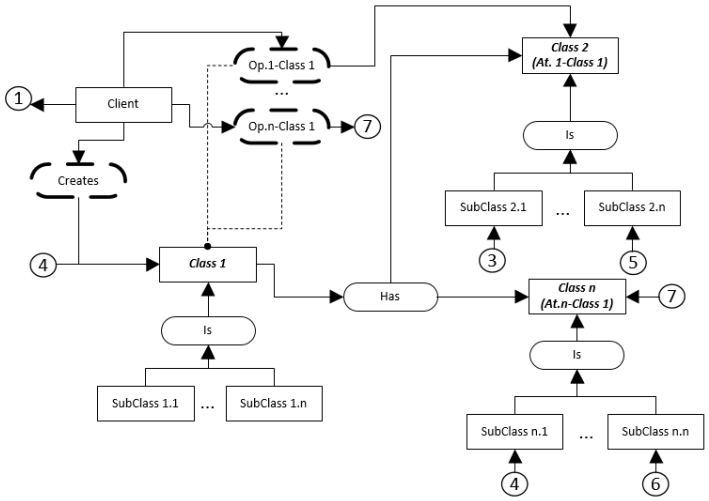
Con base en el resumen de la literatura presentado en la Sección 3.1, en esta Sección se presentan los patrones de diseño de creación que más se usan en software científico: *Abstract Factory y Builder.* Se omite el PD Factory *Method* dado que el PD *Abstract Factory* incluye todos sus elementos y funciona de la misma manera para un contexto más general (Shalloway y Trott, 2004). Estas representaciones y descripciones se crean a partir de las

especificaciones textuales de algunos problemas encontrados en la literatura (Gamma *et al*., 1994; Shalloway y Trott, 2004).

4.2.1 Representación del problema del PD *Abstract Factory* en EP.

En esta Sección se propone una representación genérica del problema que resuelve el PD *Abstract Factory* para ejemplificar la situación inicial que dispara su implementación. En la Figura 4-30 se presenta la primera parte de esta representación. Allí se describe, principalmente, la creación de un objeto complejo (concepto "*Class* 1") a partir de una familia de objetos ("*Class 2*" y "*Class n*").

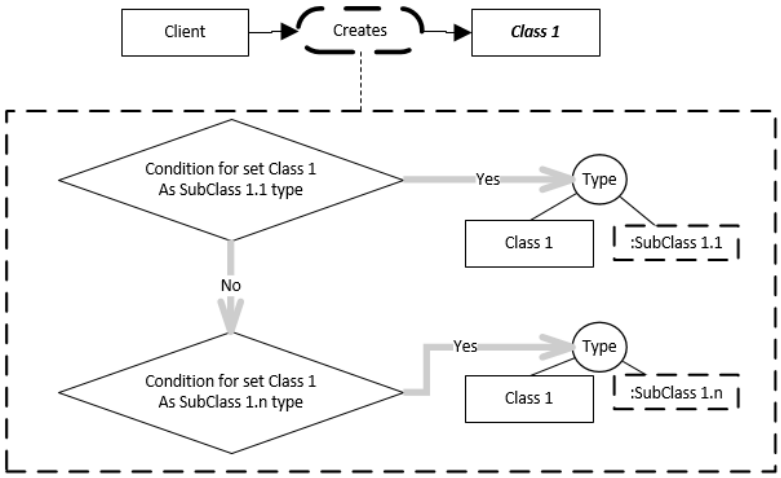
Figura 4-30: Representación en EP del problema genérico del PD *Abstract Factory* (1/3).



Fuente: Elaboración propia.

En la Figura 4-31 se presenta la segunda parte de esta representación, donde se especifica la relación dinámica "*creates*". En la definición del dominio debe existir una relación dinámica con estas características en su especificación, donde se describa la elección y asignación de un concepto hijo del concepto principal "*Class 1*" bajo cualquier criterio. Estos criterios de elección pueden variar. Una vez se ejecuta esta relación dinámica, el tipo de dato del concepto "*Class 1*" es *"SubClass 1.1"* o *"SubClass 1.n"*.

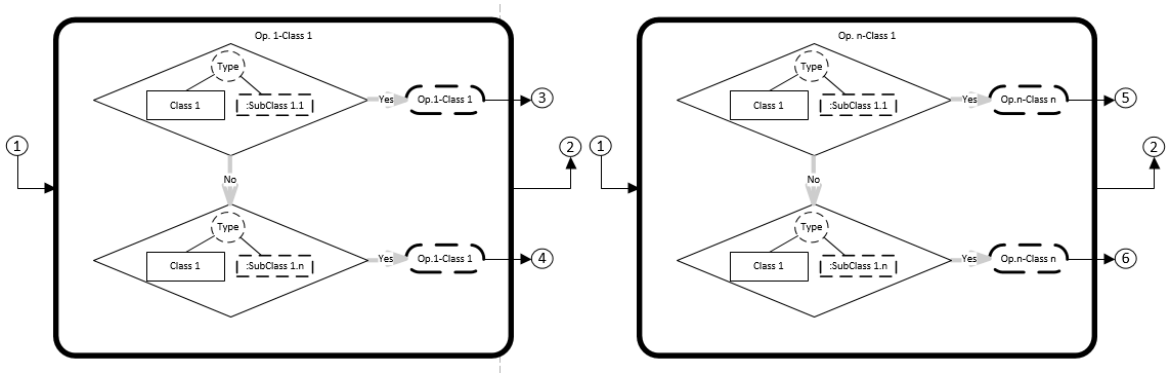
Figura 4-31: Representación en EP del problema genérico del PD *Abstract Factory* (2/3).



Fuente: Elaboración propia.

En la Figura 4-32 se muestran las especificaciones de las relaciones dinámicas "*Op. 1-Class 1*" y "*Op. n-Class 1*" del concepto "*Class* 1". Estas operaciones, al no ser abstractas, se heredan directamente a los conceptos hijo. En el dominio representado, la especificación de estas relaciones dinámicas deben incluir *switches* o condicionales que definan el comportamiento de dichas relaciones (un comportamiento diferente para cada subclase).

Figura 4-32: Representación en EP del problema genérico del PD *Abstract Factory* (3/3).

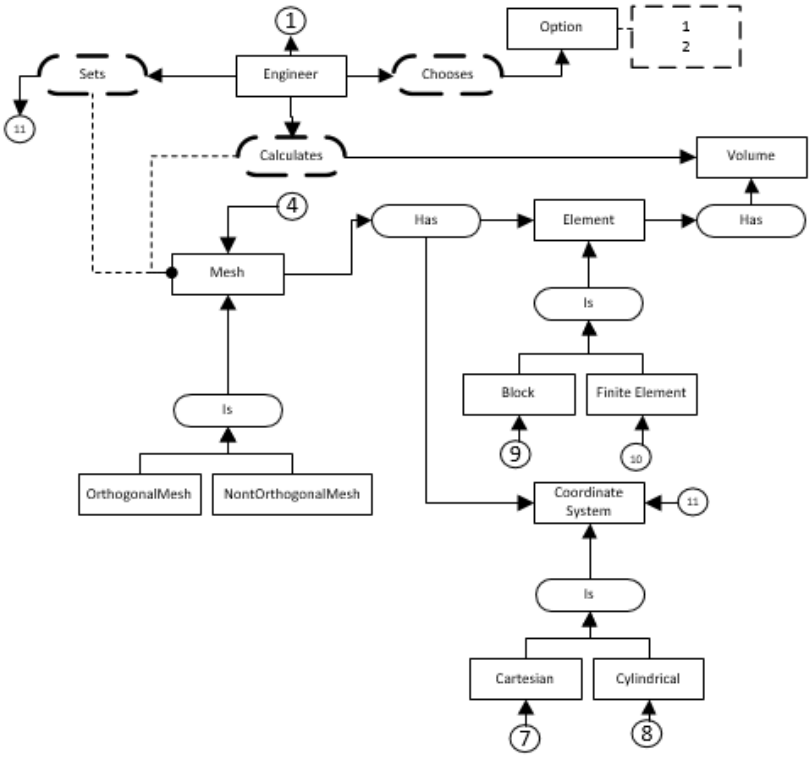


Fuente: Elaboración propia.

Este problema, en resumen, se presenta como la creación de objetos complejos y su asignación de responsabilidades a partir de *switches* y condicionales. En esta situación, por cada relación dinámica que se incluya en el dominio, deben existir estructuras de

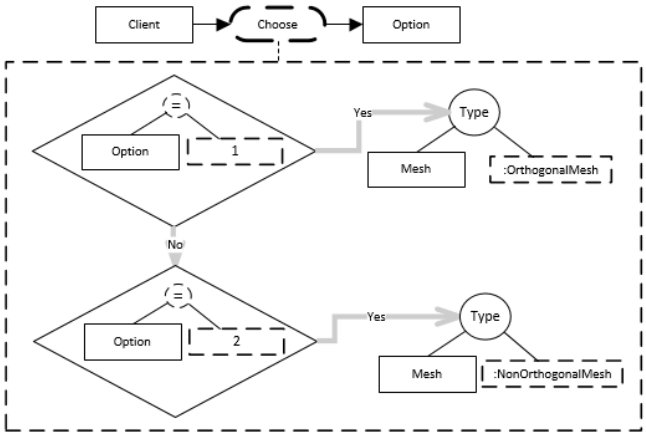
control que determinen el comportamiento en su especificación para cada uno de los conceptos hijos del concepto "*Class 1*". Por esta razón, nuevos conceptos hijos de este concepto, implican cambios en todo el dominio. En la Figura 4-33 se representa una fracción de dominio científico. En este caso, un ingeniero define el sistema coordenado de una malla y calcula el volumen de sus elementos. Existen dos tipos de malla: ortogonales y no ortogonales. Las mallas ortogonales tienen elementos de tipo bloque y sistema coordenado cartesiano. Por otro lado, las mallas no ortogonales tienen elementos de tipo elemento finito y sistema coordenado de tipo cilíndrico. En la Figura 4-34 se especifica la relación dinámica "*choose*", que controla la selección de un subtipo para el concepto clase Malla (*Mesh*). Finalmente, en la Figura 4-35 se presentan las especificaciones tipo marco de las relaciones dinámicas "*sets*" y "*calculates*", que sirven como intermediarias para disparar las relaciones dinámicas que afectan directamente al resto de conceptos del sistema.

Figura 4-33: Ejemplo de la representación del problema general del PD *Abstract Factory* en EP (1/3).



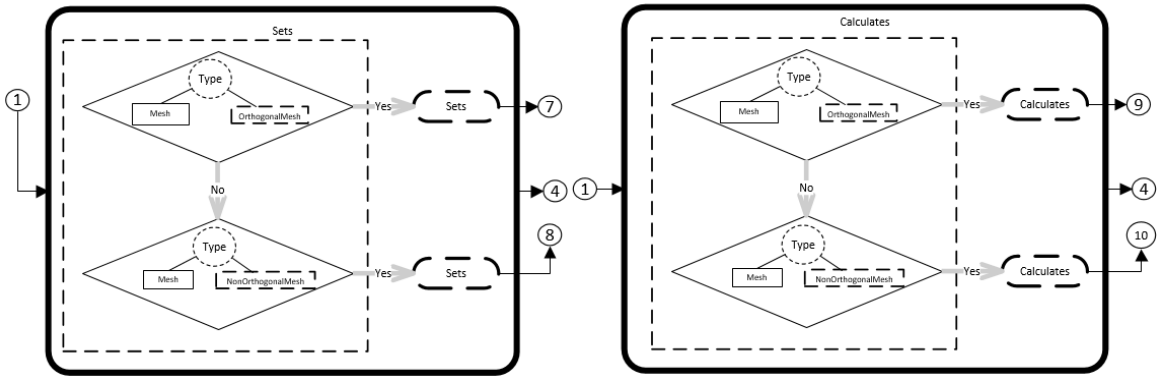
Fuente: Elaboración propia.

Figura 4-34: Ejemplo de la representación del problema general del PD *Abstract Factory* en EP (2/3).



Fuente: Elaboración propia.

Figura 4-35: Ejemplo de la representación del problema general del PD *Abstract Factory* en EP (3/3).



Fuente: Elaboración propia.

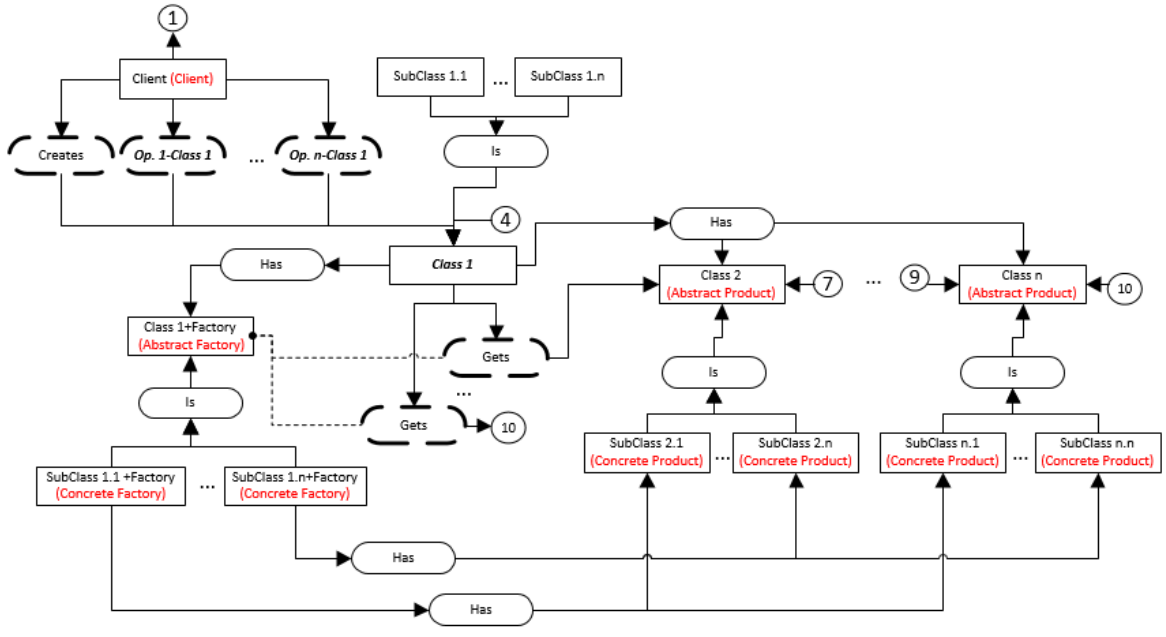
4.2.2 Representación del patrón de diseño *Abstract Factory* en Esquemas Preconceptuales.

En esta sección se propone una representación de los elementos e interacciones del PD *Abstract Factory.* Gamma *et al.* (1994) definen los elementos principales de este patrón de diseño:

- ***Client**:* este elemento se comunica con el resto del sistema por medio de los elementos abstractos "*Abstract Factory*" y "*Abstract Product*". Esta clase no conoce la implementación concreta de los elementos abstractos.
- ***Abstract Factory**:* la función de este elemento es definir una interfaz para crear los elementos de tipo *Concrete Product* que componen como familia, el objeto complejo que se crea.
- ***Concrete Factory**:* este elemento implementa las operaciones para crear una familia elementos tipo *Concrete Product* específica.
- ***Abstract Product**:* el objetivo de este elemento es definir una interfaz para la creación de elementos del tipo *Concrete Product.*
- ***Concrete Product**:* este elemento caracteriza el *Concrete Product* que se crea y, además, hereda características del elemento *Abstract Product.*

La arquitectura completa del PD *Abstract Factory* en EP se representa en la Figura 4-36. En esta representación se muestra el rol de los conceptos (de color rojo) en el contexto del PD y las relaciones que mantiene con los otros conceptos del sistema. Posteriormente, se muestra la implementación de sus relaciones.

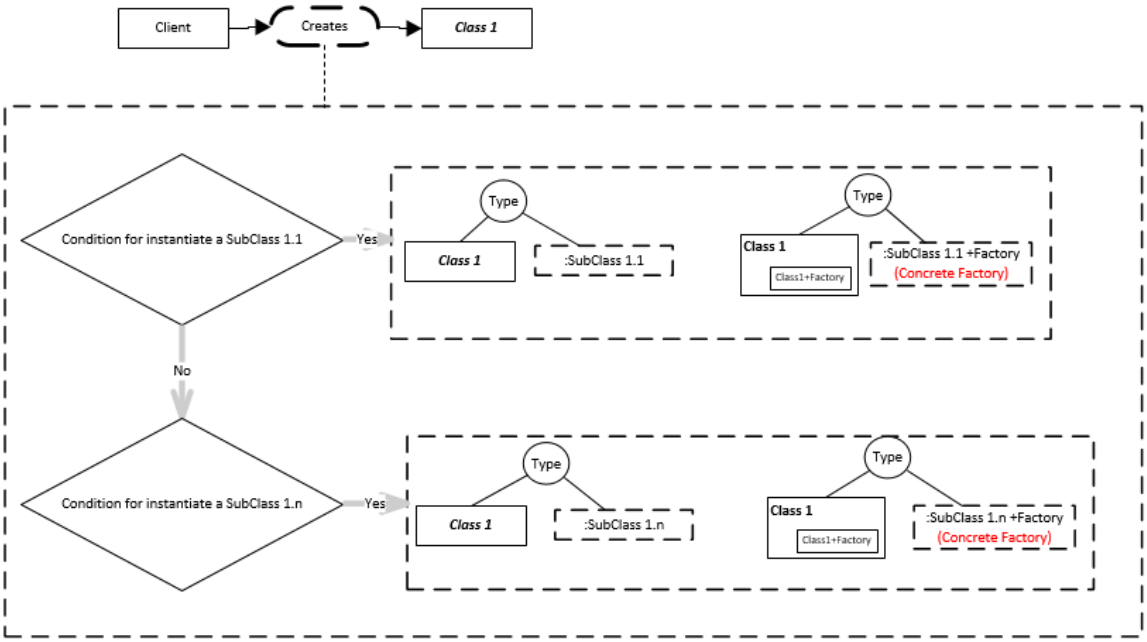
Figura 4-36: Representación del PD *Abstract Factory* en EP (1/4).



Fuente: Elaboración propia.

En la Figura 4-37 se muestra la especificación de la relación dinámica que define el tipo de *Concrete Factory* que se adopta en el sistema para la creación del resto de conceptos específicos que lo conforman (*Concrete Product*).

Figura 4-37: Representación del PD *Abstract Factory* en EP (2/4).



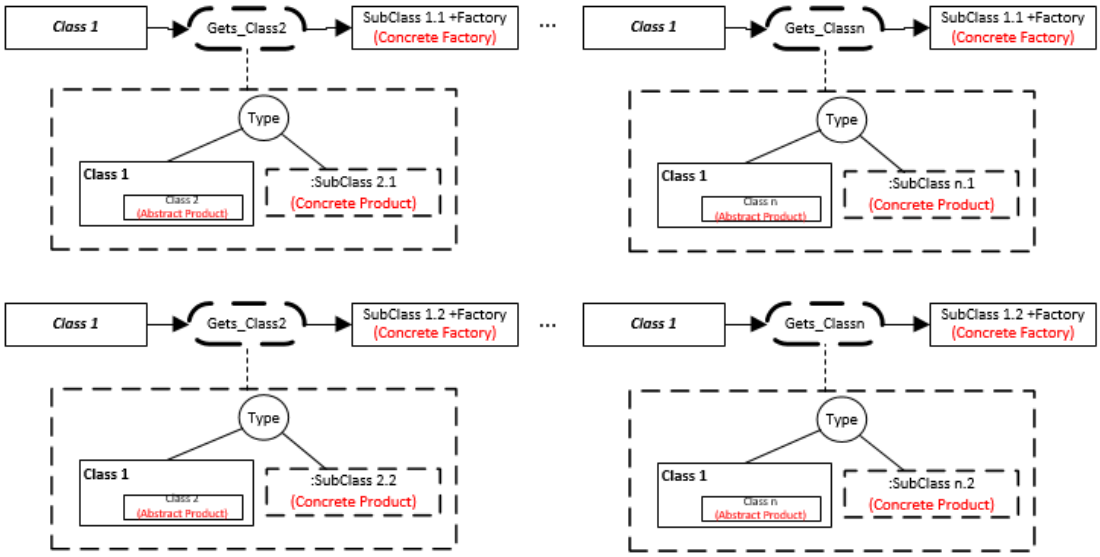
Fuente: Elaboración propia.

En la Figura 4-38 se propone la especificación de las dos relaciones dinámicas nuevas "*gets*" que aparecen una vez se aplica el PD. Estas relaciones, que conforman tríadas dinámicas con los conceptos "*Class* 2" y "*Class* n", se encargan de obtener los *Concrete Product* correctos con base en el tipo de *Concrete Factory* del sistema. Así, se logra que el objeto complejo que se crea tenga, exclusivamente, acceso a los demás objetos que lo componen.

Finalmente, en la Figura 4-39 se muestra la especificación de las relaciones dinámicas "*Op.1-Class1*" y "*Op.1-Classn*", que se conectan directamente al concepto principal "*Class* 1". Cabe resaltar que esta especificación simplifica la especificación original presentada en la Figura 4-35, dado que, al no tener la necesidad de conocer la implementación concreta

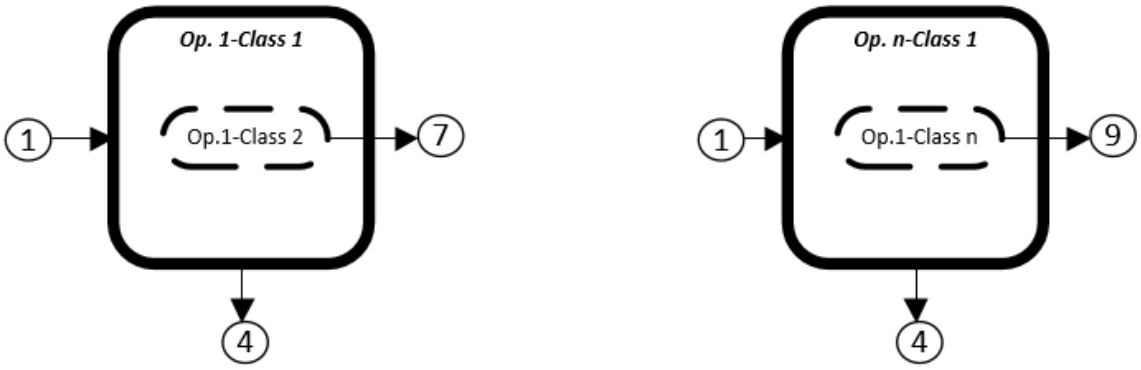
de los elementos que componen al concepto principal "*Class 1*", se pueden especificar de forma general y cada *Concrete Product* se encarga de implementarlo y ejecutarlo.

Figura 4-38: Representación del PD *Abstract Factory* en EP (3/4).



Fuente: Elaboración propia.

Figura 4-39: Representación del PD *Abstract Factory* en EP (4/4).

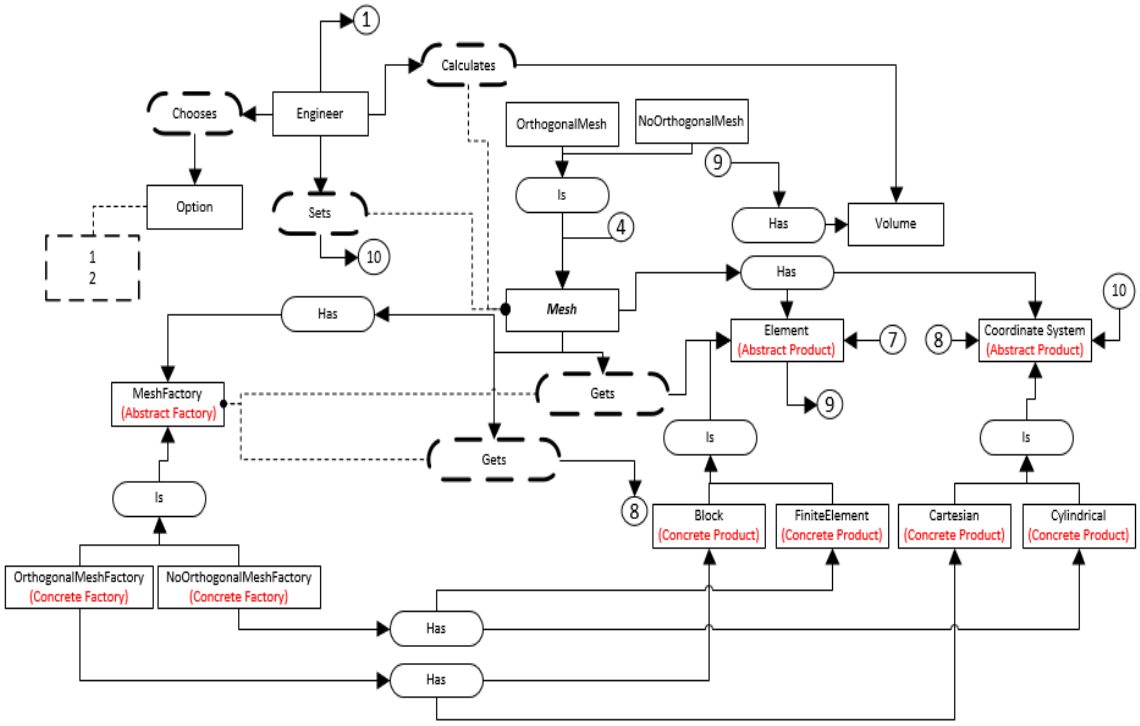


Fuente: Elaboración propia.

En la Figura 4-40 se ejemplifica la implementación del PD *Abstract Factory* sobre el ejemplo de la Figura 4-33. Esta representación muestra la misma fracción del dominio y

elimina los problemas de construcción de objetos complejos asignándole esa responsabilidad a los conceptos tipo *Concrete Factory.*

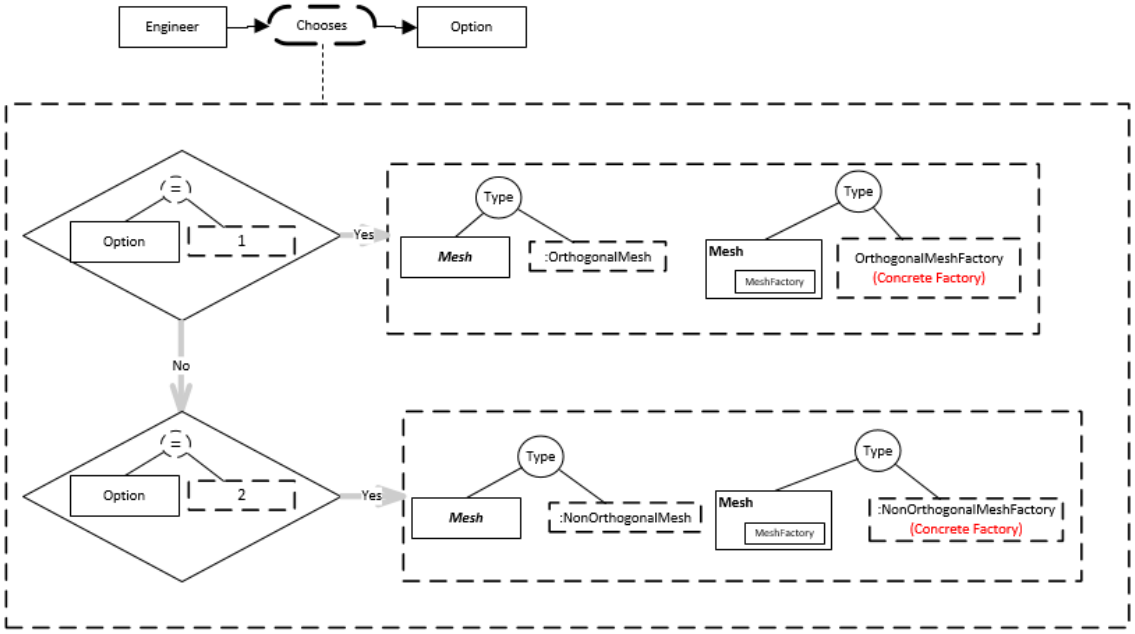
Figura 4-40: Ejemplo del PD *Abstract Factory* implementado en EP (1/4).



Fuente: Elaboración propia.

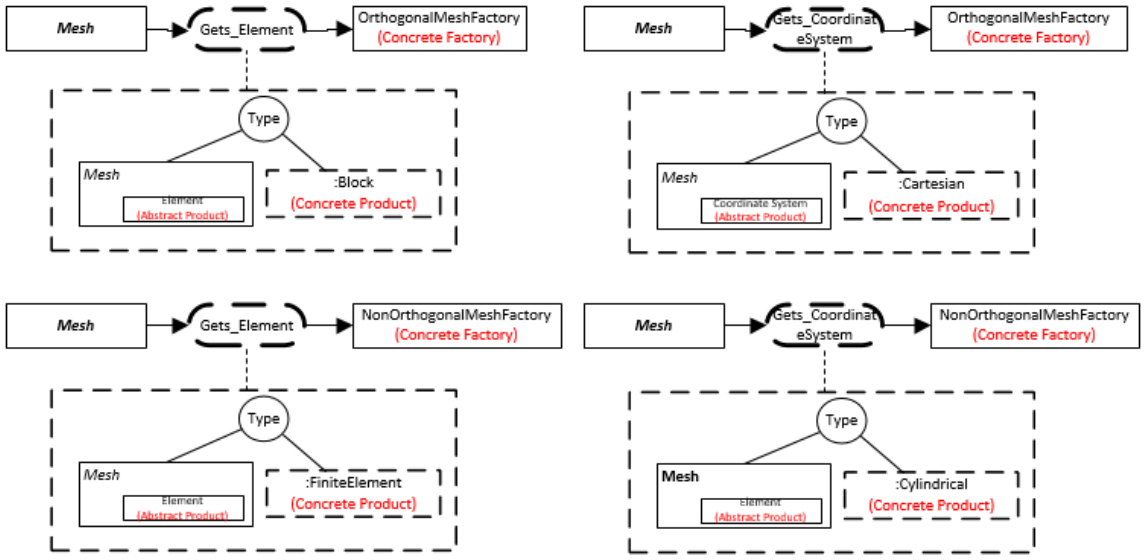
En la especificación representada en la Figura 4-41se define el tipo de Malla que se quiere construir (*Concrete Factory*). Este concepto ejecuta las operaciones necesarias para obtener sólo los elementos necesarios para componer al objeto más general (*Mesh*). Por otro lado, en la Figura 4-42 y la Figura 4-43 se especifican las otras relaciones dinámicas del sistema según el tipo de concepto *Concrete Factory* con los lineamientos presentados en esta Sección.

Figura 4-41: Ejemplo del PD *Abstract Factory* implementado en EP (2/4).



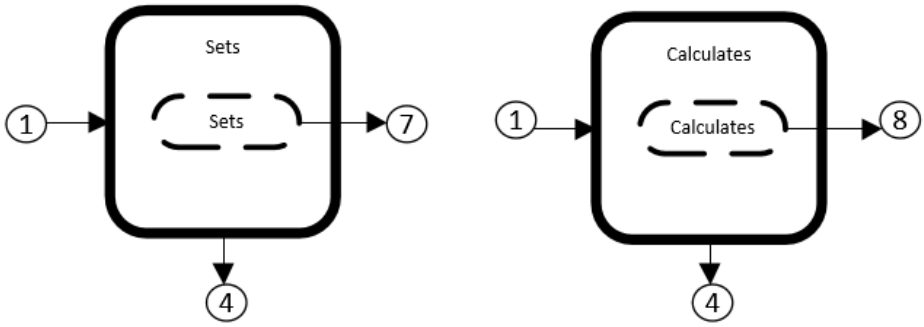
Fuente: Elaboración propia.

Figura 4-42: Ejemplo del PD *Abstract Factory* implementado en EP (3/4).



Fuente: Elaboración propia.

Figura 4-43: Ejemplo del PD *Abstract Factory* implementado en EP (4/4).

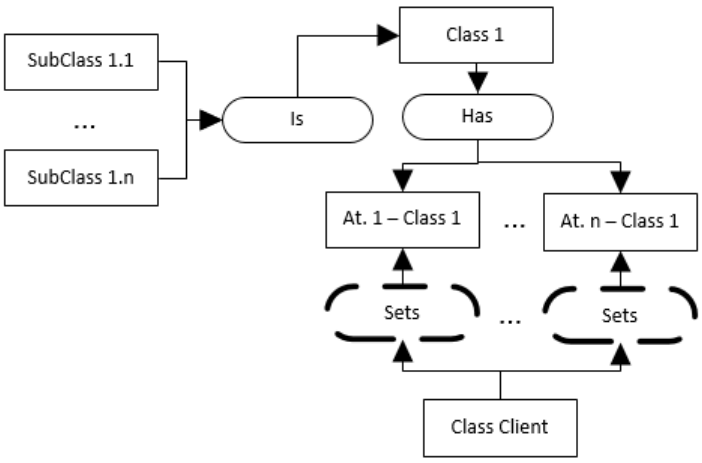


Fuente: Elaboración propia.

4.2.3 Representación del problema general del PD *Builder* en EP.

En esta Sección se representa, de una forma gráfica, el problema que resuelve el PD *Builder* a partir de las descripciones encontradas en la literatura (Gamma *et al*., 1994; Pree y Sikora, 1995). Este patrón se usa en situaciones donde se emplee el mismo proceso de creación para distintas representaciones de un objeto que, además, se controla con estructuras de flujo. En la Figura 4-44 se presenta la estructura principal de la situación problemática. Allí se puede observar que existen dos relaciones dinámicas que operan sobre el concepto "*Class 1*". Estas relaciones dinámicas, por herencia, llegarán a cada uno de sus conceptos hijos.

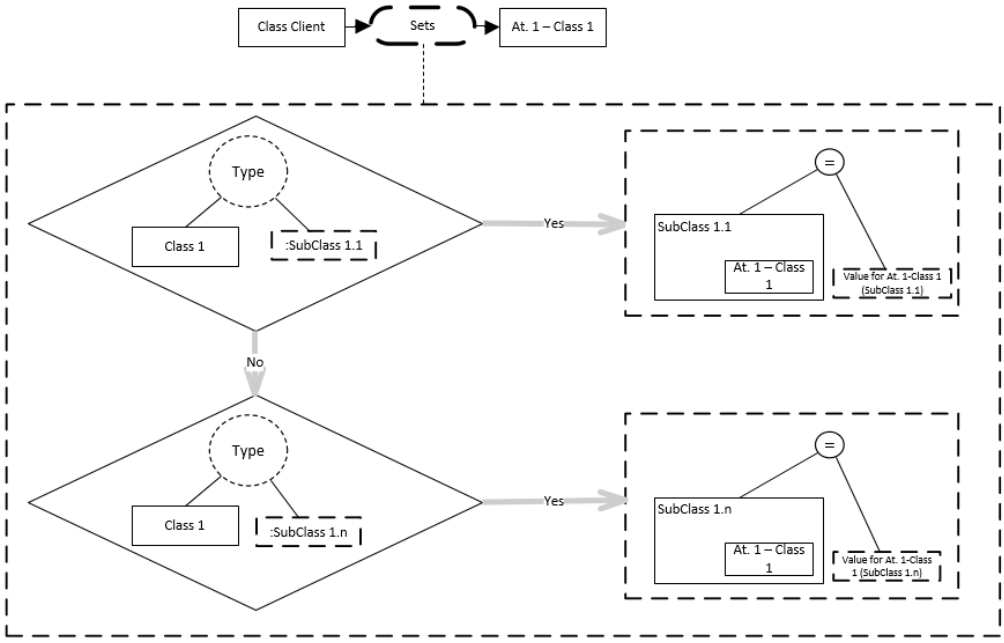
Figura 4-44: Representación del problema general del PD *Builder* en EP (1/3).



Fuente: Elaboración propia.

En la Figura 4-45 se muestra la especificación de la tríada dinámica "*Class Client Sets At.1-Class 1*". En esta especificación se puede observar que existen estructuras de control para determinar el tipo de clase específico del concepto "*Class 1*" (sólo uno de los tipos de su concepto hijo) para determinar el comportamiento de la relación dinámica.

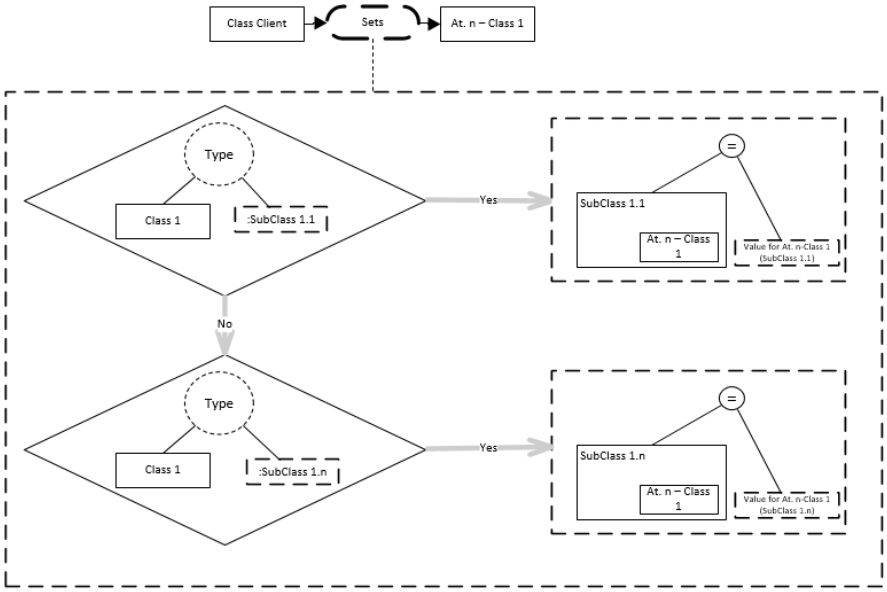
Figura 4-45: Representación del problema general del PD *Builder* en EP (2/3).



Fuente: Elaboración propia.

Asimismo, en la Figura 4-46 se presenta la especificación de la tríada dinámica "*Class Client Sets At.n-Class 1*", donde se puede apreciar una estructura exactamente igual a la de la relación dinámica anterior, pero que opera sobre otro de los atributos del concepto "*Class 1*". En resumen, con esta representación se exhibe la necesidad de tener estructuras de control para cada una de las relaciones dinámicas del sistema. Si se quiere agregar una nueva relación dinámica, se deben implementar tantos condicionales como clases hijas del concepto "*Class 1*" existan en el dominio. Por otro lado, Si se agrega un concepto hijo nuevo al dominio, se deben modificar todas las relaciones dinámicas con condicionales para especificar el comportamiento cuando el tipo del concepto "*Class 1*" se defina como uno de los nuevos conceptos hijos.

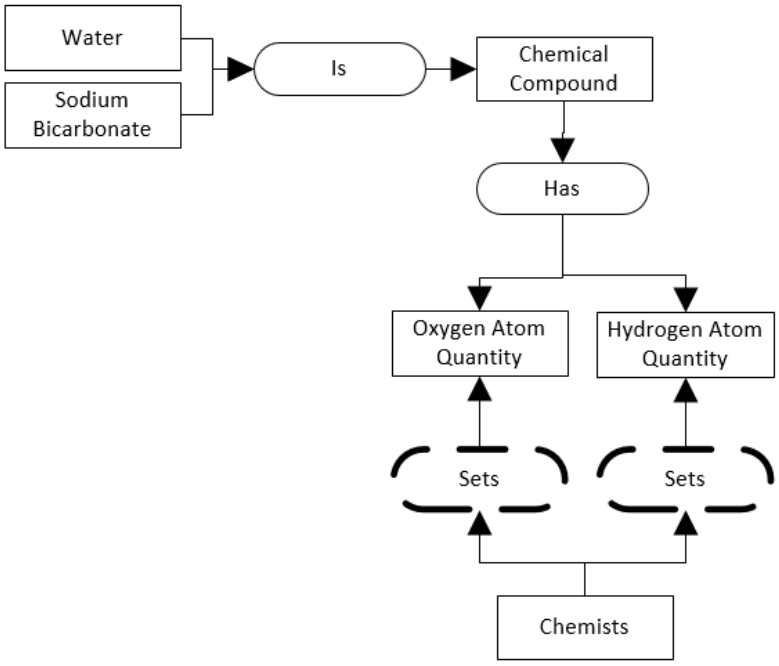
Figura 4-46: Representación del problema general del PD *Builder* en EP (3/3).



Fuente: Elaboración propia.

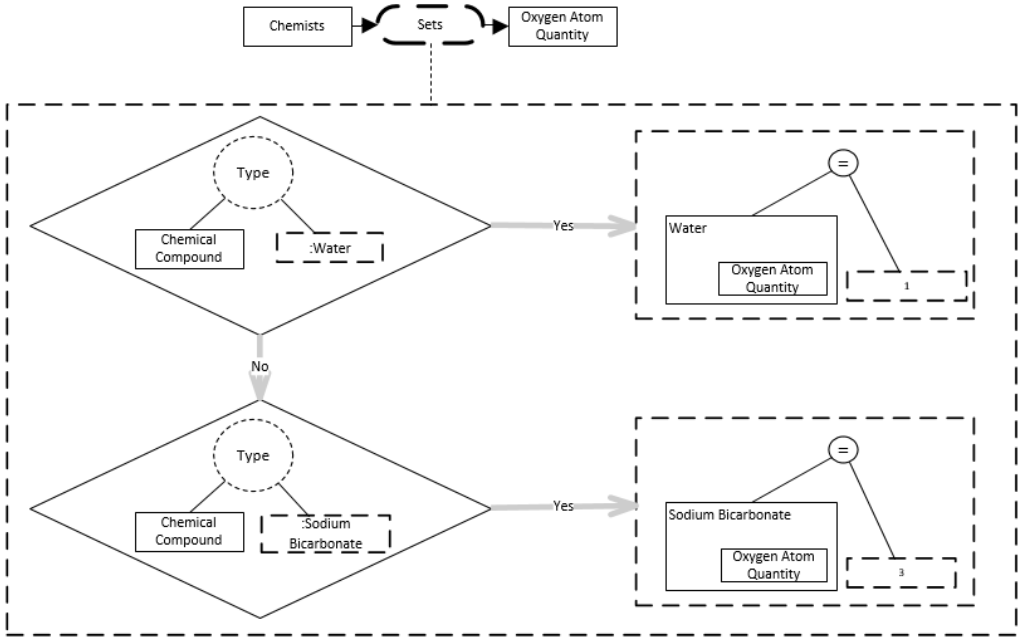
Esta aproximación limita al sistema y genera una alta dependencia entre las partes que lo componen. En la Figura 4-47 se ejemplifica la estructura del problema general del PD *Builder* en un contexto científico. Este ejemplo permite describir un fragmento de un sistema para la creación de compuestos químicos que contienen átomos de hidrógeno y oxígeno. Se puede observar que el proceso de creación o caracterización de los compuestos químicos se hace por medio de las dos mismas relaciones dinámicas. A pesar de esto, los componentes creados, son de tipos diferentes. Finalmente, en la Figura 4-48 y la Figura 4-49 se caracterizan las relaciones dinámicas del fragmento del sistema. Allí, se asignan los valores correspondientes con base en el tipo de componente químico que se trabaja.

Figura 4-47: Ejemplo de la representación del problema genérico del PD *Builder* (1/3).



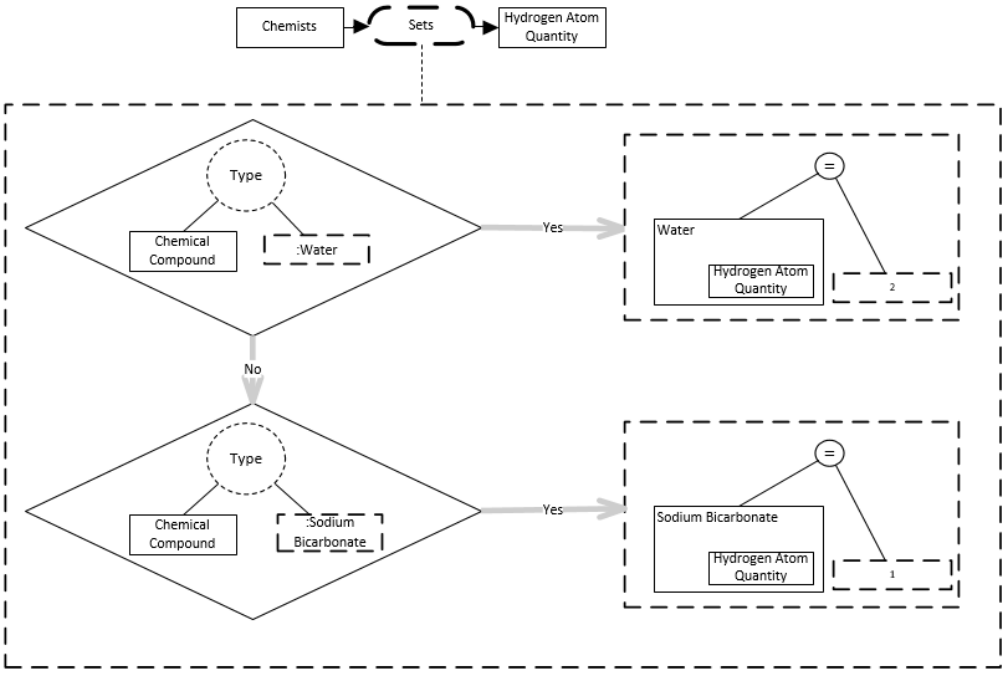
Fuente: Elaboración propia.

Figura 4-48: Ejemplo de la representación del problema genérico del PD *Builder* (2/3).



Fuente: Elaboración propia.

Figura 4-49: Ejemplo de la representación del problema genérico del PD *Builder* (3/3).



Fuente: Elaboración propia.

4.2.4 Representación del patrón de diseño *Builder* en Esquemas Preconceptuales.

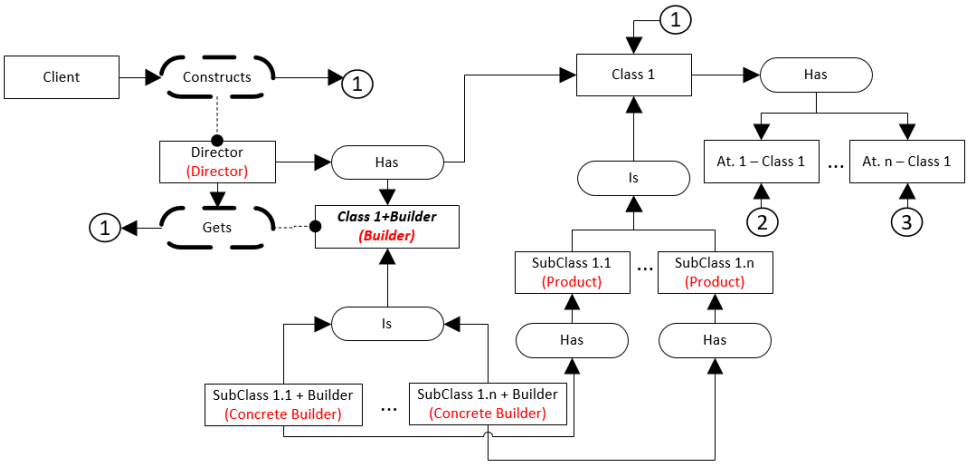
Esta arquitectura le permite al sistema implementar varias representaciones de un mismo objeto por medio del mismo proceso de creación y caracterización (Shalloway y Trott, 2004). A continuación, se listan los elementos principales de la solución que definen Gamma *et al.* (1994):

- ***Builder:*** este elemento define una interfaz abstracta que caracteriza la construcción de las partes del Producto. En este elemento, se definen las características comunes de todos los elementos de tipo *Concrete Builder.*
- ***Concrete Builder:*** el objetivo de este elemento es implementar la construcción de los productos del sistema. Además, permite definir una función para la obtención del producto terminado.
- ***Director:*** este elemento es el puente de comunicación entre el cliente y el resto del patrón. Su función es crear un producto por medio de la interfaz que proporciona el *Builder*.

- ***Product:*** el elemento *Product* representa y permite definir el objeto complejo que se crea. Incluye los atributos y clases que lo conforman.

En la Figura 4-50 se representa la primera parte del PD *Builder.* Este fragmento contiene los elementos principales (resaltados en color rojo) y las relaciones entre conceptos que definen su arquitectura general. Esta representación contiene los elementos presentados en la sección anterior y muestra cómo interactúan con los nuevos elementos propios del PD.

Figura 4-50: Representación del PD *Builder* en EP (1/6).

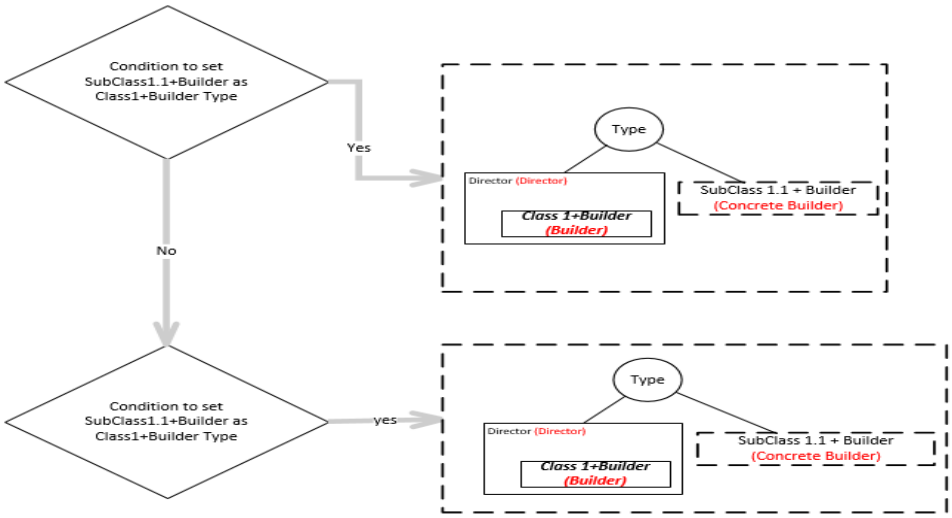


Fuente: Elaboración propia.

En la Figura 4-51 se presenta la estructura de control de flujo que define qué tipo de objeto se quiere construir. A partir de este condicional, se define qué objetos interactúan en el resto del patrón.

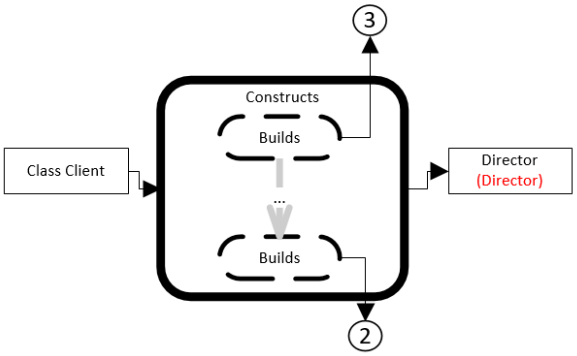
En la Figura 4-52 se muestra la especificación de la relación dinámica "*Constructs*" que dispara toda la acción de la creación de un producto específico. Dentro de su especificación, se ejecutan todas las operaciones, de una manera ordenada, que describen el proceso de creación del producto.

Figura 4-51: Representación del PD *Builder* en EP (2/6).



Fuente: Elaboración propia.

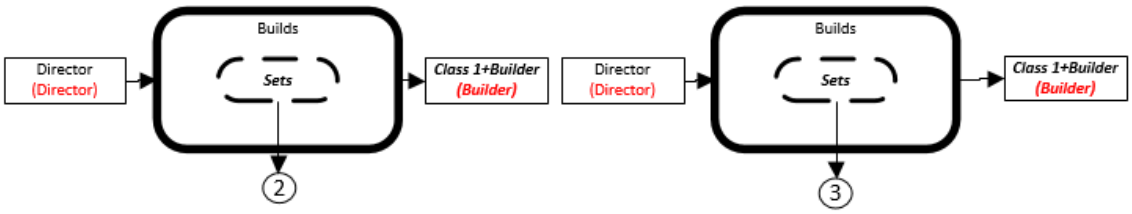
Figura 4-52: Representación del PD *Builder* en EP (3/6).



Fuente: Elaboración propia.

En la Figura 4-53 se muestra la especificación de las relaciones dinámicas "*Builds*" que componen la especificación de la relación dinámica "*Constructs*". Estas operaciones afectan directamente a los atributos del producto que se construye. Se debe tener en cuenta que, en el sistema, ya se definió un tipo para el concepto "*Class1+Builder (Builder)*" (Véase la Figura 4-51), así que la ejecución de las relaciones dinámicas "*sets*" depende de dicho tipo.

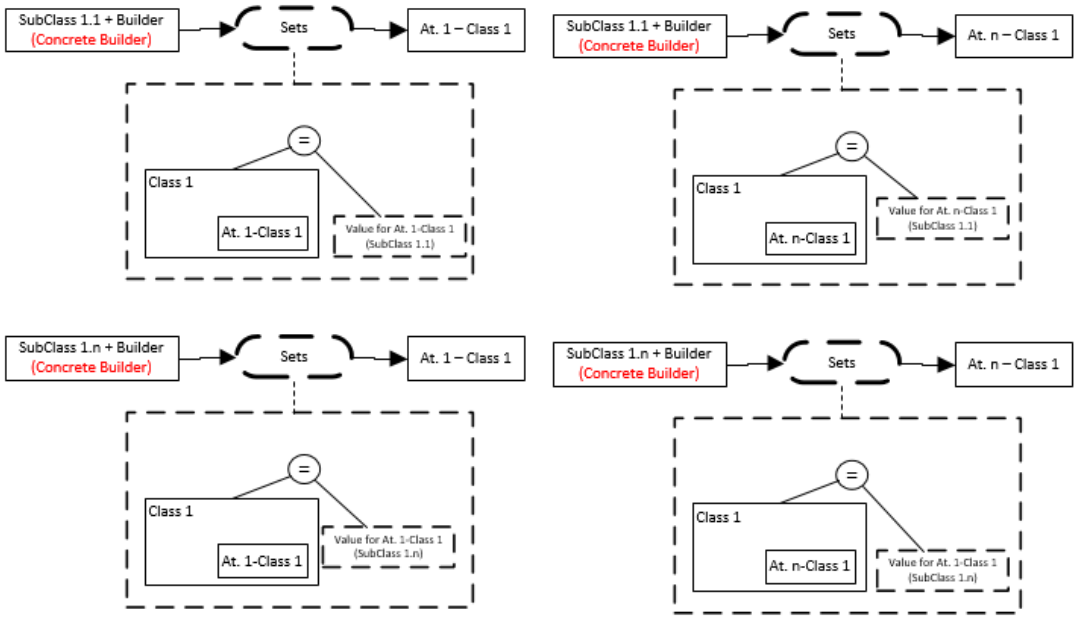
Figura 4-53: Representación del PD *Builder* en EP (4/6).



Fuente: Elaboración propia.

Para cada uno de los conceptos que representan a un *Concrete Builder* debe existir una especificación para las relaciones dinámicas "*Sets*", dado que, por herencia, todos reciben esta relación dinámica y deben definir su implementación.

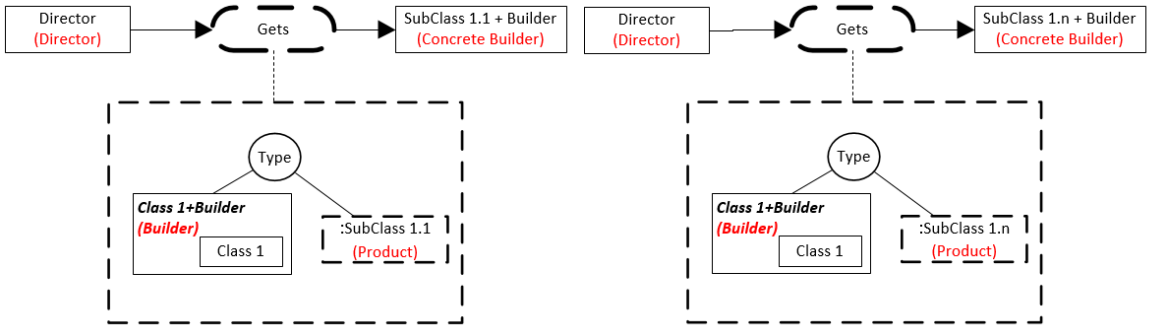
Figura 4-54: Representación del PD *Builder* en EP (5/6).



Fuente: Elaboración propia.

Finalmente, se especifican las relaciones dinámicas "*gets*" que devuelven el producto totalmente construido con base en la directriz del *Director.*

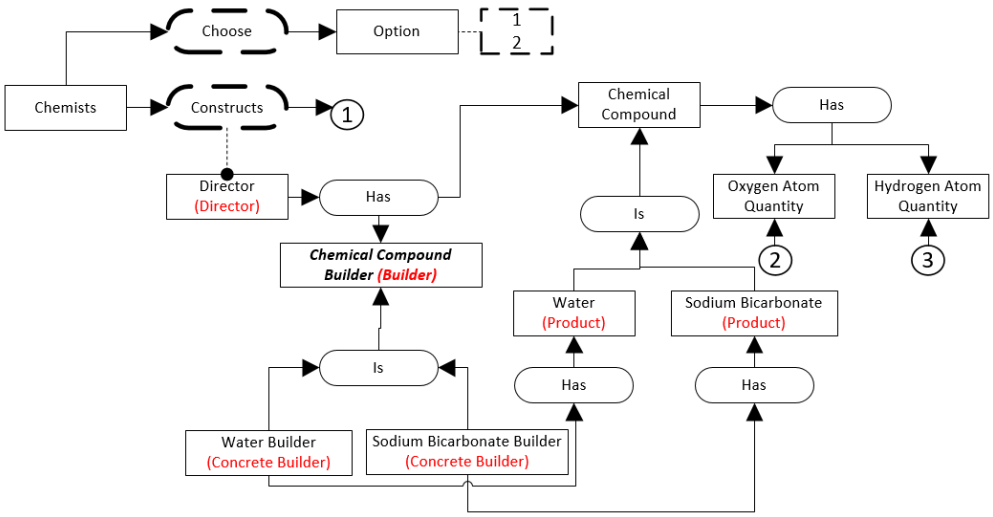
Figura 4-55: Representación del PD *Builder* en EP (6/6).



Fuente: Elaboración propia.

La representación de la arquitectura principal de este PD aplicado sobre un contexto real se ejemplifica en la Figura 4-56. Allí, se unen los conceptos presentados anteriormente en la Figura 4-47 y se añaden los nuevos elementos (resaltados con color rojo) necesarios para la implementación del PD *Builder.*

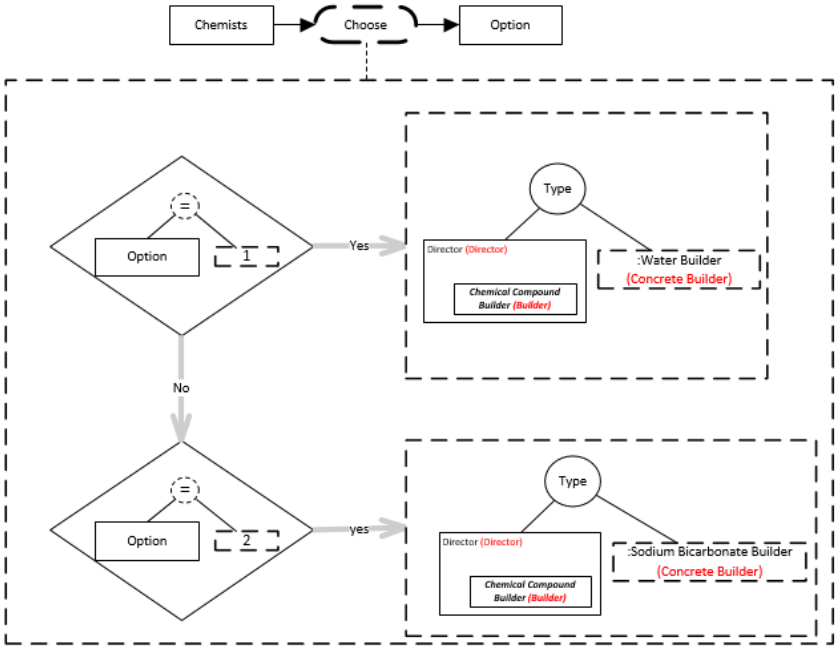
Figura 4-56: Ejemplo del PD *Builder* en EP (1/5).



Fuente: Elaboración propia.

En la Figura 4-57 se propone la especificación de relación dinámica "*Choose*", donde se implementa la estructura de control de flujo que define qué tipo objeto se requiere.

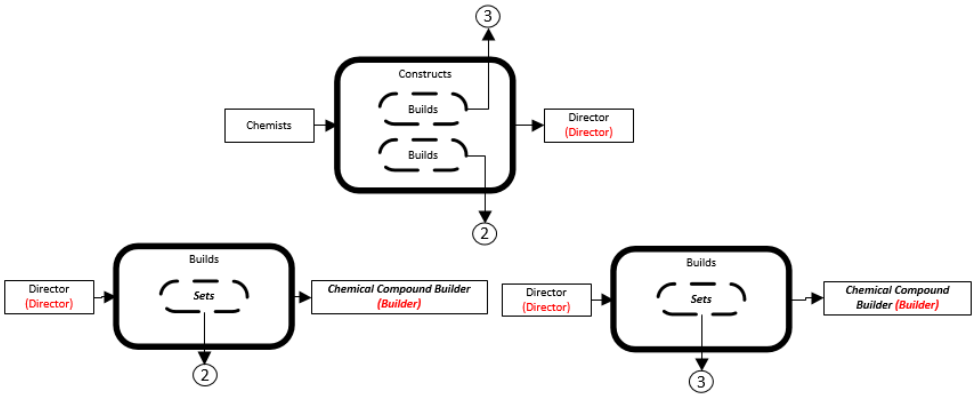
Figura 4-57: Ejemplo del PD *Builder* en EP (2/5).



Fuente: Elaboración propia.

En la Figura 4-58 se especifica la relación dinámica principal "*Constructs*" y, además, la implementación de las relaciones dinámicas que la componen.

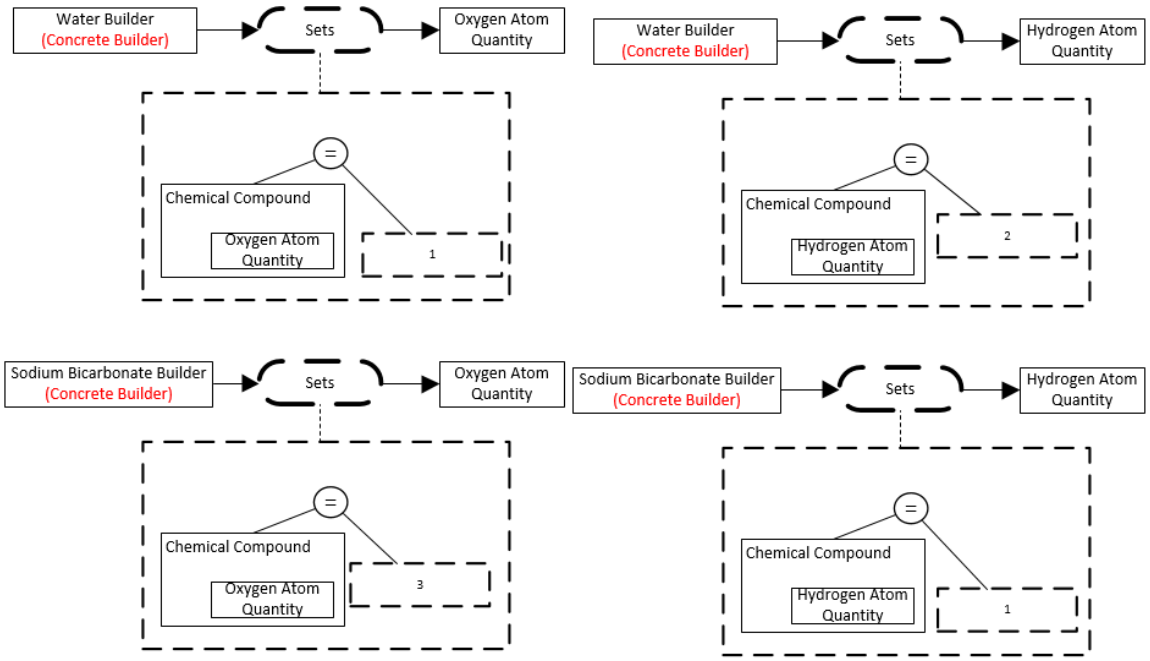
Figura 4-58: Ejemplo del PD *Builder* en EP (3/5).



Fuente: Elaboración propia.

Por otro lado, en la Figura 4-59 se exhibe la especificación individual de las relaciones dinámicas "*Sets*" que hereda cada uno de los conceptos hijo para la creación del elemento *Product* final.

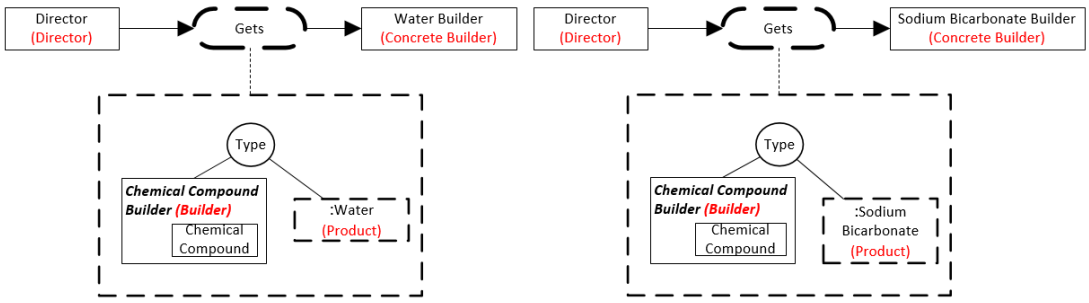
Figura 4-59: Ejemplo del PD *Builder* en EP (4/5).



Fuente: Elaboración propia.

Por último, se especifican las relaciones dinámicas "*gets*" en la Figura 4-60, cuyo objetivo es retornar el concepto *Product* adecuado (*Water* o *Sodium Bicarbonate*).

Figura 4-60: Ejemplo del PD *Builder* en EP (5/5).



Fuente: Elaboración propia.

Este patrón de diseño desacopla la responsabilidad de la creación de objetos complejos. Además, le aporta flexibilidad al sistema y le da la posibilidad de representar un objeto de diversas formas usando el mismo proceso de construcción.

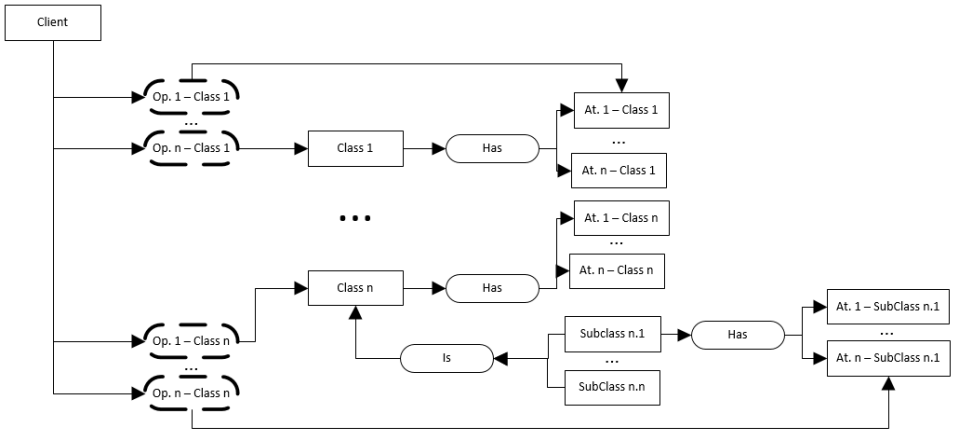
4.3 Representación de los patrones de diseño estructurales en esquemas preconceptuales.

En esta Sección se proponen las representaciones de los patrones de diseño *Facade* y *Decorator* en esquemas preconceptuales. Además, se representan y ejemplifican los problemas generales de dichos patrones.

4.3.1 Representación del problema general del patrón de diseño *Facade* en esquemas preconceptuales.

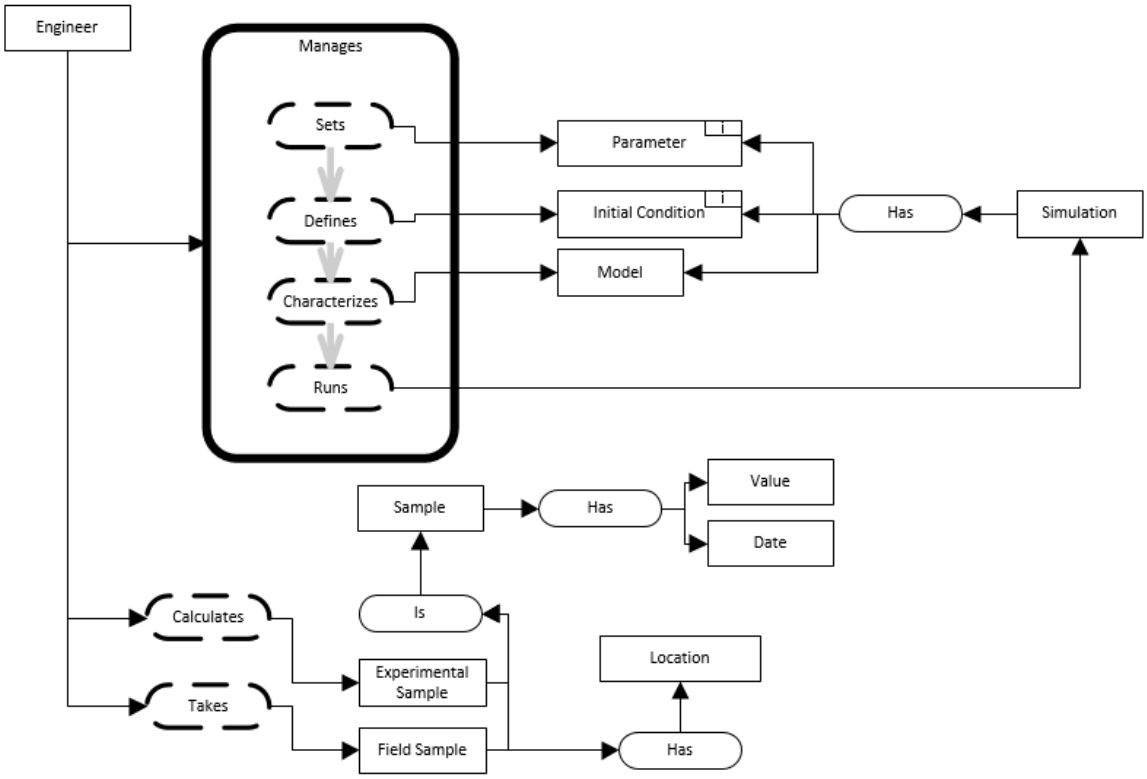
En esta sección se representa en esquemas preconceptuales el problema que soluciona el patrón de diseño *Facade* con base en los lineamientos que describen Gamma *et al.* (1994). Este patrón de diseño surge de la necesidad de reducir la complejidad de un sistema. Esta necesidad se refleja en clases que se relacionan con muchos conceptos del sistema, en aplicaciones que no están debidamente divididas en módulos o subsistemas y sistemas que tienen muchas dependencias entre una clase cliente y sus demás elementos (Stelting y Maassen, 2002). En la Figura 4-61 se representa una situación genérica que agrupa las problemáticas descritas anteriormente. Allí, existe un concepto ("*Client*") del cual dependen todas las relaciones dinámicas y, además, interactúa con muchos de los conceptos del sistema. Por otro lado, en la Figura 4-62 se ejemplifica este tipo de estructura con el fragmento de un sistema donde existe un ingeniero que realiza múltiples tareas sobre todo la aplicación. En este ejemplo, todas las responsabilidades se asignan a un solo concepto y, a su vez, éste interactúa con varios de los conceptos del dominio.

Figura 4-61: Representación gráfica del problema del patrón de diseño *Facade* en esquemas preconceptuales.



Fuente: Elaboración propia.

Figura 4-62: Ejemplo de la representación del problema del patrón de diseño *Facade* en esquemas preconceptuales.



Fuente: Elaboración propia.

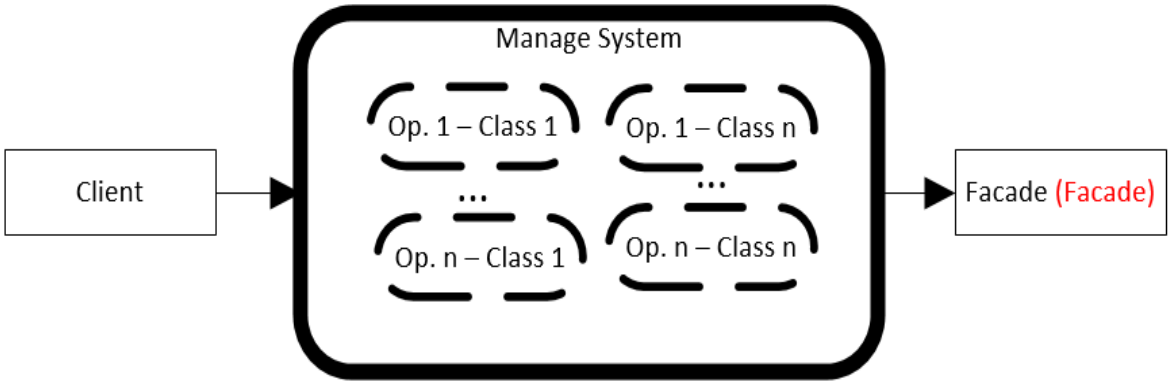
Esta arquitectura muestra múltiples dependencias desde un concepto hacia otros conceptos del sistema, le sobrecarga la responsabilidad sólo a un concepto y le da características de alto acople al sistema poco deseables. Si se quiere agregar funcionalidades adicionales al sistema, cada uno de estos cambios implica una modificación a la clase "*Client*", llevando rápidamente a un sistema difícil de mantener y poco reutilizable (Stelting y Maassen, 2002).

4.3.2 Representación del patrón de diseño *Facade* en esquemas preconceptuales.

El patrón de diseño *Facade* se propone como una interfaz unificada entre una clase cliente y el resto del sistema. Esta interfaz traduce las peticiones del cliente en llamadas a las funciones de los otros conceptos del sistema. Por otro lado, este patrón de diseño permite manejar sistemas complejos de una manera más sencilla. Usualmente, cada interacción

del cliente con la aplicación, involucra sólo una porción del sistema. El patrón *Facade* proporciona métodos más sencillos para efectuar las operaciones complejas que existen en la definición original del dominio (Shalloway y Trott, 2004). Este patrón de diseño sólo añade una clase al sistema mediante su implementación, la clase *Facade.* En la Figura 4-63 se presenta la primera parte de la representación del patrón de diseño *Facade.* Este fragmento corresponde a la nueva interacción del concepto "*Client*" con la aplicación. Se puede observar que, ahora, este concepto no interactúa directamente con el resto del sistema, pues sólo lo hace con el nuevo concepto del sistema ("*Facade*").

Figura 4-63: Representación gráfica del patrón de diseño *Facade* (1/2).

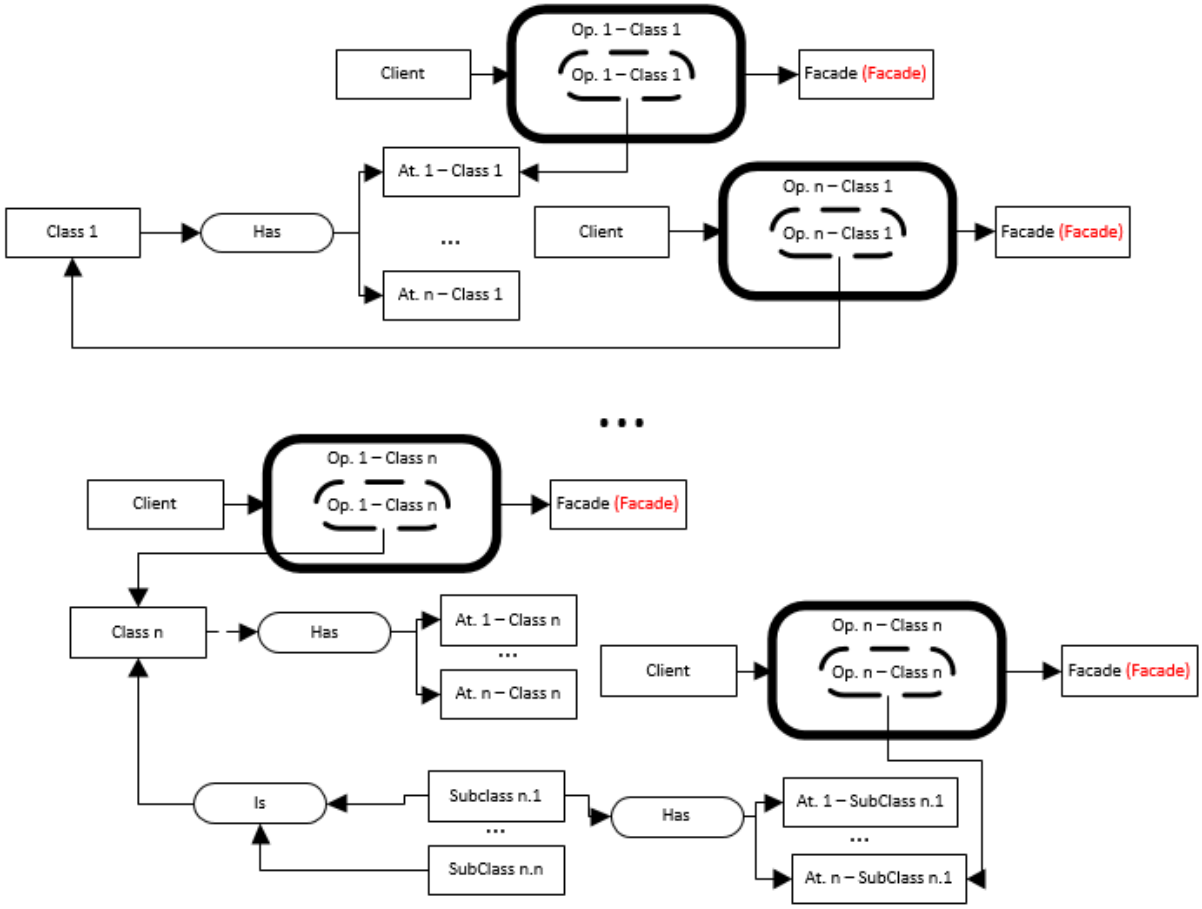


Fuente: Elaboración propia.

En la Figura 4-64 se muestra cómo el concepto "*Facade*" interactúa con el sistema con base en las relaciones dinámicas que salen desde el concepto "*Client*".

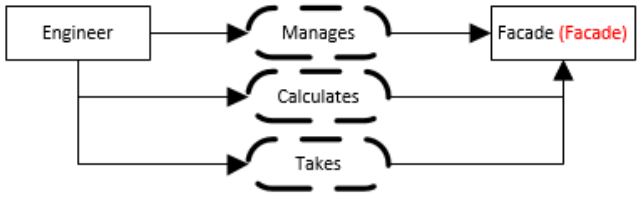
Por otro lado, en la Figura 4-65 se resumen las tríadas dinámicas que representan la interacción total del concepto "*Engineer*" y el sistema por medio del concepto "*Facade*".

Figura 4-64: Representación gráfica del patrón de diseño *Facade* (2/2).



Fuente: Elaboración propia.

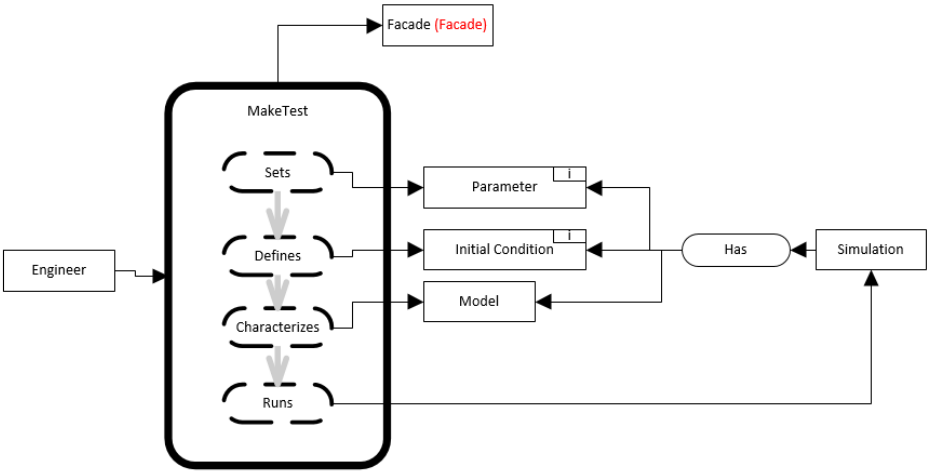
Figura 4-65: Ejemplo de la representación del patrón de diseño *Facade* (1/3).



Fuente: Elaboración propia.

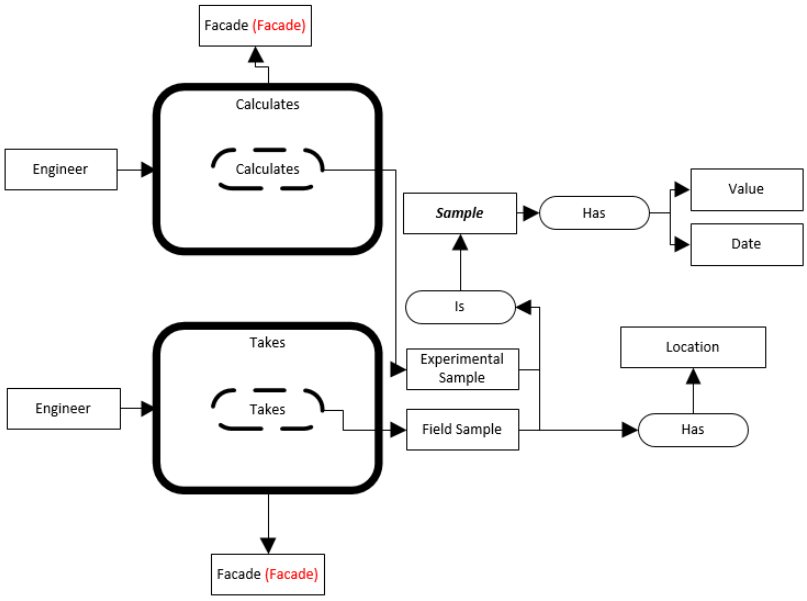
Finalmente, en la Figura 4-66 y la Figura 4-67 se representan las especificaciones de las relaciones dinámicas que componen las tríadas dinámicas que componen el concepto "*Client*" y "*Facade*".

Figura 4-66: Ejemplo de la representación del patrón de diseño *Facade* (2/3).



Fuente: Elaboración propia.

Figura 4-67: Ejemplo de la representación del patrón de diseño *Facade* (3/3).



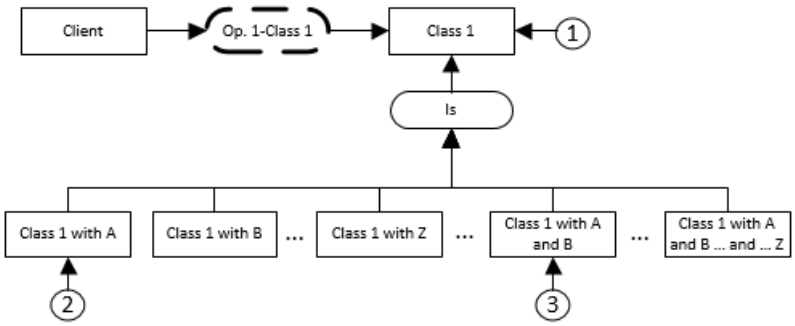
Fuente: Elaboración propia.

4.3.3 Representación del problema general del patrón de diseño *Decorator* en esquemas preconceptuales.

En esta Sección se propone gráficamente el problema del patrón de diseño *Decorator.* Este problema describe una situación donde, por medio de herencia, se logra extender las

características de un objeto (Pree y Sikora, 1995). En la Figura 4-68 se propone, en esquemas preconceptuales, la primera parte del problema genérico que resuelve este patrón, donde se muestra la estructura general que compone el problema.

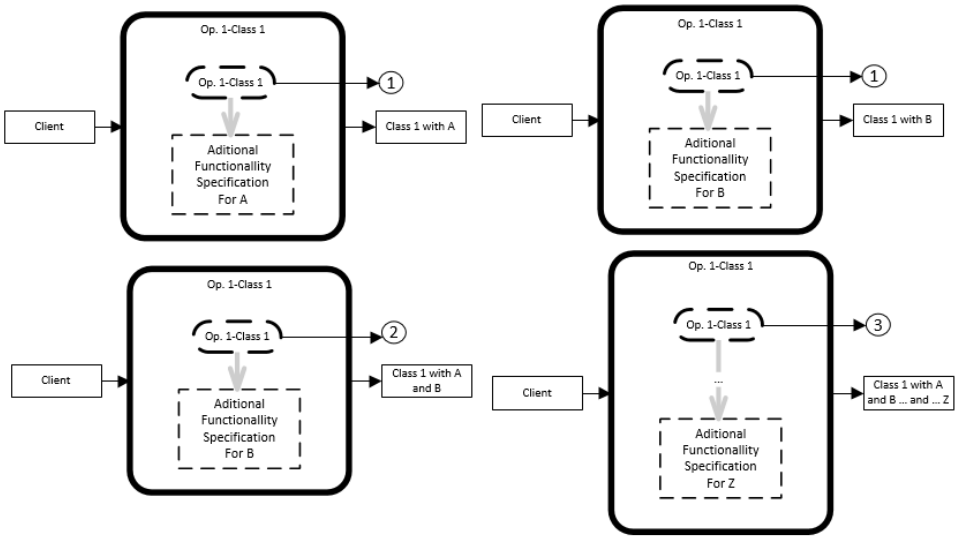
Figura 4-68: Representación gráfica del problema del patrón de diseño *Decorator* (1/3).



Fuente: Elaboración propia.

En la Figura 4-69 se especifica, de forma genérica, la única relación dinámica del sistema. Esta relación se implementa en cada uno los conceptos hijo del concepto "*Class 1*" por herencia. Para definir el problema, cada una de estas especificaciones tiene como punto de partida la especificación de dicha relación a un nivel más general y, además, un agregado que le adiciona nuevas funcionalidades a la relación dinámica conforme cambia el concepto hijo que la implementa.

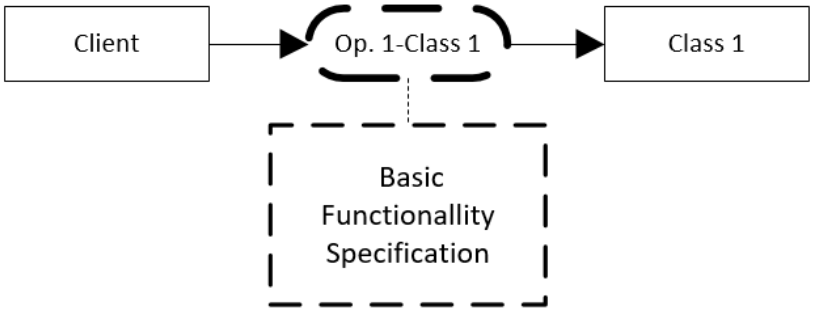
Figura 4-69: Representación gráfica del problema del patrón de diseño *Decorator* (2/3).



Fuente: Elaboración propia.

En la Figura 4-70 se presenta la especificación básica de la relación dinámica "*Op. 1-Class 1*". Esta especificación se incluye en todas las demás especificaciones de la misma relación dinámica en el sistema, acompañada de la funcionalidad adicional que defina cada concepto hijo.

Figura 4-70: Representación gráfica del problema del patrón de diseño *Decorator* (3/3).

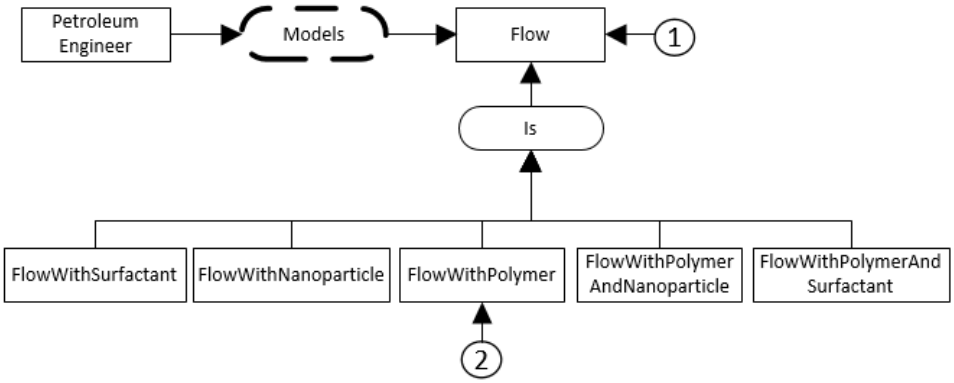


Fuente: Elaboración propia.

Este problema resalta la inflexibilidad y el gran esfuerzo que requiere incluir una nueva variación dentro del dominio. Si se quiere crear una nueva variación del concepto "*Class 1*", se deberán crear combinaciones con los demás conceptos hijos existentes y, además, por cada una de estas combinaciones, se debe modificar la relación dinámica "*Op. 1-Class 1*" que se hereda desde "*Class 1*". Un ejemplo de la estructura general de la representación del problema de este patrón de diseño se propone en la Figura 4-71. En el ejemplo, se observa un concepto ("*Flow*") que puede se puede representar con diversas variaciones, cada una agregando algo a su funcionalidad inicial. En la Figura 4-72 se ejemplifica de forma genérica la especificación de la funcionalidad básica de la relación dinámica "*models*", que sirve como punto de partida para las especificaciones de las variaciones del concepto "*Flow*".

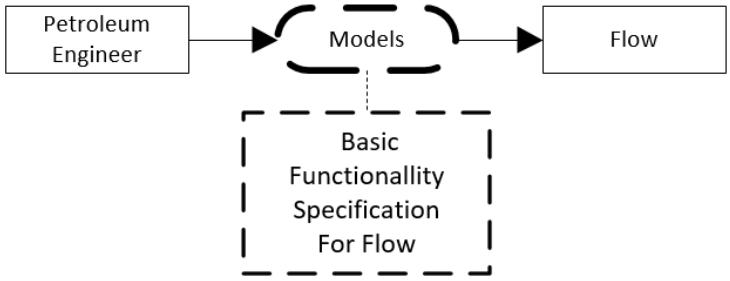
Por otra parte, en la Figura 4-73 y la Figura 4-74 se representa la especificación de la relación dinámica heredada "*models*", con las funcionalidades adicionales de cada una de las variaciones de los conceptos hijos. Se puede apreciar que, en cada una de estas especificaciones, se hace un llamado a versiones más generales de la misma relación dinámica sobre otros conceptos hijo del sistema.

Figura 4-71: Ejemplo de la representación del problema del patrón de diseño *Decorator* (1/4).



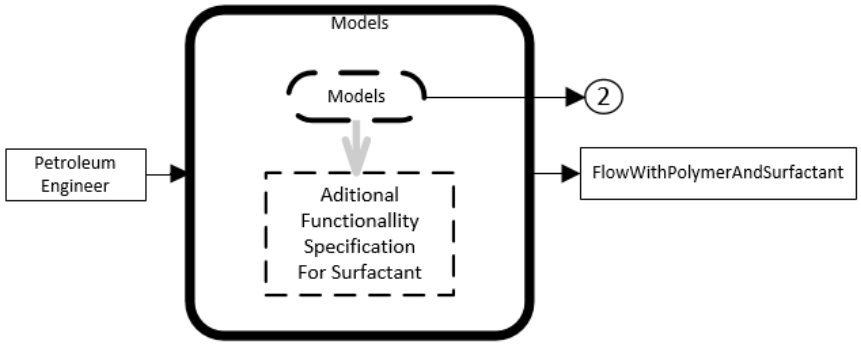
Fuente: Elaboración propia.

Figura 4-72: Ejemplo de la representación del problema del patrón de diseño *Decorator* (2/4).



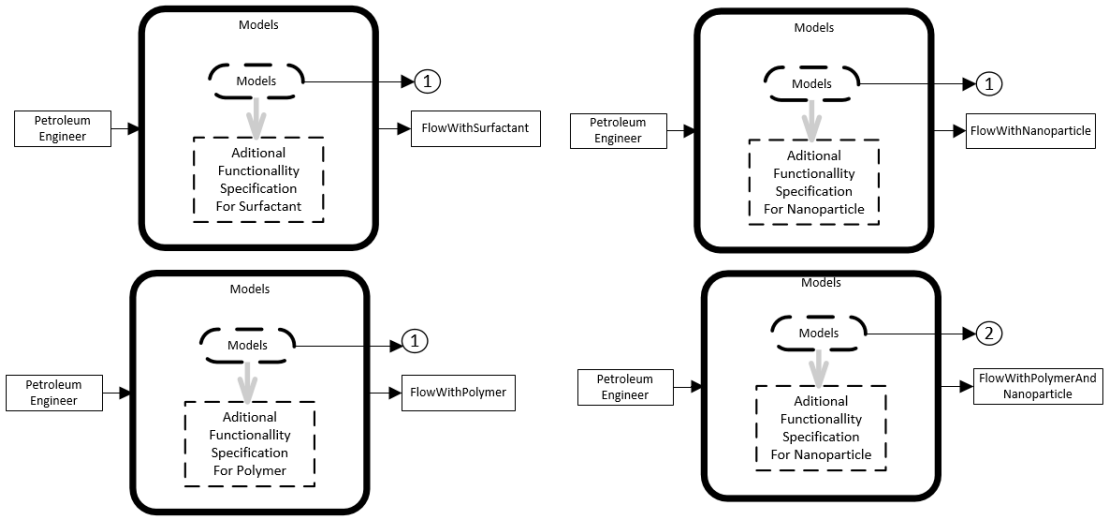
Fuente: Elaboración propia.

Figura 4-73: Ejemplo de la representación del problema del patrón de diseño *Decorator* (3/4).



Fuente: Elaboración propia.

Figura 4-74: Ejemplo de la representación del problema del patrón de diseño *Decorator* (4/4).



Fuente: Elaboración propia.

Si se implementa una nueva relación dinámica sobre el concepto "*Flow*", debe existir una especificación para cada una de las variaciones de éste.

4.3.4 Representación del patrón de diseño *Decorator* en esquemas preconceptuales.

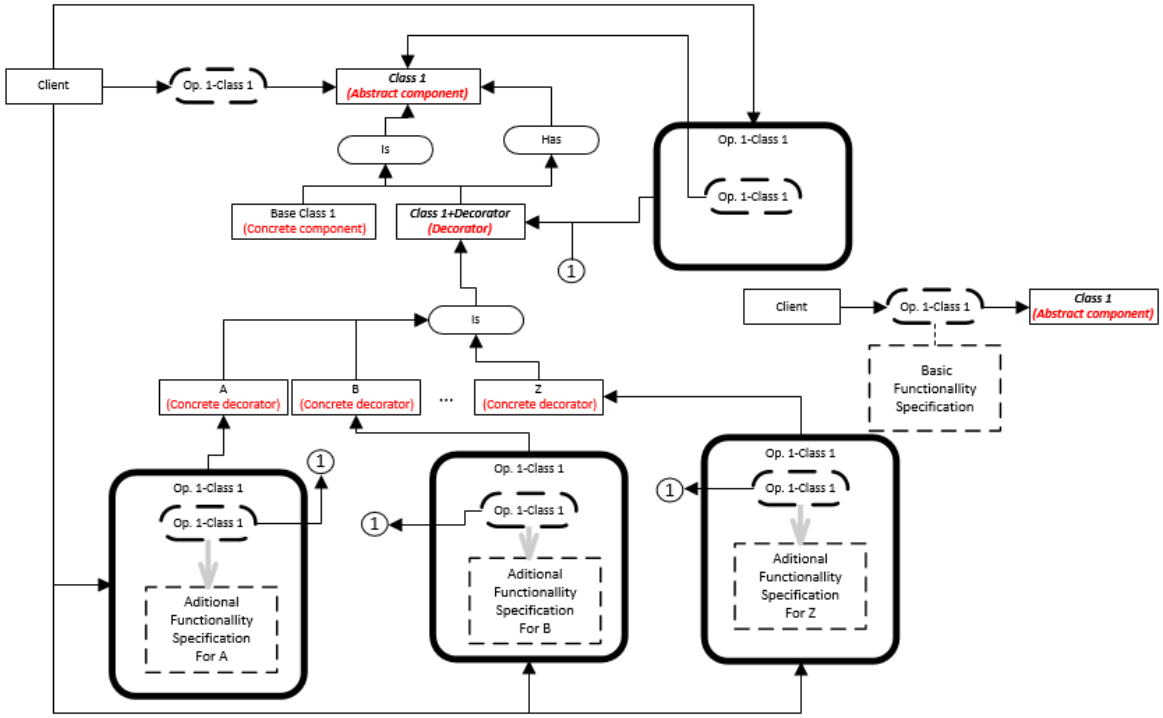
En esta Sección se propone la representación del patrón de diseño *Decorator* con base en las especificaciones de Gamma *et al.* (1994). A continuación, se listan sus elementos principales:

- ***Abstract Component:*** este elemento define la interfaz de los objetos que pueden variar su funcionalidad dinámicamente.
- ***Concrete Component:*** el *Concrete Component* define un objeto al cual se le pueden agregar funcionalidades adicionales. Hereda del elemento *Abstract Component*.
- ***Decorator:*** este elemento mantiene una referencia al *Abstract Component* y define la interfaz para los objetos que cambian la funcionalidad en los *Concrete Component*.
- ***Concrete Decorator:*** el *Concrete Decorator* añade responsabilidades al elemento *Concrete Component.*

En la Figura 4-75 se representa en esquemas preconceptuales el patrón de diseño *Decorator.* En esta representación se puede observar, de forma genérica, sus participantes (resaltados de color rojo) y las relaciones que mantienen entre sí. Este patrón de diseño,

desacopla las características que pueden variar y enriquecer la funcionalidad de la clase base del sistema (“*Class 1*”) para permitir la evolución independiente de cada una de las partes del sistema (Stelting y Maassen, 2002). Se define la clase “*Base Class 1*” como una representación de la clase original “*Class 1*” de donde se desprenden todas las variaciones. Además, se separan las variaciones y sus funcionalidades adicionales y se especifican como individuales en los conceptos “A”, “B” y “Z”.

Figura 4-75: Representación gráfica del patrón de diseño *Decorator*.

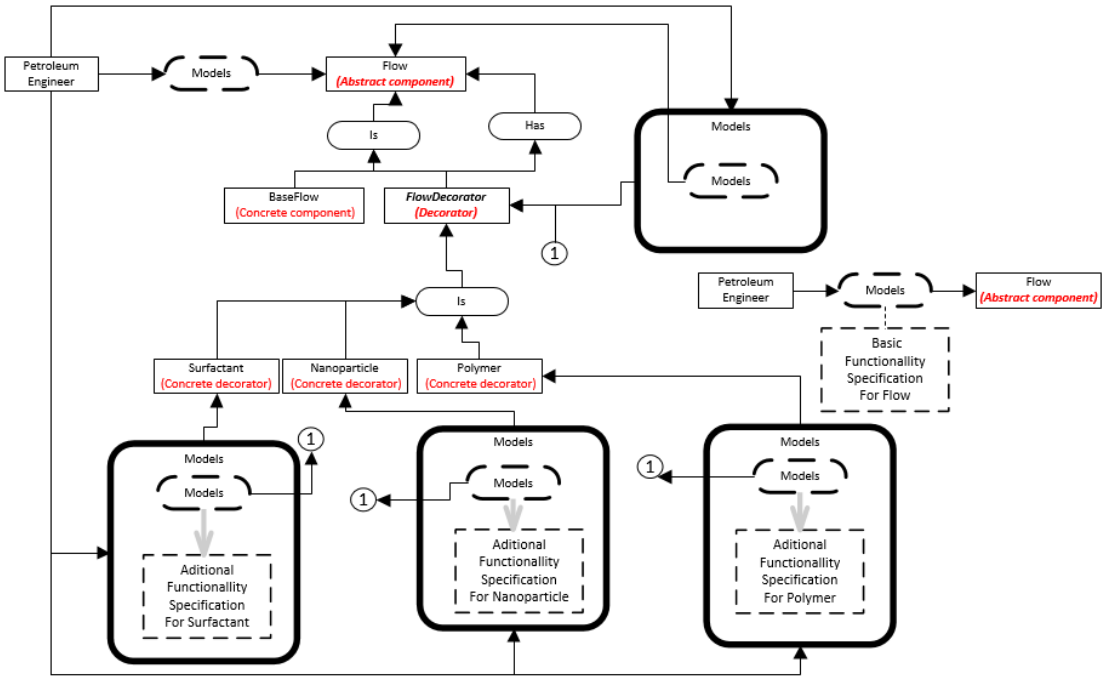


Fuente: Elaboración propia.

Esta arquitectura, permite la inclusión fácil de nuevos tipos de variación o componentes, sin afectar al resto del sistema. Finalmente, en la Figura 4-76 se ejemplifica la implementación del patrón de diseño *Decorator* sobre el contexto científico descrito anteriormente en la Figura 4-71.

En este ejemplo, se resalta el papel de cada uno de los conceptos y cómo interactúan con los demás participantes del sistema.

Figura 4-76: Ejemplo de la representación del patrón de diseño *Decorator* en esquemas preconceptuales.



Fuente: Elaboración propia.

4.4 Representación de los patrones de diseño de comportamiento en esquemas preconceptuales.

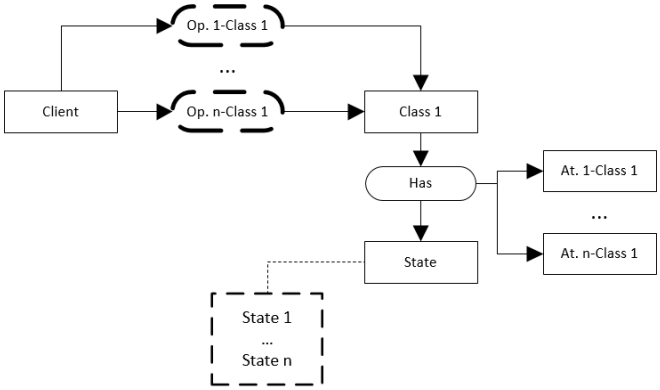
En esta Sección se proponen en esquemas preconceptuales los patrones de diseño *State* y *Strategy*. Adicionalmente, se representan los problemas generales que resuelve cada patrón y se proponen ejemplos bajo un contexto científico, tanto para la representación del problema como para la implementación del patrón de diseño.

4.4.1 Representación del problema general del patrón de diseño *State* en esquemas preconceptuales.

Gamma *et al.* (1994) caracterizan el problema de este patrón de diseño como la necesidad de definir el comportamiento de un objeto por medio de condicionales o *switches* con base en su estado interno. Cuando el comportamiento se define con este tipo de elementos, el sistema no puede variar dinámicamente su comportamiento y, además, en algún punto es necesario intervenir la clase para modificar la especificación de las funciones teniendo en cuenta los estados internos del objeto. En la Figura 4-77 se propone una representación

genérica de este problema. En el dominio del sistema, debe existir un concepto que, entre sus atributos, tenga uno que almacene el estado del objeto.

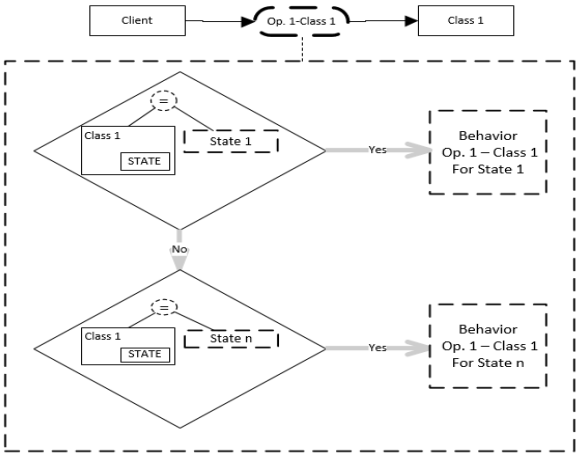
Figura 4-77: Representación del problema general del patrón de diseño *State* (1/3).



Fuente: Elaboración propia.

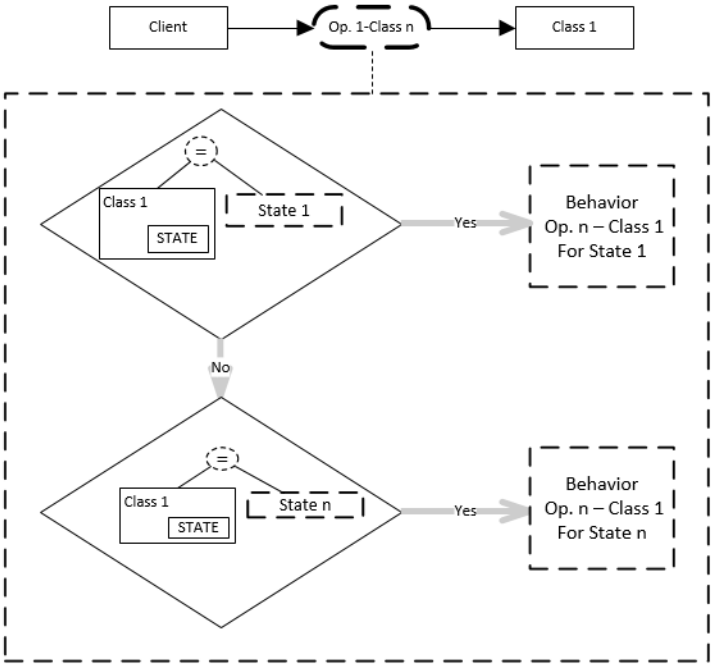
Por otra parte, debe existir al menos una relación dinámica que se conecte a este concepto, cuya especificación involucre condicionales que definan su comportamiento con base en los valores del atributo que almacena su estado. En la Figura 4-78 y la Figura 4-79, se propone la especificación de las relaciones dinámicas "*Op. 1-Class 1*" y "*Op. 1-Class n", respectivamente*. En estas especificaciones se denota el uso de condicionales para definir el comportamiento del sistema con base en los valores del concepto "*State*", que define el estado del objeto.

Figura 4-78: Representación del problema general del patrón de diseño *State* (2/3).



Fuente: Elaboración propia.

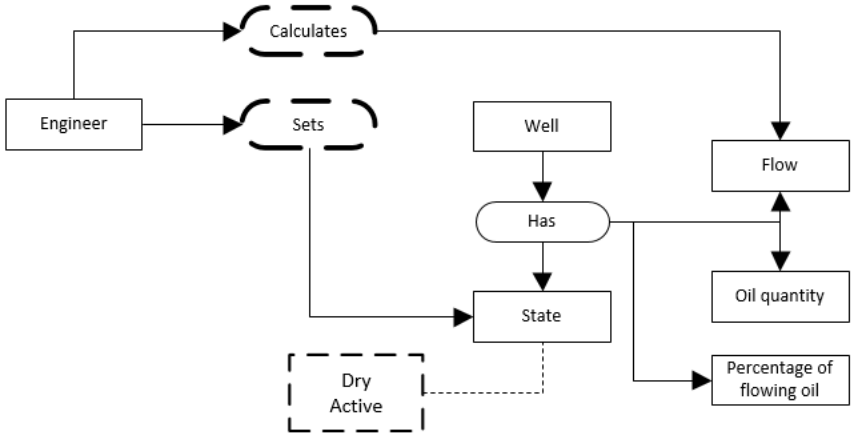
Figura 4-79: Representación del problema general del patrón de diseño *State* (3/3).



Fuente: Elaboración propia.

Este problema resalta la incapacidad del sistema de variar su comportamiento en tiempo de ejecución. Además, obliga a definir por cada relación dinámica presente en el dominio, un condicional que describa cada uno de los estados posibles de un concepto clase. En la Figura 4-80 se propone un ejemplo enmarcado en la estructura del problema genérico de este patrón de diseño.

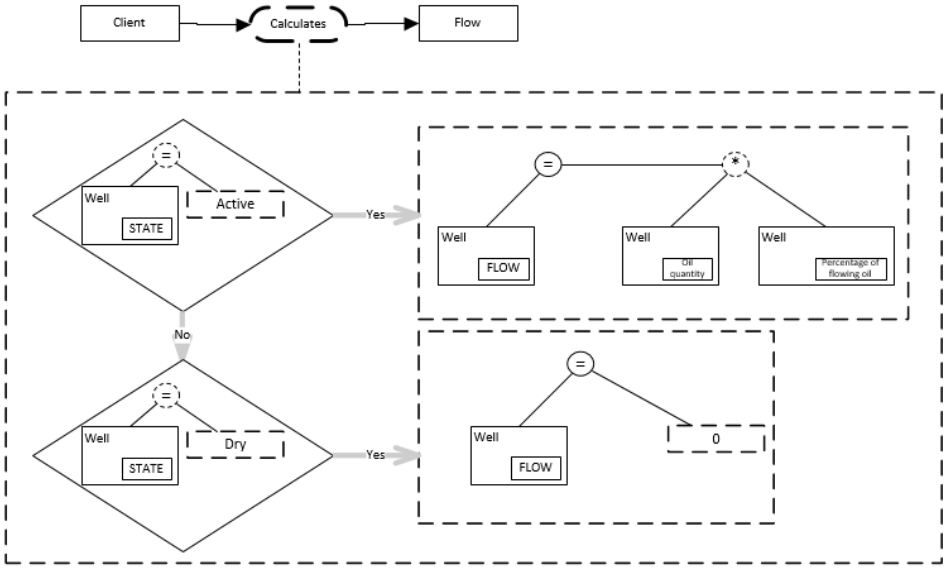
Figura 4-80: Ejemplo de la representación del problema del patrón de diseño *State* (1/2).



Fuente: Elaboración propia.

En la Figura 4-81 se representa la especificación de la relación dinámica "*Calculates*". Allí, se puede observar que la operación se condiciona con el valor del estado del concepto "*Flow*".

Figura 4-81: Ejemplo de la representación del problema del patrón de diseño *State* (2/2).



Fuente: Elaboración propia.

Este ejemplo, cumple con las condiciones propuestas: existe un concepto en el cual uno de sus atributos define un estado ("*Well*") y, además, en la especificación de una de sus relaciones dinámicas ("*Calculates*"), el comportamiento se rige con el valor del atributo que almacena el estado del objeto.

4.4.2 Representación del patrón de diseño *State* en esquemas preconceptuales.

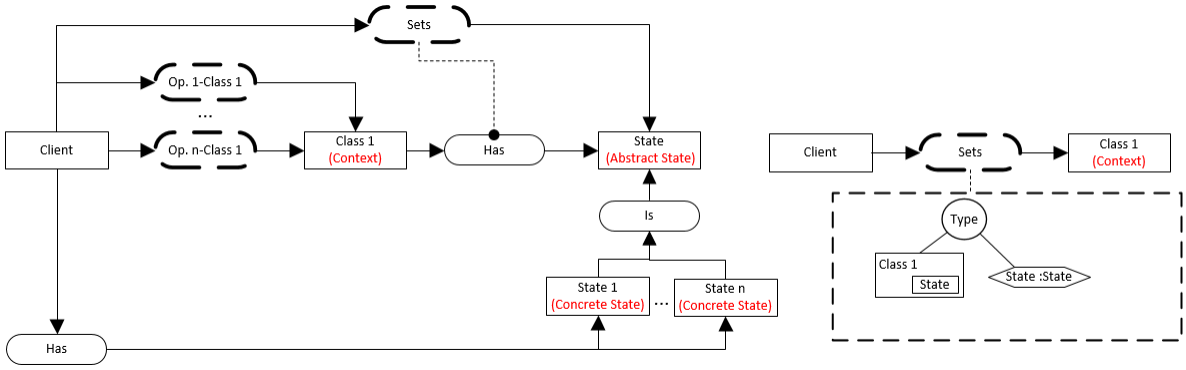
Este patrón de diseño le permite a un objeto variar su comportamiento con base en su estado. Se usa en contextos científicos para maximizar la participación de objetos dinámicos en el sistema (Barbieri *et al*., 2012). Gamma *et al.* (1994) definen los participantes principales en este patrón de diseño:

- ***Context*:** este elemento define la interfaz con la que se comunica el cliente. Usualmente, representa al objeto principal del dominio (el objeto que puede cambiar de estado). El elemento *Context,* además, mantiene una referencia a *State*.

- ***State*:** el elemento *State* define una interfaz que encapsula el comportamiento asociado con un estado específico.
- ***Concrete State*:** este elemento hereda las características del *State.* Cada uno de estas subclases implementa el comportamiento específico de los estados posibles asociados con el contexto.

En la Figura 4-82 se representa la arquitectura general del patrón de diseño y la especificación de la relación dinámica "*sets*". Por otro lado, se resaltan (con color rojo) los participantes del patrón de diseño y las relaciones entre dichos conceptos. Esta representación mantiene los conceptos originales presentados en la representación del problema y adecúa algunos conceptos nuevos para la definición del patrón de diseño. En la representación del patrón de diseño, los valores del concepto "*State*" de la representación del ejemplo, se convierten en clases independientes y permiten definir, por sí solos, el comportamiento correspondiente de las relaciones dinámicas con base en el estado. Ahora, el concepto "*Class 1*" (*Context*), puede variar su comportamiento dinámicamente en tiempo de ejecución directamente desde su estado.

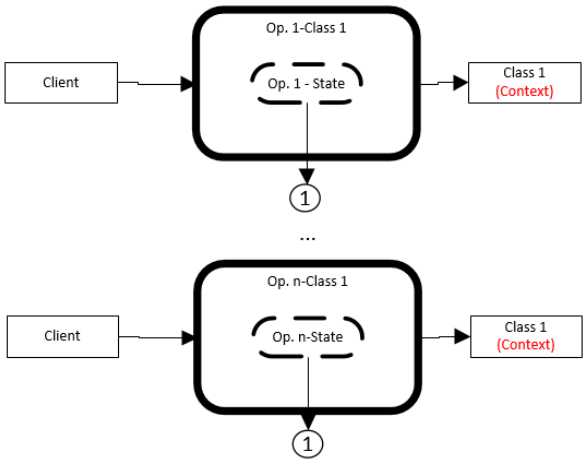
Figura 4-82: Representación gráfica del patrón de diseño *State* (1/3).



Fuente: Elaboración propia.

En la Figura 4-83 se exhiben las especificaciones de las relaciones dinámicas "*Op. 1-Class 1*" y "*Op. n – Class 1*". Estas especificaciones se definen de una manera genérica, ya que, quien las implemente, definirá el comportamiento al interior de su clase.

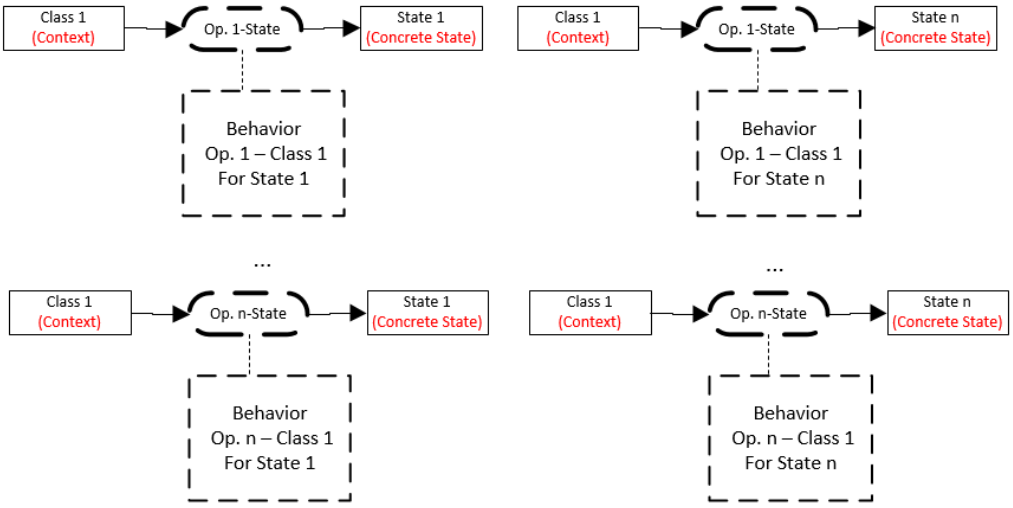
Figura 4-83: Representación gráfica del patrón de diseño *State* (2/3).



Fuente: Elaboración propia.

Siguiendo con las especificaciones de las relaciones dinámicas, en la Figura 4-84 se describe el comportamiento específico de las relaciones dinámicas en cada estado del concepto "*Class 1*". Esto le permite al sistema, variar independientemente su comportamiento sin afectar las clases principales.

Figura 4-84: Representación gráfica del patrón de diseño *State* (3/3).

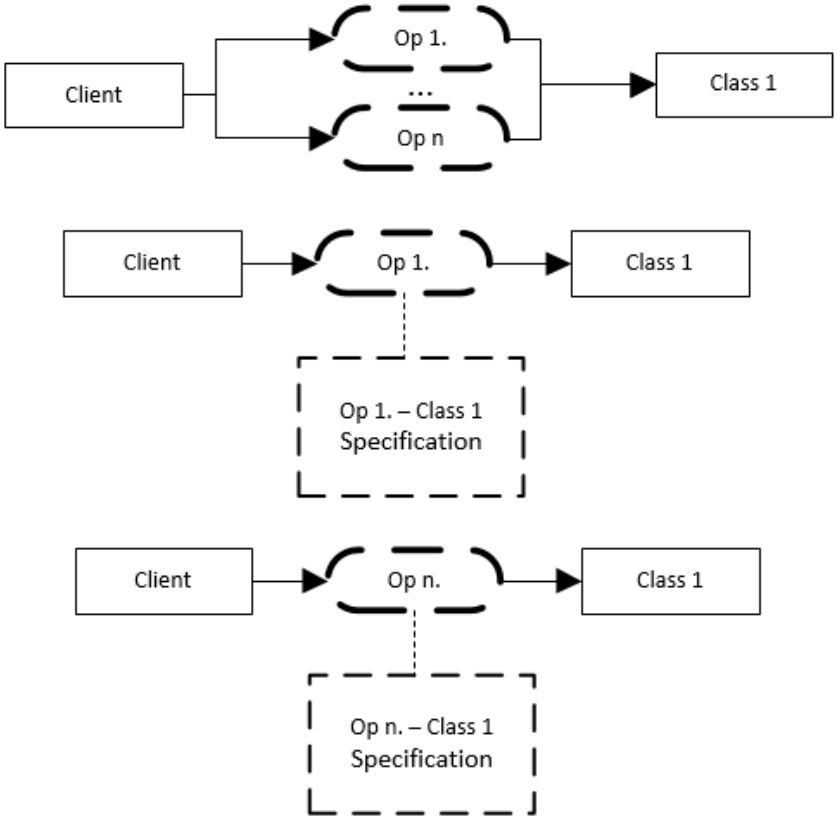


Fuente: Elaboración propia.

4.4.3 Representación del problema general del patrón de diseño *Strategy* en esquemas preconceptuales.

El problema que resuelve el patrón de diseño *Strategy* se caracteriza con un dominio donde existen diversos algoritmos o aproximaciones para llevar a cabo una sola tarea. Al tener varios métodos para ejecutar una acción, la clase cliente se hace más compleja y difícil de mantener según (Gamma *et al.*, 1994). En la Figura 4-85 se propone la representación en esquemas preconceptuales del problema genérico del patrón de diseño *Strategy*. En este esquema se puede apreciar la existencia de una o varias relaciones dinámicas que actúan sobre el mismo concepto. Para esta representación, se debe suponer que dichas relaciones dinámicas completan la misma tarea pero de maneras diferentes.

Figura 4-85: Representación gráfica del problema del patrón de diseño *Strategy*.

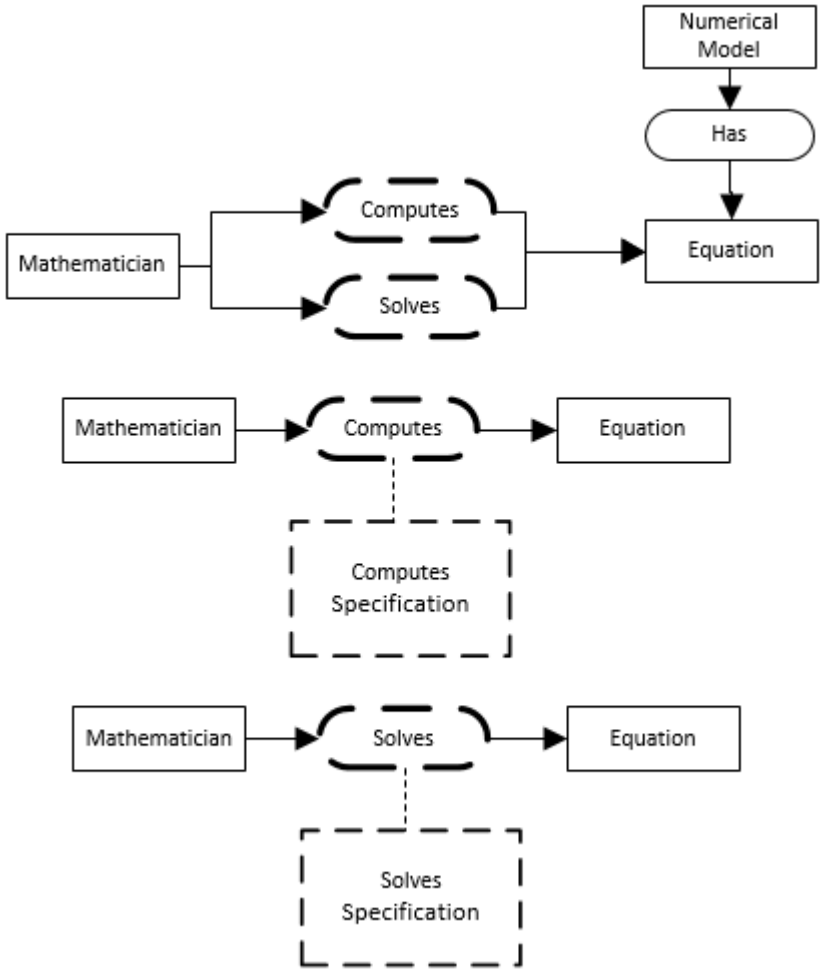


Fuente: Elaboración propia.

En la Figura 4-86 se ejemplifica, bajo un contexto científico, la estructura del problema del patrón de diseño *Strategy* que indicaría la necesidad de implementación de dicho patrón.

En el ejemplo, se asume que las relaciones dinámicas “*computes*” y “*solves*”, realizan la misma tarea pero varían su implementación.

Figura 4-86: Ejemplo de la representación del problema del patrón de diseño *Strategy*.



Fuente: Elaboración propia.

Cuando existen diversos algoritmos para completar una tarea y no se tiene un control de encapsulamiento para ellos, se hace muy complejo el mantenimiento de las clases involucradas cuando el sistema crece (Shalloway y Trott, 2004).

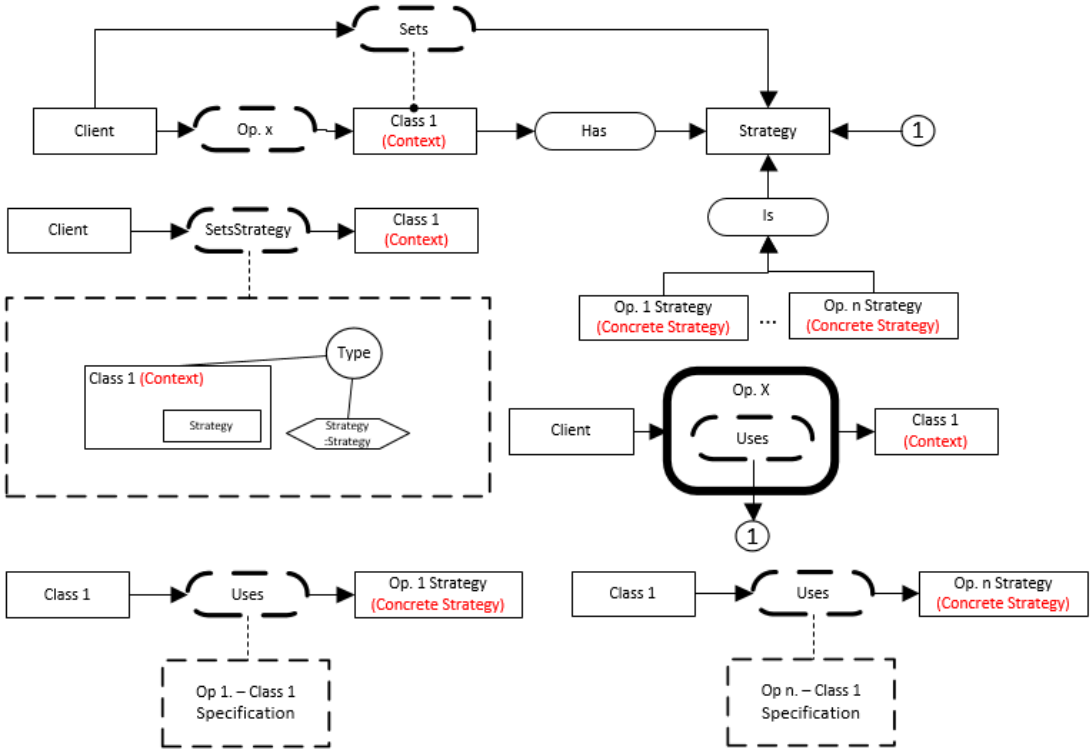
4.4.4 Representación del patrón de diseño *Strategy* en esquemas preconceptuales.

Gamma *et al.* (1994) definen los participantes principales del patrón de diseño *Strategy*:

- ***Strategy:*** este elemento permite definir una interfaz común para todos los algoritmos especificados en el dominio que realicen la misma tarea. El elemento *Context* usa esta interfaz para llamar al algoritmo necesario por medio de una de sus subclases.
- ***Concrete Strategy:*** los elementos *Concrete Strategy* implementan los algoritmos definidos en el elemento *Strategy*.
- ***Context:*** este elemento representa el concepto sobre el que se efectúan las diferentes operaciones en el contexto original. Además, permite mantener una referencia a una *Concrete Strategy* para elegir el algoritmo adecuado para su ejecución.

En la Figura 4-87 se presenta la representación en esquemas preconceptuales del patrón de diseño *Strategy*.

Figura 4-87: Representación gráfica del patrón de diseño *Strategy*.

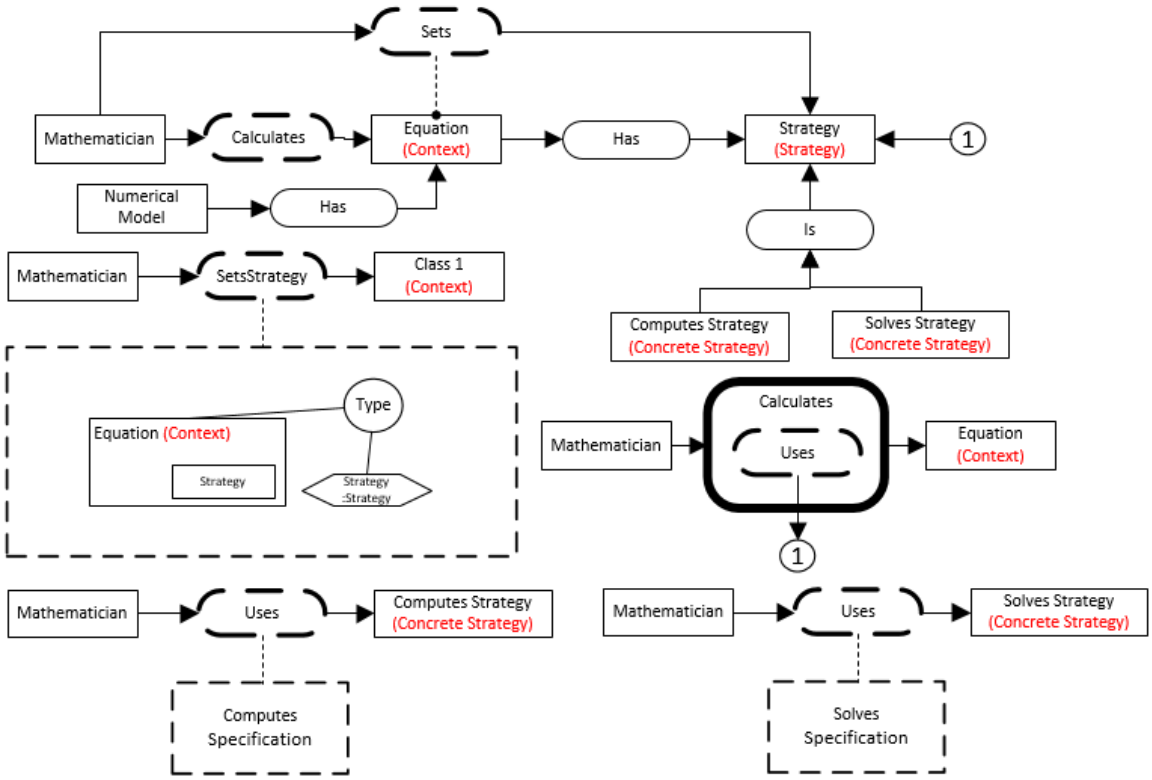


Fuente: Elaboración propia.

En esta representación se resaltan (en color rojo) los nuevos roles de los conceptos del dominio que se representaron como problema en la Figura 4-85. Además, se presenta la arquitectura que debe poseer este conjunto de elementos. Finalmente, se presentan las especificaciones de las relaciones dinámicas inmersas en el sistema. Se puede observar

que las relaciones dinámicas presentadas en el problema, ahora se constituyen como conceptos independientes, cuya responsabilidad es implementar los algoritmos según las peticiones del cliente. Por otro lado, en la Figura 4-88 se ejemplifica la implementación del patrón de diseño *Strategy* en el contexto científico presentado en la Figura 4-86 como precondición para la obtención de esta nueva arquitectura.

Figura 4-88: Ejemplo de la representación del patrón de diseño *Strategy*.



Fuente: Elaboración propia.

5. Validación

Para la validación de esta Tesis de Maestría se usaron ejemplos de la literatura en representación de patrones de diseño para comparar los elementos y colaboraciones presentes en estas representaciones con las estructuras que se presentan en el Capítulo 4.

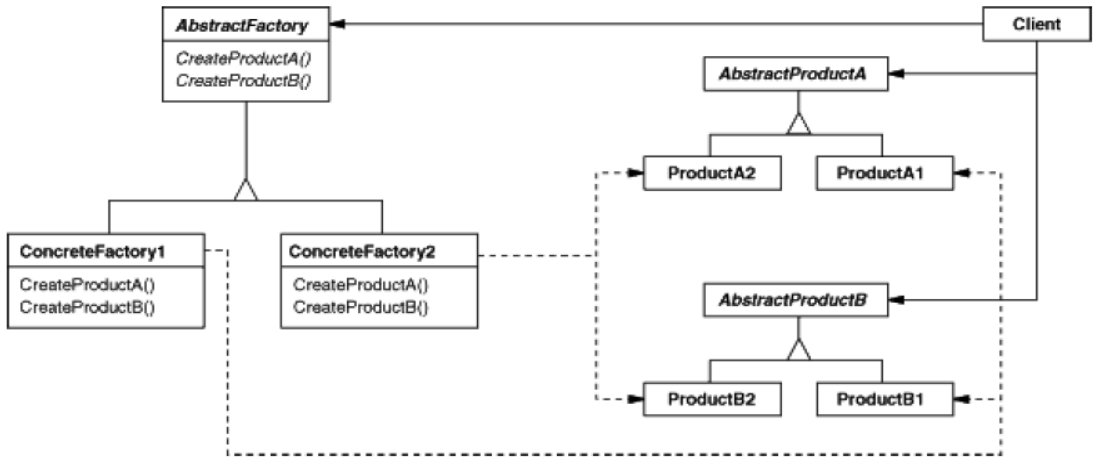
En cada validación se presenta una Tabla que resume los elementos que definen al patrón de diseño y cuáles están presentes en la representación de la literatura y la representación propuesta en esta Tesis. Asimismo, una Tabla que resume las colaboraciones necesarias para la implementación del patrón y su presencia en las representaciones propuestas. Finalmente, se extrae un ejemplo de la literatura y se representa en esquemas preconceptuales para validar la robustez de la propuesta.

5.1 Validación de la representación del patrón de diseño *Abstract Factory*

En esta Sección se comparan las representaciones del patrón de diseño *Abstract Factory* a partir de diagramas de clases (DC) y esquemas preconceptuales (EP). En la Figura 5-1 se presenta la representación del patrón *Abstract Factory* a partir de diagrama de clases. Ésta es la clásica representación que se define en la literatura.

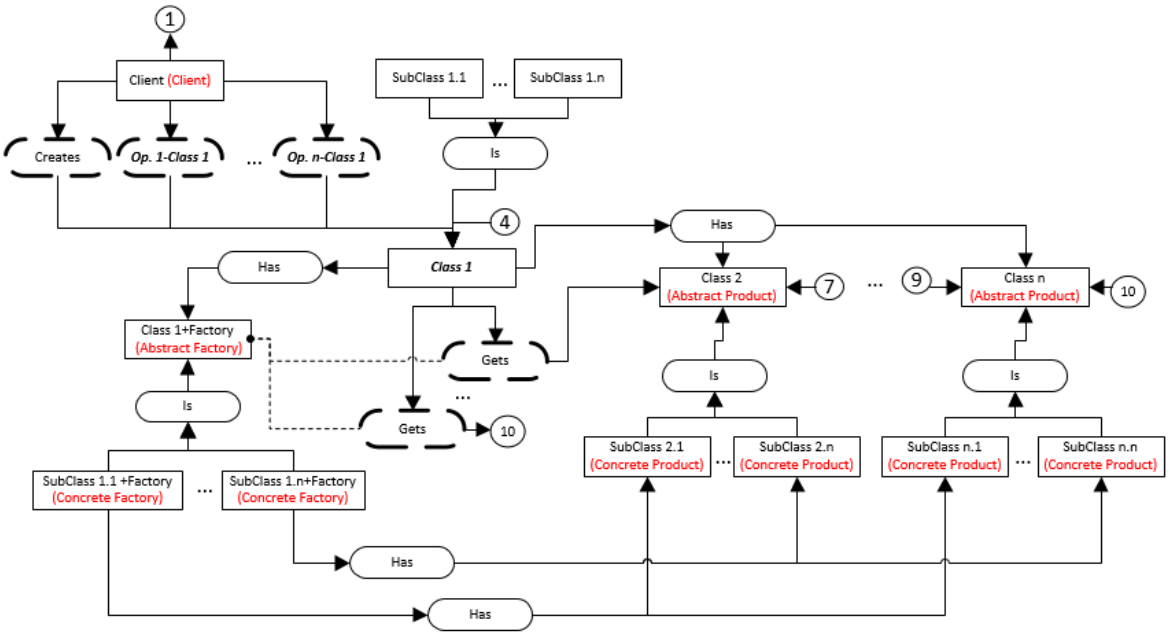
Por otro lado, en la Figura 5-2 se presenta un solo un fragmento de la representación presentada en esta Tesis. En este fragmento, se representa la estructura completa del patrón de diseño presentado en la literatura. Allí se encuentran todos los participantes y relaciones involucrados en la implementación del patrón de diseño *Abstract Factory*.

Figura 5-1: Representación del patrón de diseño *Abstract Factory*.



Fuente: Tomada de Gamma *et al.* (1994).

Figura 5-2: Fragmento de la representación del patrón de diseño *Abstract Factory* en esquemas preconceptuales.



Fuente: Elaboración propia.

En la Sección 4.2 se presentan, adicionalmente, la caracterización de las relaciones dinámicas y otras colaboraciones del sistema que facilitan el entendimiento del patrón de diseño a partir de la representación propuesta en esta Tesis. Estas descripciones

adicionales no se encuentran explícitas en la representación de Gamma *et al.* (1994). Adicionalmente, en la Tabla 5-1 se resumen los elementos principales del patrón de diseño *Abstract Factory* definidos en la literatura que están presentes en la representación en DC y la representación en EP. Asimismo, en la Tabla 5-2 se resumen las colaboraciones entre los elementos principales del patrón de diseño que están presentes en ambas representaciones.

Tabla 5-1: Resumen de los elementos principales del patrón de diseño *Abstract Factory* (Gamma *et al*., 1994).

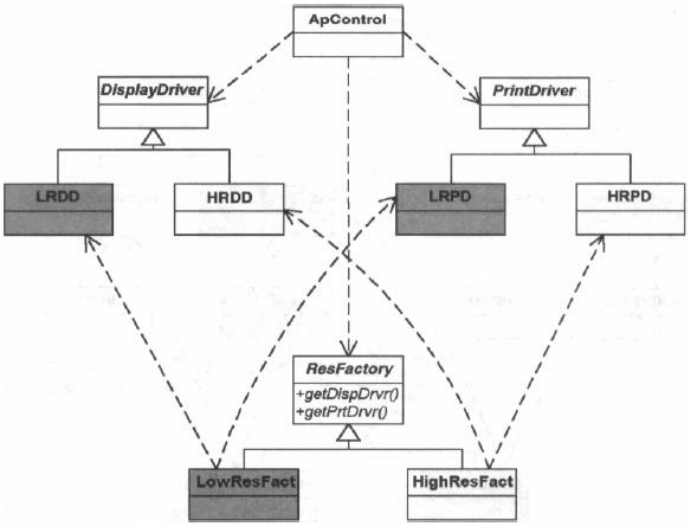
Elementos principales	Representación DC	Representación EP
Abstract Factory	x	x
Concrete Factory	x	x
Abstract Product	x	x
Concrete Product o Product	x	x
Client	x	x

Tabla 5-2: Resumen de las colaboraciones principales del patrón de diseño *Abstract Factory* (Gamma *et al*., 1994).

Colaboraciones	Representación DC	Representación EP
Cada *Concrete Factory* crea un producto específico.	x	x
El concepto *Client* usa diferentes *Concrete Factory* para crear diferentes *Products*	x	x
El *Abstract Product* aplaza la creación de objetos hasta la implementación en sus clases hijas (*Concrete Products*)	x	x

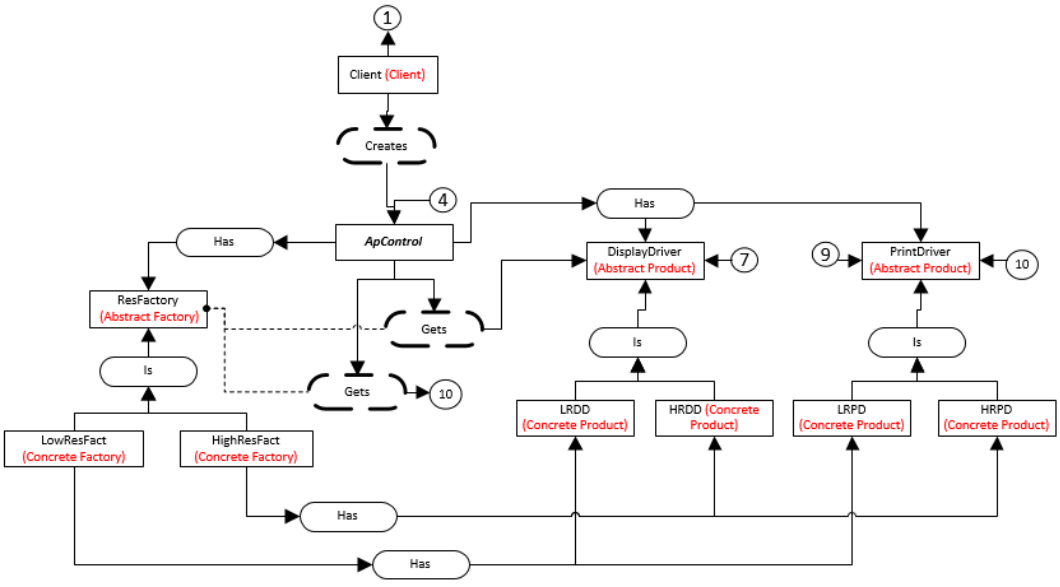
Por otro lado, en la Figura 5-3 se presenta un ejemplo de aplicación en la notación de diagrama de clases. Adicionalmente, en la Figura 5-4 se representa el mismo ejemplo en la arquitectura propuesta para los esquemas preconceptuales.

Figura 5-3: Ejemplo del patrón *Abstract Factory* en diagrama de clases.



Fuente: Tomado de (Shalloway y Trott, 2004).

Figura 5-4: Ejemplo del patrón *Abstract Factory* en esquemas preconceptuales.



Fuente: Elaboración propia.

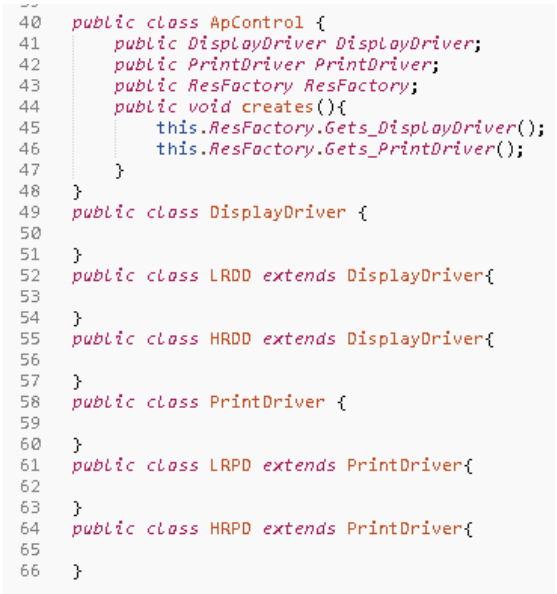
Finalmente, se presenta el código en Java generado a partir de esta representación con base en las reglas de obtención de código presentadas en Zapata y Arango (2007), Zapata (2012) y en la Sección 4.1 de esta Tesis en la Figura 5-5 y la Figura 5-6.

Figura 5-5: Código generado a partir de la representación en EP (1/2).

```
public class Client {
    public static void main(String[] args) {
        ApControl ApControl = new ApControl();
        ApControl.ResFactory = new LowResFact();
        ApControl.creates();
    }
}
public abstract class ResFactory {
    public abstract DisplayDriver Gets_DisplayDriver();
    public abstract PrintDriver Gets_PrintDriver();
}
public class LowResFact extends ResFactory{
    HRDD LRDD = new HRDD();
    HRPD LRPD = new HRPD();
    @Override
    public DisplayDriver Gets_DisplayDriver() {
        return LRDD;
    }
    @Override
    public PrintDriver Gets_PrintDriver() {
        return LRPD;
    }
}
public class HighResFact extends ResFactory{
    HRDD HRDD = new HRDD();
    HRPD HRPD = new HRPD();

    @Override
    public DisplayDriver Gets_DisplayDriver() {
        return HRDD;
    }

    @Override
    public PrintDriver Gets_PrintDriver() {
        return HRPD;
    }
}
```

Fuente: Elaboración propia.

Figura 5-6: Código generado a partir de la representación en EP (2/2).



```
public class ApControl {
    public DisplayDriver DisplayDriver;
    public PrintDriver PrintDriver;
    public ResFactory ResFactory;
    public void creates(){
        this.ResFactory.Gets_DisplayDriver();
        this.ResFactory.Gets_PrintDriver();
    }
}
public class DisplayDriver {

}
public class LRDD extends DisplayDriver{

}
public class HRDD extends DisplayDriver{

}
public class PrintDriver {

}
public class LRPD extends PrintDriver{

}
public class HRPD extends PrintDriver{

}
```

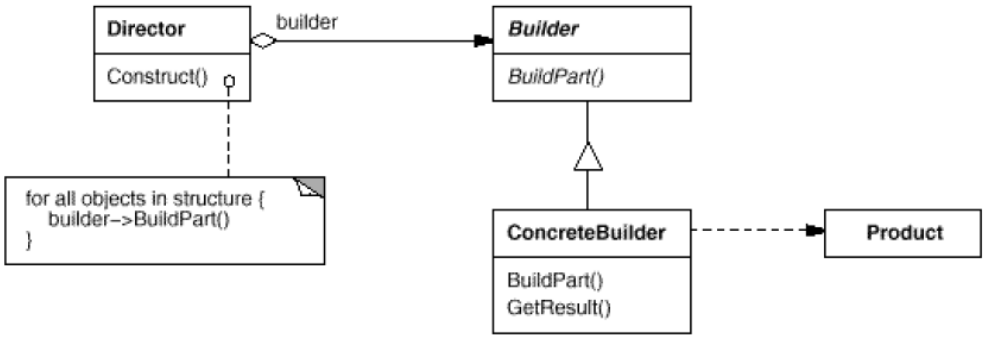
Fuente: Elaboración propia.

En resumen, se puede observar que ambas representaciones contienen los mismos elementos y relaciones principales que determinan el comportamiento del patrón de diseño. Además, la representación en esquemas preconceptuales, aporta un contexto dinámico, al representar las interacciones en el mismo diagrama.

5.2 Validación de la representación del patrón de diseño *Builder*

En esta Sección se compara la representación clásica del PD *Builder* a partir de diagrama de clases, con la representación en EP presentada en esta Tesis. En la Figura 5-7 se presenta la representación basada en diagrama de clases. En esta representación se pueden observar, de una manera general, los conceptos relevantes del patrón.

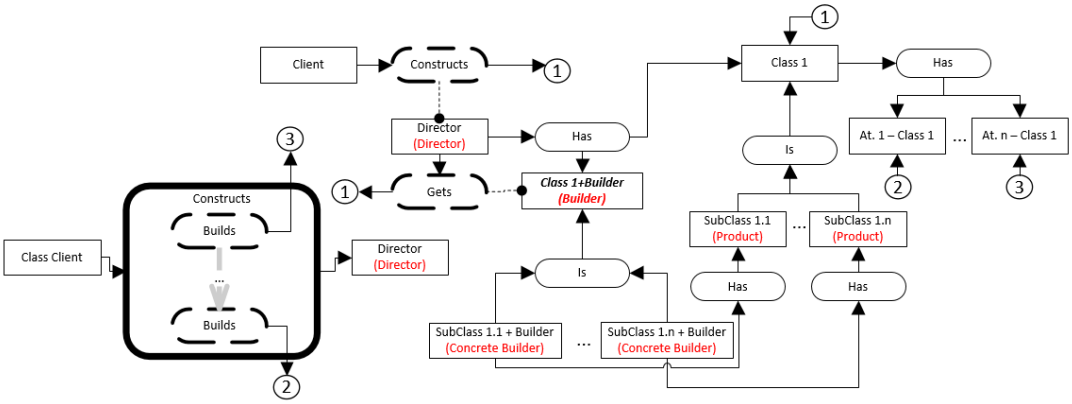
Figura 5-7: Representación del PD *Builder* en DC.



Fuente: Tomada de Gamma *et al.* (1994).

Asimismo, en la Figura 5-8 se presenta la representación en esquemas preconceptuales. En esta Figura se presenta nuevamente sólo una fracción de la representación propuesta en la Sección 4.2.3, ya que con esta fracción de la representación se contiene a la representación genérica que definen Gamma *et al.* (1994). Los demás elementos propuestos especifican, de una manera más detallada, las interacciones entre los participantes del patrón y no intervienen en su estructura.

Figura 5-8: Representación del PD *Builder* en EP.



Fuente: Elaboración propia.

En la Tabla 5-3 se resumen los conceptos principales que existen en la representación basada en DC y la representación en EP.

Tabla 5-3: Resumen de los elementos principales del patrón de diseño *Builder* (Gamma *et al*., 1994).

Elementos principales	Representación DC	Representación EP
Director	x	x
Builder	x	x
Concrete Builder	x	x
Product	x	x

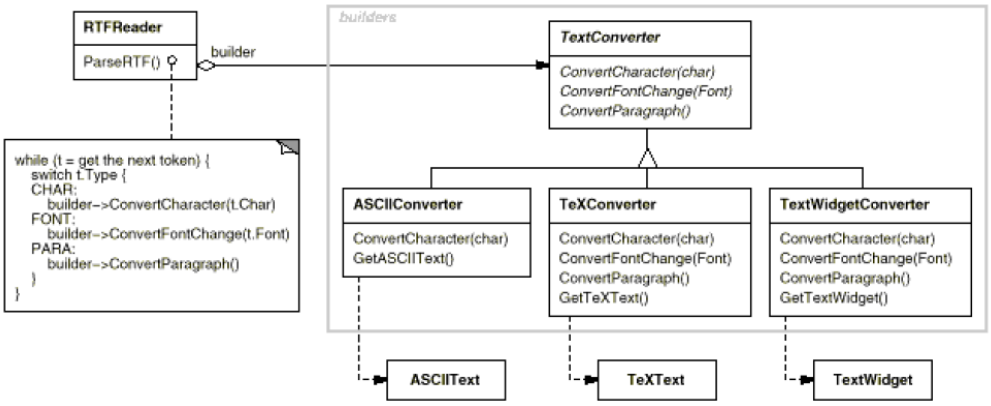
Además, en la Tabla 5-4 se resumen las colaboraciones existentes en este patrón de diseño y que se encuentran representadas en los DC y los EP.

Tabla 5-4: Resumen de las colaboraciones principales del patrón de diseño *Builder* (Gamma *et al*., 1994).

Colaboraciones	Representación DC	Representación EP
Un objeto cliente es el encargado de crear al *Director*	x	x
El *Director* gestiona las peticiones del cliente y se las pasa al *Builder*	x	x
El *Builder* devuelve el objeto construido	x	x
El *Abstract Product* aplaza la creación de objetos hasta la implementación en sus clases hijas (*Concrete Products*)	x	x

En la Figura 5-9 se presenta un ejemplo del patrón *Builder* tomado de Gamma *et al.* (1994), representado por medio de diagrama de clases.

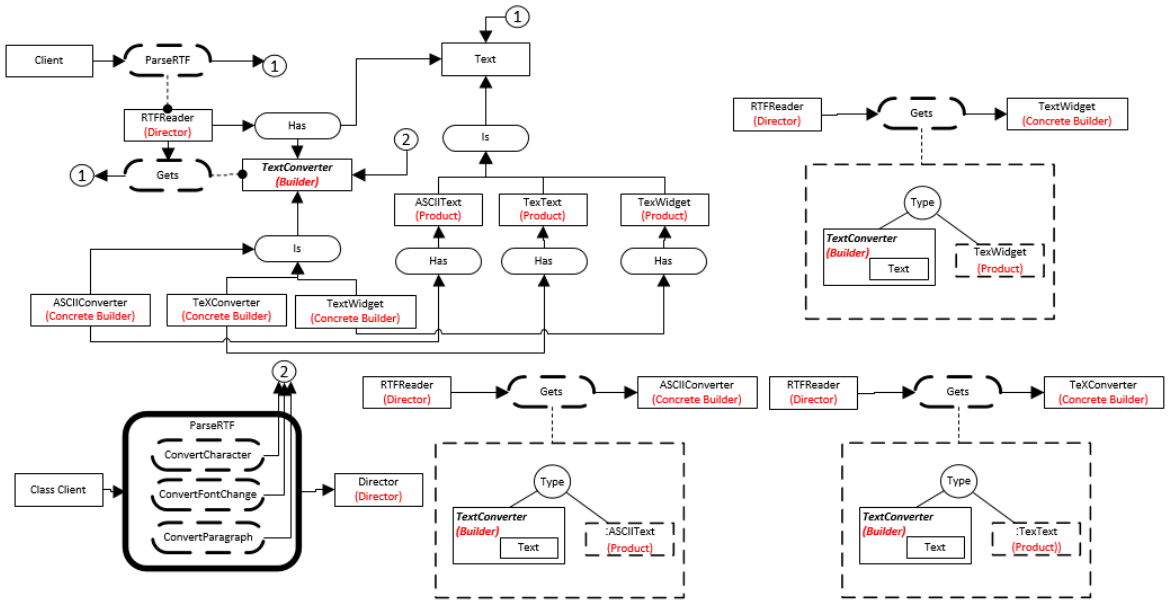
Figura 5-9: Ejemplo del PD *Builder* en DC.



Fuente: tomada de Gamma *et al.* (1994).

En la Figura 5-10 se presenta el mismo ejemplo representado en esquemas preconceptuales. En esta representación se muestra, adicionalmente, la especificación de sus relaciones dinámicas (métodos) para explicar el ejemplo de una manera más detallada.

Figura 5-10: Ejemplo del PD *Builder* en EP.



Fuente: Elaboración propia.

En la Figura 5-11 y la Figura 5-12 se presenta el código obtenido a partir de este ejemplo.

Figura 5-11: Código en Java del ejemplo de implementación del patrón de diseño *Builder* (1/2)

```
118 public class TexWidget extends TextConverter{
119
120     TextWidget TextWidget = new TextWidget();
121     @Override
122     public Text gets() {
123         return TextWidget;
124     }
125
126     @Override
127     public void ConvertCharacter() {
128         //Specification...
129     }
130
131     @Override
132     public void ConvertFontChange() {
133         //Specification...
134     }
135
136     @Override
137     public void ConvertParagraph() {
138         //Specification...
139     }
140
141 }
142 public class TextWidget extends Text{
143
144 }
145 public class ASCIIText extends Text{
146
147 }
148 public class TexText extends Text{
149
150 }
151 public class Text {
152     public Text Gets()
153     {
154         return this;
155     }
156 }
```

Fuente: Elaboración propia.

Finalmente, se demuestra que en la representación a partir de EP se incluyen todos los elementos necesarios para implementar este patrón de diseño. Además, se aporta información adicional desde la misma representación para detallar las relaciones dinámicas existentes en el sistema.

Figura 5-12: Código en Java del ejemplo de implementación del patrón de diseño *Builder* (2/2)

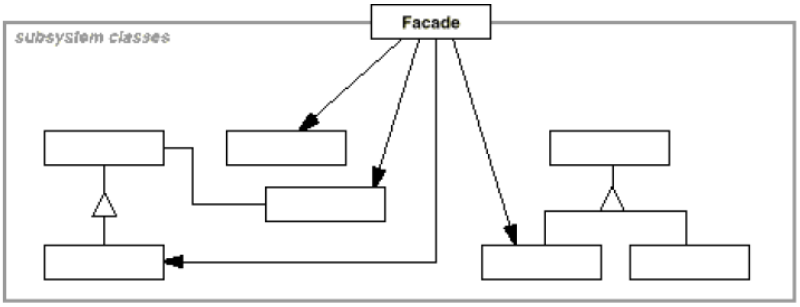
```
70  public class Client {
71
72      public static void main(String[] args) {
73          RTFReader RTFReader = new RTFReader();
74          RTFReader.ParseRTF();
75
76      }
77  }
78  public class RTFReader {
79      TextConverter TextConverter;
80      Text Text;
81      public void ParseRTF()
82      {
83          TextConverter.ConvertCharacter();
84          TextConverter.ConvertFontChange();
85          TextConverter.ConvertParagraph();
86      }
87  }
88  public abstract class TextConverter{
89      public abstract Text gets();
90      public abstract void ConvertCharacter();
91      public abstract void ConvertFontChange();
92      public abstract void ConvertParagraph();
93  }
94  public class ASCIIConverter extends TextConverter{
95
96      ASCIIText ASCIIText = new ASCIIText();
97      @Override
98      public Text gets() {
99          return ASCIIText;
100     }
101
102     @Override
103     public void ConvertCharacter() {
104         //Specification
105     }
106
107     @Override
108     public void ConvertFontChange() {
109         //Specification
110     }
111
112     @Override
113     public void ConvertParagraph() {
114         //Specification
115     }
116 }
```

Fuente: Elaboración propia.

5.3 Validación de la representación del patrón de diseño *Facade*

En esta Sección se comparan las representaciones en esquemas preconceptuales y diagrama de clases para el patrón de diseño *Facade*. En la Figura 5-13 se representa el patrón de diseño *Facade* con la notación de diagrama de clases de UML. Esta representación denota el encapsulamiento de un subsistema por medio de un solo elemento (“*Facade*”).

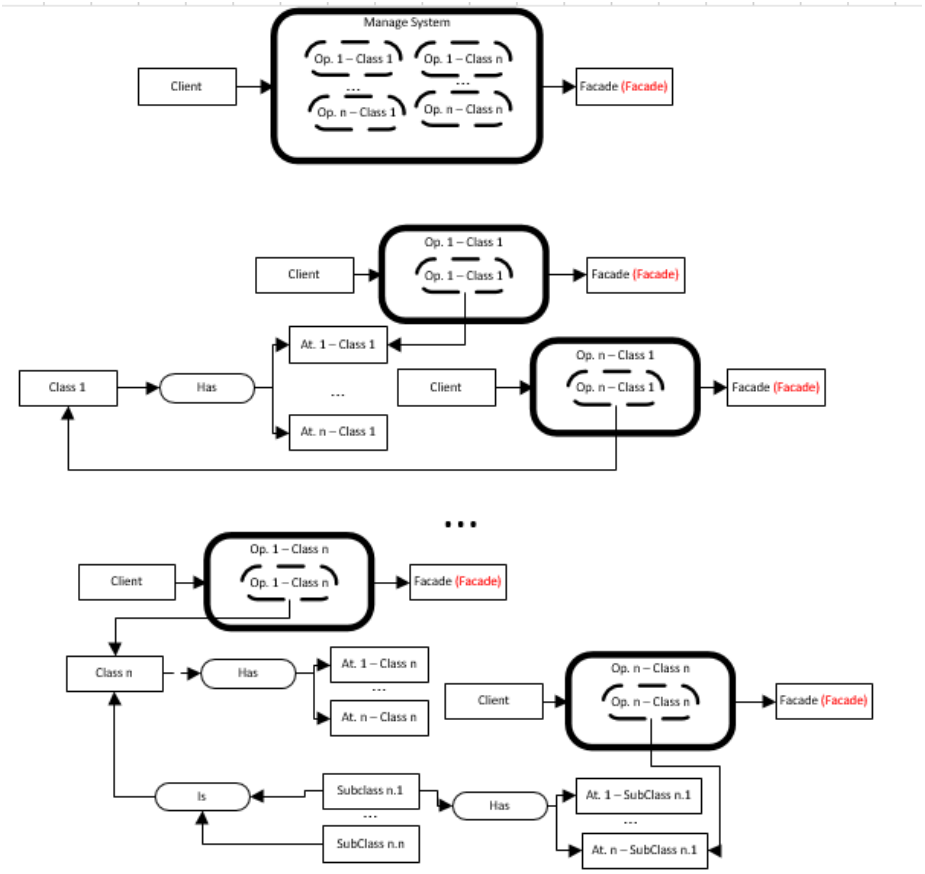
Figura 5-13: Representación en DC del patrón de diseño *Facade*.



Fuente: Tomada de Gamma *et al.* (1994).

Por otra parte, en la Figura 5-14 se presenta este patrón según lo propuesto en esta Tesis. Esta representación es un poco más específica y muestra, en el caso de tener relaciones dinámicas, cómo se comporta el sistema para atenderlas.

Figura 5-14: Representación en EP del patrón de diseño *Facade.*



Fuente: Elaboración propia.

En la Tabla 5-5 se resumen los elementos principales del patrón de diseño y su existencia en cada una de las representaciones.

Tabla 5-5: Resumen de los elementos principales del patrón de diseño *Facade* (Gamma *et al.,* 1994).

Elementos principales	Representación DC	Representación EP
Facade	x	x

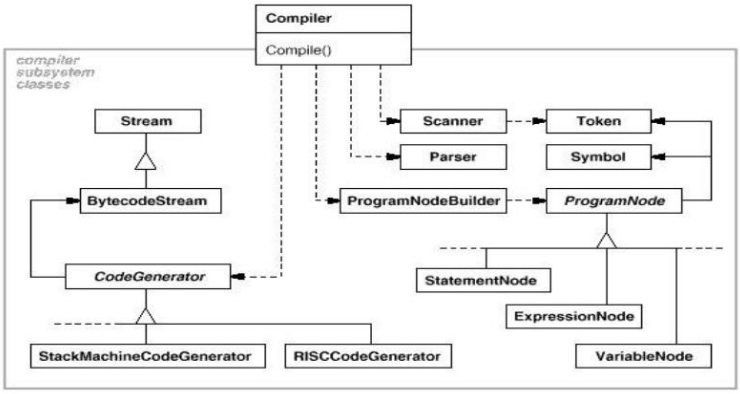
Además, en la Tabla 5-6 se resumen las colaboraciones existentes en este patrón de diseño y que se encuentran representadas en los DC y los EP.

Tabla 5-6: Resumen de las colaboraciones principales del patrón de diseño *Facade* (Gamma *et al*., 1994).

Colaboraciones	Representación DC	Representación EP
El elemento *Facade* conoce todos los elementos del sistema necesarios para efectuar las operaciones	x	x
El elemento *Facade* traduce las peticiones del cliente en llamadas al resto del sistema	x	x

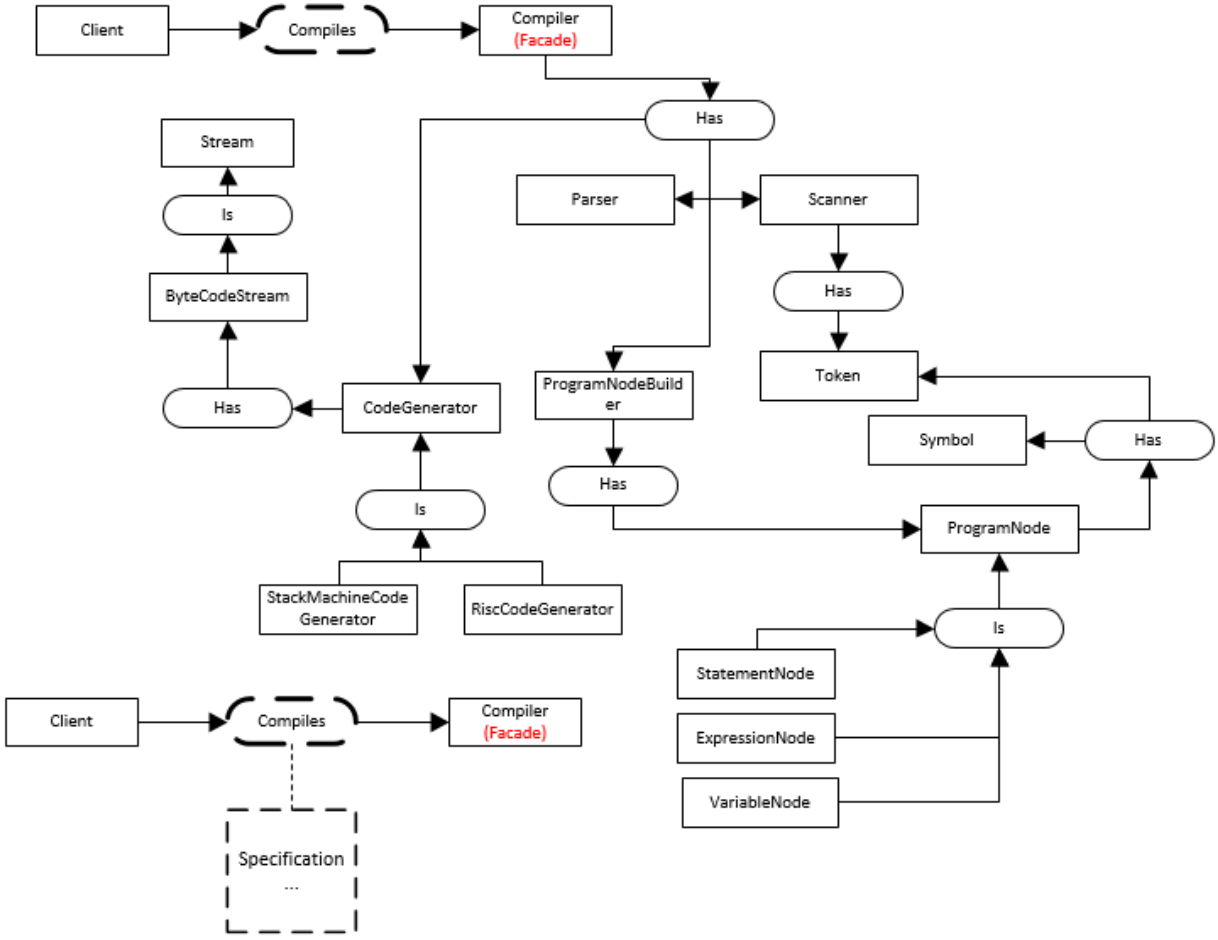
En la Figura 5-15 se presenta un ejemplo del patrón de diseño *Facade* aplicado en diagrama de clases y en la Figura 5-16 se ejemplifica el mismo patrón de diseño en esquemas preconceptuales. En estas representaciones se exhibe la participación del elemento principal *Facade* como centro del sistema que acopla.

Figura 5-15: Ejemplo del patrón de diseño *Facade* en DC.



Fuente: Tomada de Gamma *et al.* (1994).

Figura 5-16: Ejemplo del patrón de diseño *Facade* en esquemas preconceptuales.



Fuente: Elaboración propia.

Finalmente, en la Figura 5-17 se muestra el código a partir de la representación del ejemplo en esquemas preconceptuales. En esta validación, se puede notar que la especificación en esquemas preconceptuales atiende a todos los aspectos del patrón de diseño *Facade* y, además, ejemplifica la manera de extender el sistema y de especificar las relaciones dinámicas del dominio.

Figura 5-17: Código Java obtenido a partir de la representación del ejemplo en esquemas preconceptuales

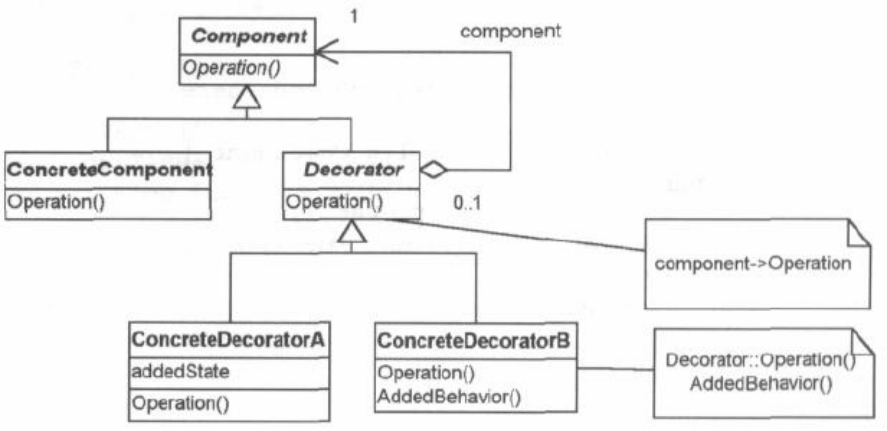
```
public class Client {

    public static void main(String[] args) {
        Compiler Compiler = new Compiler();
        Compiler.compiles();
    }
}
public class Compiler {
    CodeGenerator CodeGenerator = new CodeGenerator();
    ProgramNodeBuilder ProgramNodeBuilder = new ProgramNodeBuilder();
    Parser Parser = new Parser();
    Scanner Scanner = new Scanner();
    public void compiles()
    {
        //Specification
    }
}
public class Stream {
}
public class ByteCodeStream extends Stream{
}
public class CodeGenerator {
    ByteCodeStream ByteCodeStream = new ByteCodeStream();
}
public class StrackMachineCodeGenerator extends CodeGenerator{
}
public class RISCCodeGenerator extends CodeGenerator{

}
public class Scanner {
    Token Token = new Token();
}
public class Parser {
}
public class ProgramNodeBuilder {
    ProgramNode ProgramNode = new ProgramNode();
}
public class Token {
}
public class Symbol {
}
public class ProgramNode {
    Symbol Symbol = new Symbol();
    Token Token = new Token();
}
public class StatementNode extends ProgramNode{

}

```

Fuente: Elaboración propia.

5.4 Validación de la representación del patrón de diseño *Decorator*

En esta Sección se realiza la validación de la representación en esquemas preconceptuales del patrón de diseño *Decorator.* Esta validación se hace mediante la comparación de la arquitectura de la representación en EP con el de la representación estándar en DC de la literatura. Inicialmente, se muestra en la Figura 5-18 la representación del patrón de diseño *Decorator* en diagrama de clases. En esta representación se agregan unas notas que definen el comportamiento de algunas operaciones.

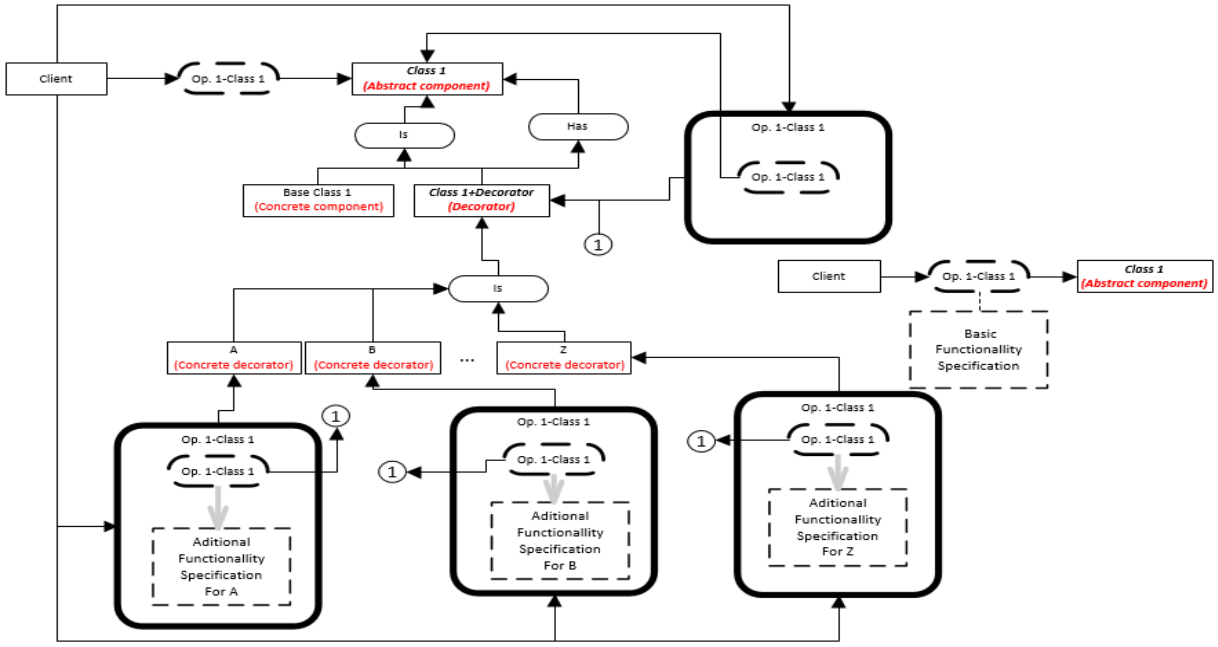
Figura 5-18: Representación del patrón de diseño *Decorator* a partir de DC.



Fuente: tomado de (Shalloway y Trott, 2004).

Por otro lado, en la Figura 5-19 se representa este patrón de diseño en EP. Esta representación no sólo muestra los componentes y relaciones principales del patrón, sino también la especificación de las funciones para ayudar al entendimiento de la interacción de estos elementos. Además, se emplea una notación que muestra cómo extender el patrón de diseño.

Figura 5-19: Representación del patrón de diseño *Decorator* a partir de EP.



Fuente: Elaboración propia.

En la Tabla 5-7 se resumen los elementos principales del patrón de diseño *Decorator* presentes en las representaciones estándar a partir de DC y la representación propuesta en esta Tesis en EP.

Tabla 5-7: Resumen de los elementos principales del patrón de diseño *Decorator* (Gamma *et al.*, 1994).

Elementos principales	Representación DC	Representación EP
Component	x	x
ConcreteComponent	x	x
Decorator	x	x
ConcreteDecorator	x	x

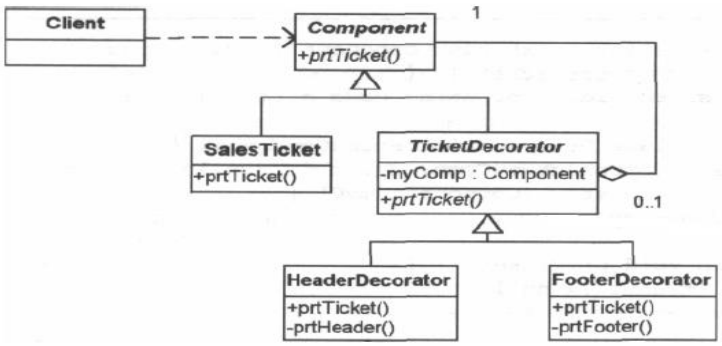
Asimismo, en la Tabla 5-8 se resumen las colaboraciones que definen el comportamiento específico del patrón de diseño y se define si están presentes o no en las representaciones mostradas en esta Sección.

Tabla 5-8: Resumen de las colaboraciones principales del patrón de diseño Decorator (Gamma *et al.*, 1994).

Colaboraciones	Representación DC	Representación EP
El elemento *Decorator* envía solicitudes al elemento *Component,* quien decide si agrega o no funcionalidad adicional al sistema	x	x

A continuación, se presenta un ejemplo del patrón de diseño *Decorator* en diagrama de clases en la Figura 5-20.

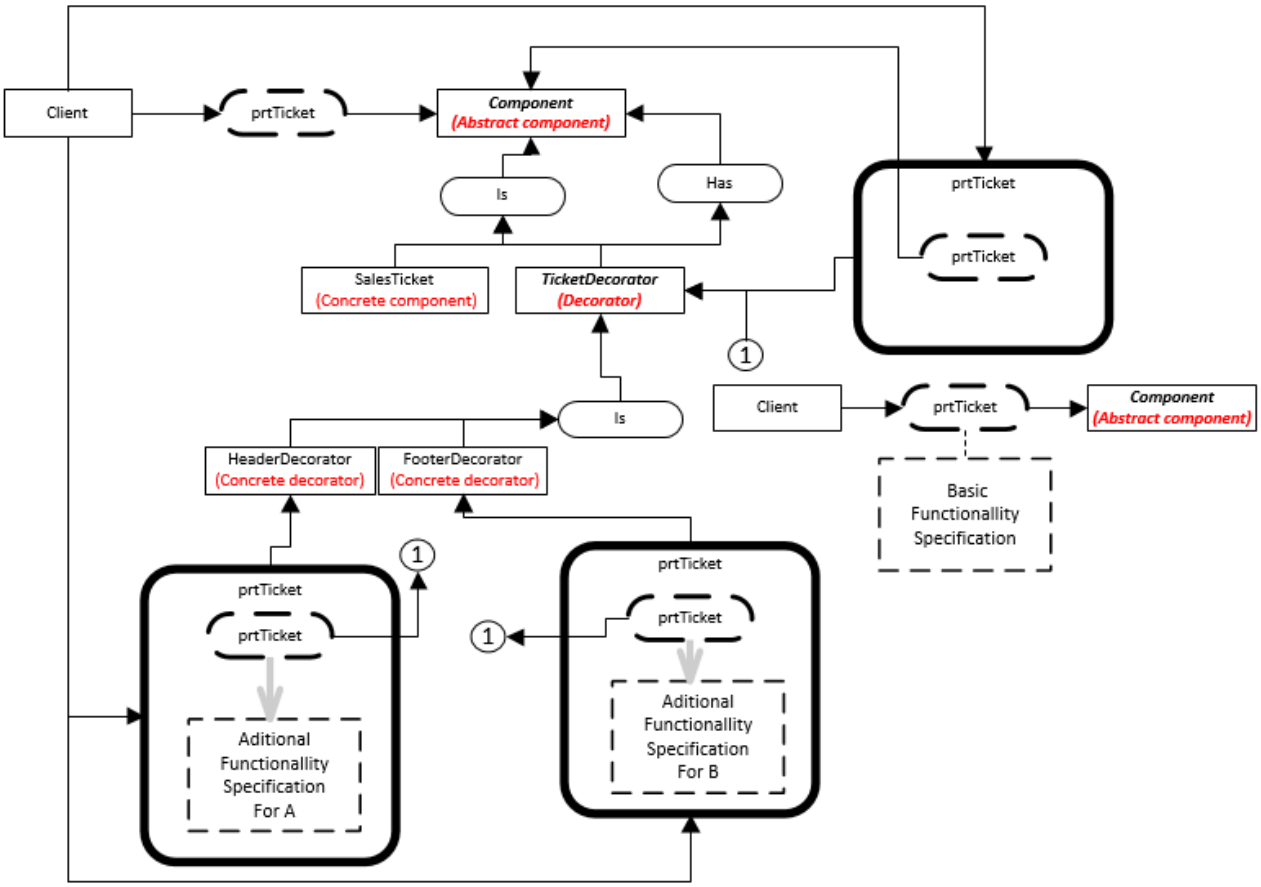
Figura 5-20: Ejemplo del patrón de diseño *Decorator* en DC.



Fuente: Tomado de (Shalloway y Trott, 2004).

Adicionalmente, se presenta el mismo ejemplo representado en EP usando la estructura propuesta en esta Tesis para implementar el patrón *Decorator* en la Figura 5-21.

Figura 5-21: Ejemplo del patrón de diseño *Decorator* en EP.



Fuente: Elaboración propia.

En esta representación, se pueden observar las características estáticas y dinámicas de la arquitectura. Además de contener todos los elementos y colaboraciones principales del patrón, agrupa el comportamiento total en un solo diagrama. Finalmente, en la Figura 5-22 se muestra el código en Java extraído de la representación del ejemplo del patrón de diseño *Decorator.*

Figura 5-22: Código en Java obtenido a partir del ejemplo del patrón de diseño *Decorator.*

```java
public class Client {

    public static void main(String[] args) {
        Component Component = new SalesTicket();
        Component.prtTicket();
    }

}
public abstract class Component {
    public abstract void prtTicket();
}
public class SalesTicket extends Component{

    @Override
    public void prtTicket() {
        //Bascir Functionality Specification
    }
}
public class TicketDecorator extends Component{

    Component Component;
    @Override
    public void prtTicket() {
        Component.prtTicket();
    }
}
public class HeaderDecorator extends TicketDecorator{
    public void prtTicket()
    {
        super.prtTicket();
        //Additional funcionality Specification
    }
}
public class FooterDecorator extends TicketDecorator{
        public void prtTicket()
    {
        super.prtTicket();
        //Additional funcionality Specification
    }
}
```

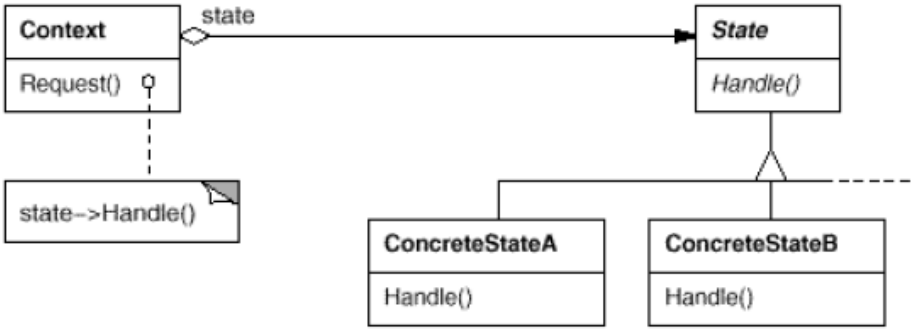
Fuente: Elaboración propia.

5.5 Validación de la representación del patrón de diseño *State*

En esta Sección se comparan las representaciones en diagrama de clases y esquemas preconceptuales del patrón de diseño *State* para validar que la representación presentada en esta Tesis soporte la implementación de dicho patrón.

En la Figura 5-23 se representa el patrón de diseño *State* por medio de diagrama de clases. Allí, se pueden observar los elementos y relaciones principales del patrón.

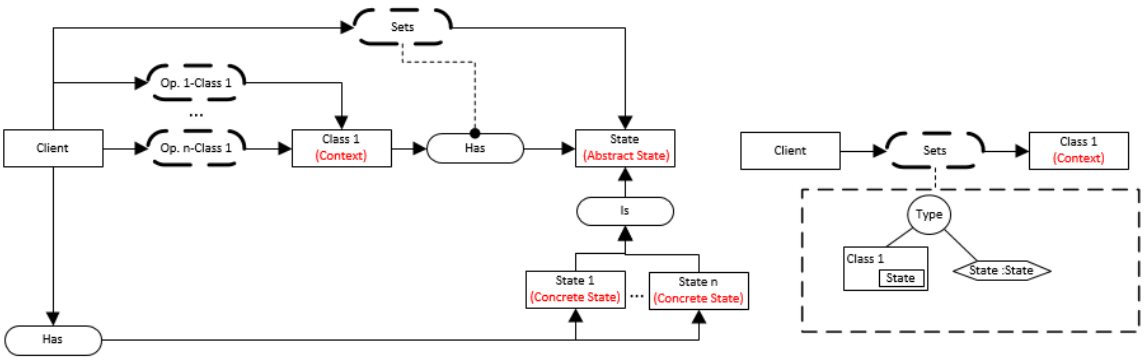
Figura 5-23: Representación del patrón de diseño *State* en DC.



Fuente: Tomada de Gamma *et al.* (1994).

Por otra parte, en la Figura 5-24 se presenta la caracterización del patrón *State* a partir de esquemas preconceptuales. En este fragmento de la representación están presentes todos los elementos necesarios para representar e implementar el patrón de diseño.

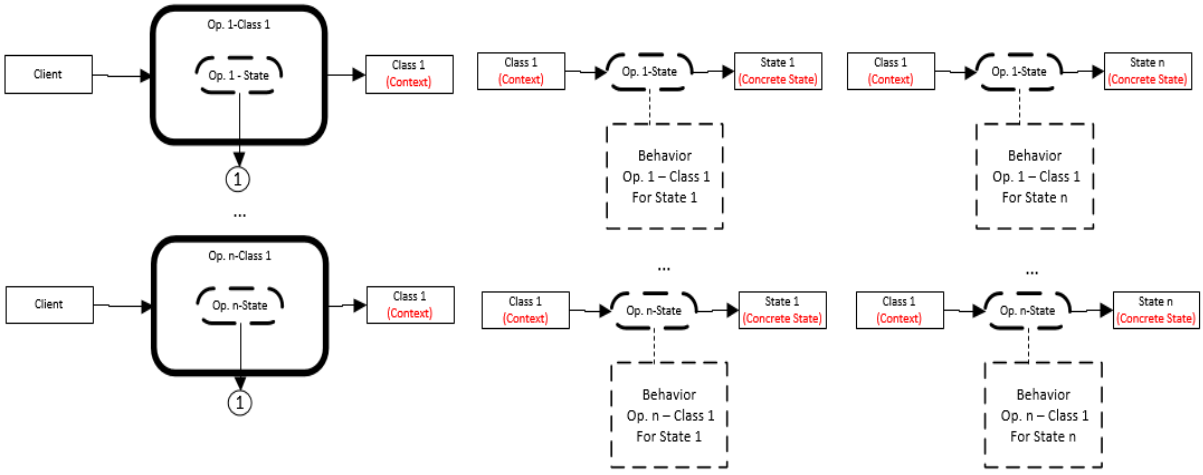
Figura 5-24: Representación del patrón de diseño *State* en EP (1/2).



Fuente: Elaboración propia.

Adicionalmente, se extiende la representación para especificar detalladamente los comportamientos de las relaciones dinámicas de la arquitectura y para observar más fácilmente la comunicación entre los diversos componentes del patrón de diseño. En la Figura 5-25 se especifican todas las relaciones dinámicas de la representación.

Figura 5-25: Representación del patrón de diseño *State* en EP (2/2).



Fuente: Elaboración propia.

En la Tabla 5-9 se resumen los elementos presentes en las representaciones del patrón de diseño *State* que se describieron anteriormente. Estos elementos hacen parte de la arquitectura del patrón de diseño.

Tabla 5-9: Resumen de los elementos principales del patrón de diseño *State* (Gamma *et al.,* 1994).

Elementos principales	Representación DC	Representación EP
Context	X	X
State	X	X
ConcreteState	X	X

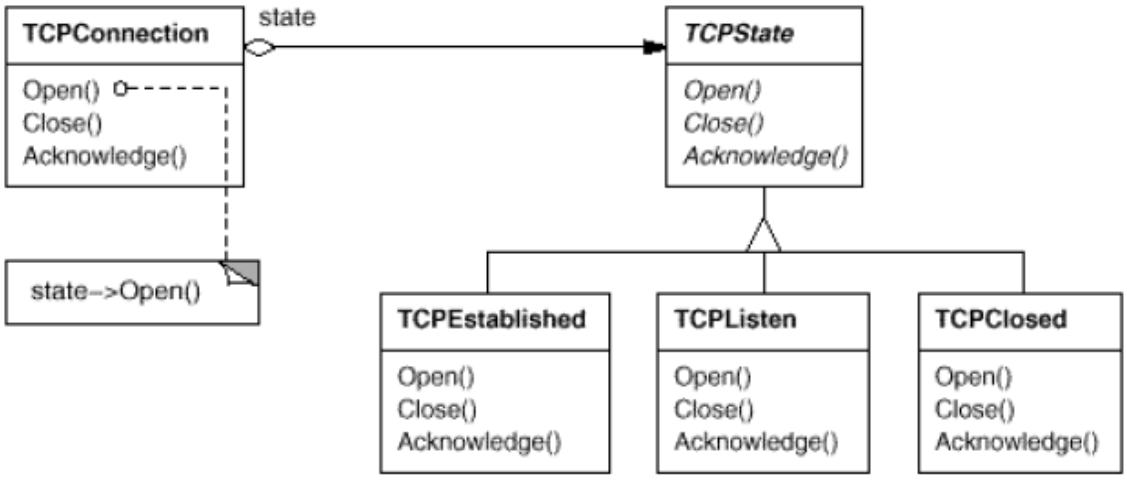
Asimismo, en la Tabla 5-10 se resumen las colaboraciones que definen el comportamiento específico del patrón de diseño y se muestra si están en la representación de diagrama de clases o de esquemas preconceptuales.

Tabla 5-10: Resumen de las colaboraciones principales del patrón de diseño *State* (Gamma *et al.,* 1994).

Colaboraciones	Representación DC	Representación EP
El elemento *Context* delega las peticiones al elemento ConcreteState adecuado.	x	x
El elemento *Context* representa la interfaz al usuario para interactuar con el sistema	x	x
El elemento *Context* posee una referencia a un objeto tipo *State*	x	x

Por otro lado, en la Figura 5-26 se presenta un ejemplo del patrón de diseño *State* implementado. En esta representación se pueden observar todos los participantes y las relaciones básicas entre ellos.

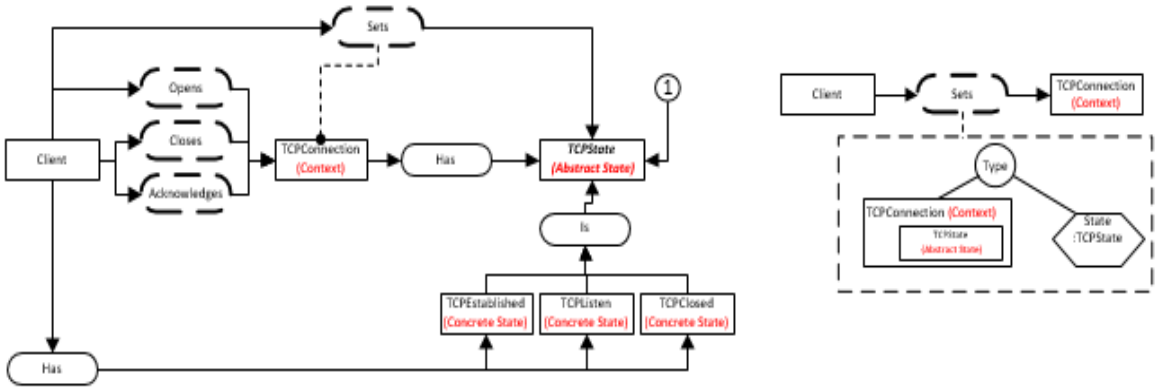
Figura 5-26: Ejemplo del patrón de diseño *State* en DC.



Fuente: Tomada de Gamma *et al.* (1994).

Asimismo, se expresa en la Figura 5-27 un fragmento de la representación de este ejemplo donde se pueden observar los elementos principales del patrón y una pate de la especificación de la interacción que hay entre ellos.

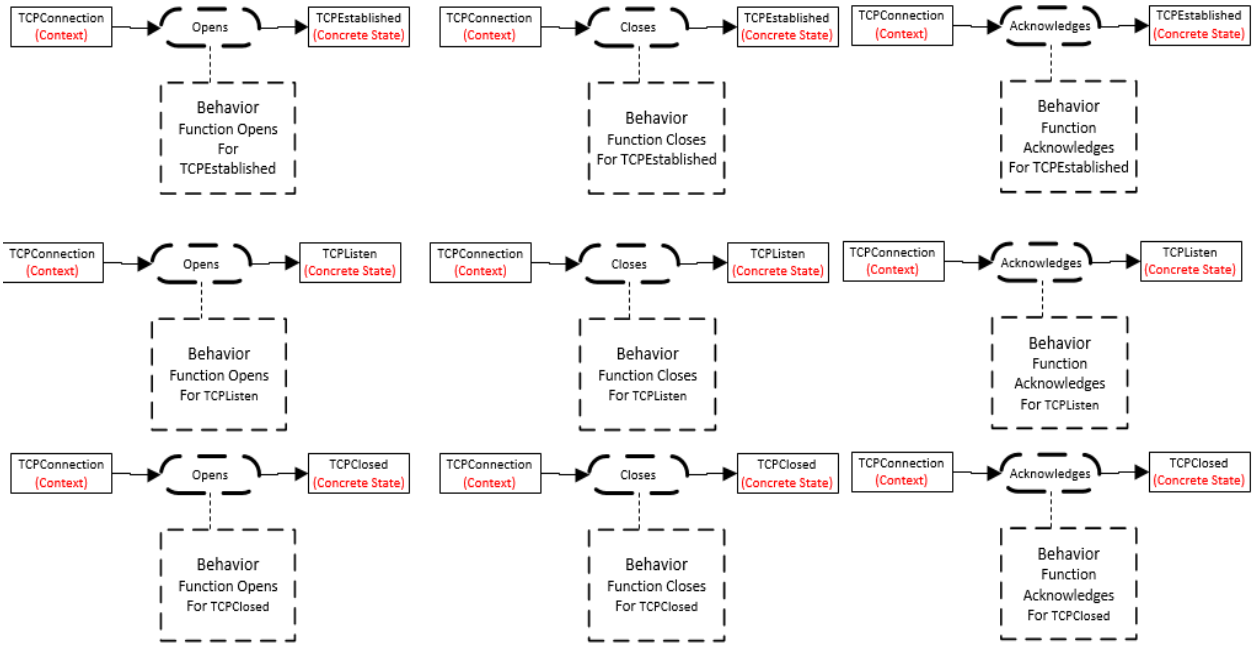
Figura 5-27: Ejemplo del patrón de diseño *State* en EP (1/3).



Fuente: Elaboración propia.

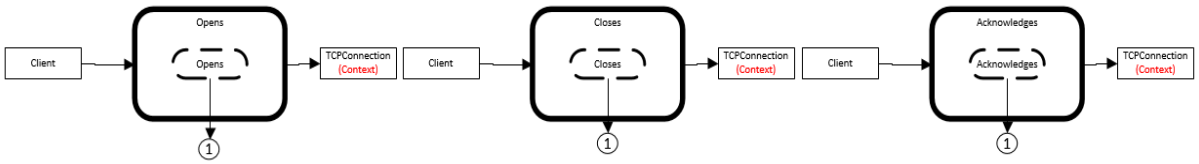
En la Figura 5-28 y la Figura 5-29 se representan, además, las especificaciones de todas las interacciones del sistema. Esto permite entender la manera en que se delegan las tareas en este patrón.

Figura 5-28: Ejemplo del patrón de diseño *State* en EP (2/3).



Fuente: Elaboración propia.

Figura 5-29: Ejemplo del patrón de diseño *State* en EP (3/3).



Fuente: Elaboración propia.

Finalmente, se presenta en la Figura 5-30 y la Figura 5-31 el código Java que se obtiene a partir de estas representaciones. En síntesis, es posible sólo con una fracción de la representación del esquema preconceptual, tener la implementación completa del patrón de diseño *State.* Además, al tener parte de su estructura dinámica, facilita el entendimiento y la reutilización de este componente.

Figura 5-30: Código en Java obtenido a partir del ejemplo del patrón *State* (1/2).

```
public class Client {

    public static void main(String[] args) {
        TCPConnection TCPConnection = new TCPConnection();
        TCPEstablished TCPEstablished = new TCPEstablished();
        TCPListen TCPListen = new TCPListen();
        TCPClosed TCPClosed = new TCPClosed();
    }

}
public class TCPConnection {
    TCPState TCPState;
    public void Opens()
    {
        TCPState.Opens();
    }
    public void Closes()
    {
        TCPState.Closes();
    }
    public void Acknowledges()
    {
        TCPState.AcknowLedges();
    }
    public void sets(TCPState State)
    {
        this.TCPState = State;
    }
}
public class TCPEstablished extends TCPState{

    @Override
    public void Opens() {
        //Behavior Function Opens For TCPEstablished
    }

    @Override
    public void Closes() {
        //Behavior Function Closes For TCPEstablished
    }

    @Override
    public void Acknowledges() {
        //Behavior Function AcknowLedges For TCPEstablished
    }

}
```

Fuente: Elaboración propia.

Figura 5-31: Código en Java obtenido a partir del ejemplo del patrón *State* (2/2).

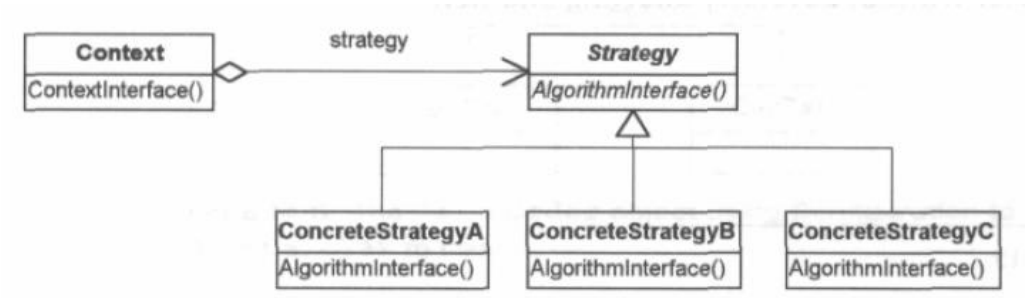
```
50  public class TCPListen extends TCPState{
51
52      @Override
53      public void Opens() {
54          //Behavior Function Opens For TCPListen
55      }
56
57      @Override
58      public void Closes() {
59          //Behavior Function Closes For TCPListen
60      }
61
62      @Override
63      public void Acknowledges() {
64          //Behavior Function Acknowledges For TCPListen
65      }
66
67  }
68  public class TCPClosed extends TCPState{
69
70      @Override
71      public void Opens() {
72          //Behavior Function Opens For TCPClosed
73      }
74
75      @Override
76      public void Closes() {
77          //Behavior Function Closes For TCPClosed
78      }
79
80      @Override
81      public void Acknowledges() {
82          //Behavior Function Acknowledges For TCPClosed
83      }
84
85  }
86  public abstract class TCPState {
87      public abstract void Opens();
88      public abstract void Closes();
89      public abstract void Acknowledges();
90  }
```

Fuente: Elaboración propia.

5.6 Validación de la representación del patrón de diseño *Strategy*

En esta Sección se compara la representación en esquemas preconceptuales con la representación estándar del patrón de diseño *Strategy.* En la Figura 5-32 se enseña la representación tradicional del patrón de diseño *Strategy* a partir de un diagrama de clases.

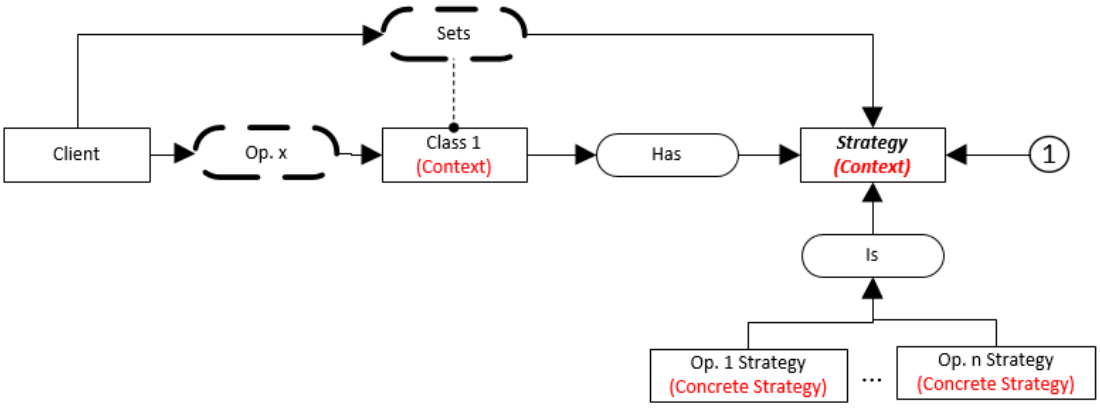
Figura 5-32: Representación del patrón de diseño *Strategy* en DC.



Fuente: Tomado de (Shalloway y Trott, 2004).

Por otro lado, en la Figura 5-33 se muestra una porción de la representación de este patrón de diseño por medio de esquemas preconceptuales. En este fragmento se pueden observar los elementos principales del patrón de diseño, así como sus relaciones.

Figura 5-33: Representación del patrón de diseño *Strategy* en EP.



Fuente: Elaboración propia.

En la Tabla 5-11 se resumen los elementos que componen el patrón de diseño *Strategy* y si existen o no en la representación basada en diagramas de clases o en esquemas preconceptuales.

Tabla 5-11: Resumen de los elementos principales del patrón de diseño *Strategy.*

Elementos principales	Representación DC	Representación EP
Strategy	x	x
ConcreteStrategy	x	x
Context	x	x

Por otro lado, en la Tabla 5-12 se sintetizan las colaboraciones que definen el patrón de diseño *Strategy.*

Tabla 5-12: Resumen de las colaboraciones principales del patrón de diseño *Strategy.*

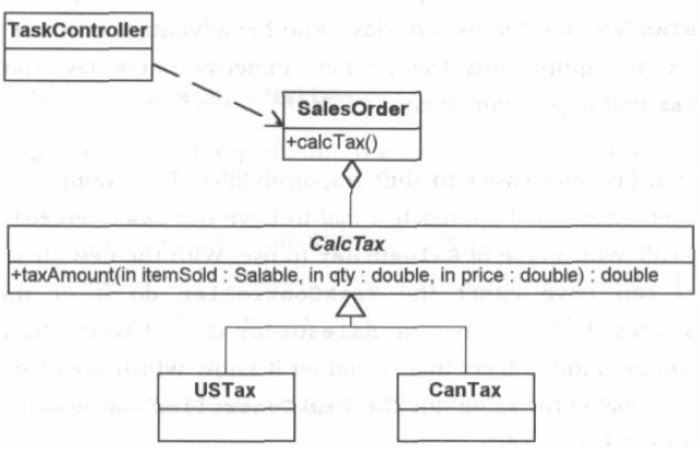
Colaboraciones	Representación DC	Representación EP
El elemento *Strategy* y el elemento *Context* interactúan para implementar el algoritmo escogido.	x	x

Tabla 5-12: (Continuación) Resumen de las colaboraciones principales del patrón de diseño *Strategy.*

Colaboraciones	Representación DC	Representación EP
La jerarquía de elementos *Strategy* representan la familia de algoritmos del sistema.	x	x
Los elementos *Strategy* eliminan condicionales de la especificación de las relaciones dinámicas.	x	x

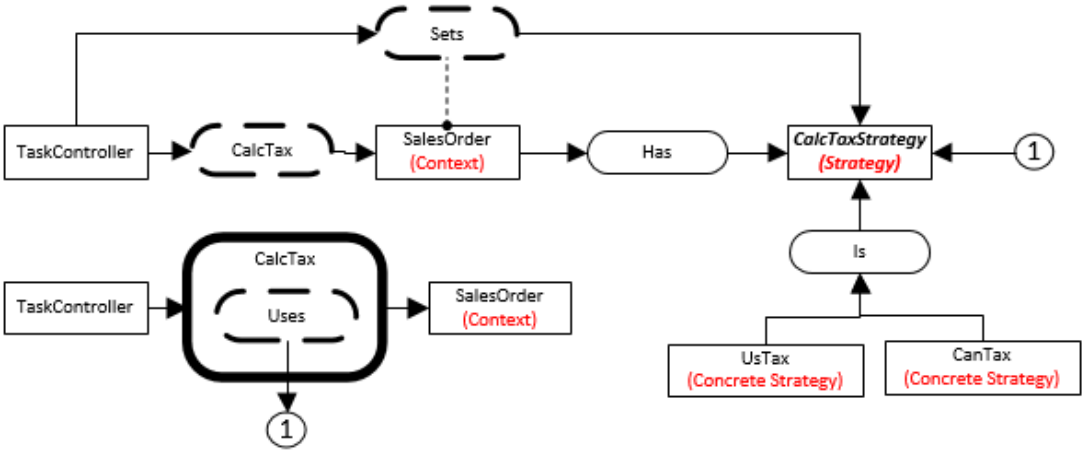
En la Figura 5-34 se expone un ejemplo del patrón de diseño *Strategy* implementado a partir de un diagrama de clases. En esta representación, aunque los nombres no sean explícitos, se puede ver que tienen la arquitectura correcta para implementar diferentes formas de resolver un mismo problema, en este caso el cálculo de impuestos (“CalcTax”). Asimismo, en la Figura 5-35 se representa el mismo ejemplo a partir de esquemas preconceptuales.

Figura 5-34: Ejemplo del patrón de diseño *Strategy* implementado en un DC.



Fuente: Tomado de (Shalloway y Trott, 2004).

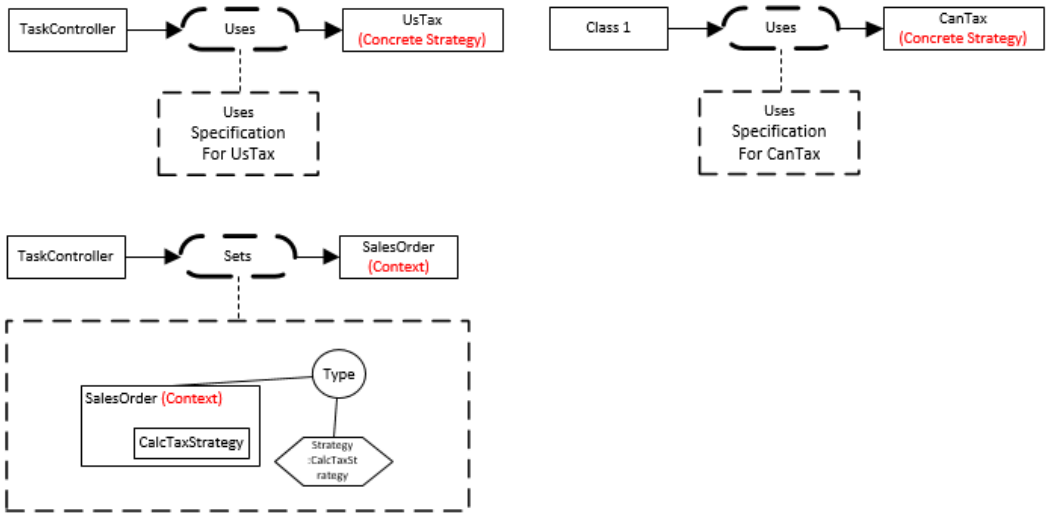
Figura 5-35: Ejemplo del patrón de diseño *Strategy* implementado en EP (1/2).



Fuente: Elaboración propia.

En la representación a partir de esquemas preconceptuales, se puede observar el comportamiento de las relaciones dinámicas del sistema. Además, se especifica cada uno de los posibles flujos que define cada estrategia en el patrón de diseño. Esto se evidencia mejor en la Figura 5-36.

Figura 5-36: Ejemplo del patrón de diseño *Strategy* implementado en EP (2/2).



Fuente: Elaboración propia.

Finalmente, en la Figura 5-37 se presenta el código en Java obtenido desde la representación del patrón de diseño *Strategy* presentado en esta Sección.

Figura 5-37: Código en Java obtenido a partir de las representaciones del PD *Strategy.*

```java
public class TaskController {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        SalesOrder SalesOrder = new SalesOrder();
    }

}
public class SalesOrder {
    CalcTaxStrategy CalcTaxStrategy;
    public void CalcTax()
    {
        CalcTaxStrategy.uses();
    }
    public void Sets(CalcTaxStrategy CalcTaxStrategy )
    {
        this.CalcTaxStrategy =CalcTaxStrategy;
    }
}
public abstract class CalcTaxStrategy {
    public abstract void uses();
}
public class UsTax extends CalcTaxStrategy{

    @Override
    public void uses() {
        //Uses Specification for UsTax
    }

}
public class CanTax extends CalcTaxStrategy{
    @Override
    public void uses() {
        //Uses Specification for CanTax
    }
}

```

Fuente: Elaboración propia.

En la representación estándar de este patrón de diseño, se omiten algunas dinámicas importantes del sistema como la elección de una estrategia. La representación propuesta en esta Tesis para el patrón de diseño *Strategy* incluye todos los elementos y colaboraciones necesarias para implementar el patrón, además de las especificaciones de las operaciones del sistema y de la asignación de elementos importantes como el elemento tipo *Strategy* del dominio del patrón.

6. Conclusiones y trabajo futuro

6.1 Conclusiones

En esta Tesis de Maestría se propusieron herramientas para cerrar la brecha entre la ingeniería de software y la comunidad científica que desarrolla software (Segal, 2008).

El uso de esquemas preconceptuales como método de comunicación y representación en contextos científicos facilita la comunicación entre ingenieros de software y científicos e incluso, entre científicos de disciplinas diferentes. Dada la cercanía de estos esquemas con el lenguaje natural, se facilita la comunicación y validación de los dominios complejos presentes en este contexto.

Los patrones de diseño aplicados a software científico constituyen una de las mejores aproximaciones a la programación orientada a objetos. Estas arquitecturas proporcionan una solución al impacto en la eficiencia del software científico, que surge cuando se adapta un software complejo a un paradigma objetual.

Las representaciones de los patrones de diseño que se encuentran en la literatura carecen de un modelo integrador que condense la representación completa de un patrón de diseño. Además, no existen aproximaciones para caracterizar, de una manera gráfica, los problemas que resuelven los patrones de diseño. En esta Tesis se propuso una representación integral del patrón de diseño, contemplando sus características estáticas y dinámicas. Además, se propuso una representación gráfica del problema genérico que resuelve cada patrón de diseño.

A partir de la representación del problema genérico del patrón de diseño, estos patrones se pueden usar en dominios predefinidos que se ajusten a la arquitectura propuesta en estas representaciones. Las especificaciones de los problemas de diseño, sirven para la extracción de reglas que permitan implementar los patrones de diseño automáticamente.

Por otro lado, teniendo en cuenta la facilidad de entendimiento que proporcionan los esquemas preconceptuales, las representaciones propuestas en este trabajo le permiten a un grupo más amplio de profesionales en distintas áreas el uso de los patrones de diseño en sus soluciones informáticas.

6.2 Trabajo futuro

A partir de esta Tesis de Maestría se identifican varias líneas de investigación por desarrollar:

- Automatizar la identificación de los problemas genéricos de los patrones de diseño en dominios predefinidos.
- Automatizar la implementación de los patrones de diseño en dominios predefinidos.
- Caracterizar y representar los otros patrones de diseño presentes en la literatura y extrapolarlos a otros dominios y contextos.
- Definir un metamodelo con los patrones de diseño para automatizar la selección de uno o varios patrones de diseño para problemas que no estén predefinidos en la literatura.
- Caracterizar y representar algunos patrones de análisis para definir nuevos patrones de diseño.

Bibliografía

Ackroyd, K., Kinder, S., Mant, G., Miller, M., Ramsdale, C., y Stephenson, P. (2008). Scientific Software Development at a Research Facility. *IEEE Software*, 8.

Ahmed, Z. (2015). Essential Design Modeling for Scientific Software Solutions Development. *Peerj*.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., y Angel, S. (1977). *A Pattern Language* (1st ed.). New York: Oxford University Press. Retrieved from http://library.uniteddiversity.coop/Ecological_Building/A_Pattern_Language.pdf

Aras, K., Cickovski, T., y Izaguirre, J. (2004). Empirical Evaluation of Design Patterns in Scientific Application. *Citeseer*. Retrieved from http://scholar.google.com/scholar?hl=enybtnG=Searchyq=intitle:Empirical+Evaluation+of+Design+Patterns+in+Scientific+Application#0

Barbieri, D., Cardellini, V., Filippone, S., y Rouson, D. (2012). Design Patterns for Scientific Computations on Sparse Matrices. *Lectures in Computer Science*. https://doi.org/10.1007/3-540-68339-9_34

Bashir, R. S., Lee, S. P., Khan, S. U. R., Chang, V., y Farid, S. (2016). UML models consistency management: Guidelines for software quality manager. *International Journal of Information Management*, *36*(6), 883–899. https://doi.org/10.1016/j.ijinfomgt.2016.05.024

Bass, L., Clements, P., y Kazman, R. (2003). *Software Architecture in Practice*. *Vasa* (Third, Vol. 1st). Boston: Pearson. https://doi.org/10.1024/0301-1526.32.1.54

Bonfè, M., Fantuzzi, C., y Secchi, C. (2013). Design patterns for model-based automation software design and implementation. *Control Engineering Practice*, *21*(11), 1608–1619. https://doi.org/10.1016/j.conengprac.2012.03.017

Carver, J., Kendall, R., Squires, S., y Post, D. (2007). Software Development Environments for Scientific and Engineering Software: A Series of Case Studies. In *29th International conference on Software Engineering (ICSE'07)* (p. 10). Mississipi: IEEE.

Charles, B. (2002). *Patterns in scientific software: An Introduction*. *Feature software patterns* (2nd ed.). Computing In Science y Engineering.

Cickovski, T., Matthey, T., y Izaguirre, J. (2004). Design Patterns for Generic Object-Oriented Scientific Software. Bergen.

Clune, R., Connor, J. J., Ochsendorf, J. A., y Kelliher, D. (2012). An object-oriented architecture for extensible structural design software. *Computers and Structures*, *100–101*, 1–17. https://doi.org/10.1016/j.compstruc.2012.02.002

Cuéllar, E., Muñoz, G. R., y Gómez, J. (2005). Aplicación de Patrones de Software en el Dominio de los Simuladores de Procesos Dinámicos. Aguascalientes.

Davis, C. (2012). Software engineering for computational science and engineering. *Computing in Science y Engineering*, *14*(2), 8–11. https://doi.org/10.1109/MCSE.2012.31

De Oliveira, K. S., y Soares, M. S. (2013). A systematic review on aspects in software architecture design. *Proceedings - International Conference of the Chilean Computer Science Society, SCCC*, 21–28. https://doi.org/10.1109/SCCC.2012.10

Decyk, V., y Gardner, H. (2006). Object-Oriented Design Patterns in Scientific Codes using Fortran95. *Elsevier Science2*, 4–6.

Decyk, V., y Gardner, H. (2007). Object-Oriented Design Patterns in Fortran. *Elsevier Science*, (July 2007).

Dijkstra, E. W. (1972). Selected Writings on Computing: A Personal Perspective. *Ewd*, *477*, 60–66. https://doi.org/10.1007/978-1-4612-5695-3

Dodig-crnkovic, G. (2002). Scientific Methods in Computer Science. *Computer (Long. Beach. Calif).*, 126–130. Retrieved from http://www.mrtc.mdh.se/~gdc/work/cs_method.pdf

Dong, J. (2002). UML extensions for design pattern compositions. *Journal of Object Technology*, *1*(5), 149–161. Retrieved from http://www.jot.fm/issues/issue_2002_11/article3.pdf

Dong, J. (2003). Representing the applications and compositions of design patterns in UML. *SAC '03: Proceedings of the 2003 ACM Symposium on Applied Computing*, 1092–1098 PU–ACM Press. Retrieved from http://dx.doi.org/10.1145/952532.952746

Dong, J., y Yang, S. (2003). Visualizing design patterns with a UML profile. *Proceedings - 2003 IEEE Symposium on Human Centric Computing Languages and Environments,*

HCC 2003, 123–125. https://doi.org/10.1109/HCC.2003.1260215

Dong, J., Yang, S., y Zhang, K. (2007). Visualizing design patterns in their applications and compositions. *IEEE Transactions on Software Engineering*, *33*(7), 433–453. https://doi.org/10.1109/TSE.2007.1012

Fowler, M. (2004). *UML Distilled: A brief guide to the Standard Object Modeling Language* (3rd ed.). Canada: Addison Wesley. Retrieved from http://www.agentgroup.unimore.it/~nicola/courses/IngegneriaDelSoftware/uml/books/UMLDistilled.pdf

Fowler, M., y Kendall, S. (1999). *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (2nd ed.). Canada: Addison Wesley. Retrieved from www.awl.com/cseng/

Gamma, E., Helm, R., Johnson, R., y Vissides, J. (1994). *Design Patterns: Elements of reussable oriented-objected software*. (Ke. Zhang, Ed.) (6th ed.). Sidney.

Gardner, H. (2004). *Design Patterns in Scientific Software*. *Design patterns in scientific software* (4th ed.). Berlin: Springer-Verlag.

Gardner, H., y Manduchi, G. (2007). Design Patterns for e-Science. *Elsevier Science*, 64–81.

Garlan, D., Shaw, M., Company, W. S. P., y Institute, C. M. U. S. E. (1994). An Introduction to Software Architecture. *Cmu/Sei-94-Tr-21*, (January).

Garlan, D., T. Monroe, R., y Wile, D. (2000). ACME: An architectural description of component-based systems. *Foundations of Component-Based Systems*, 47–68. Retrieved from http://www.cs.cmu.edu/afs/cs/project/able/ftp/acme-fcbs/acme-fcbs.pdf

Hannay, E., MacLeod, C., Singer, J., Langtangen, H. P., Pfahl, D., y Wilson, G. (2009). How Do Scientist Develop and Use Scientific Software? In *SECSE'09* (p. 8). Vancouver: IEEE Computer Society.

Hartmanis, J., y Leeuwen, J. Van. (2001). *Large-Scale Scientific Computing* (1st ed.). Sozopol: Springer.

Hasheminejad, S. M. H., y Jalili, S. (2012). Design patterns selection: An automatic two-phase method. *Journal of Systems and Software*, *85*(2), 408–424. https://doi.org/10.1016/j.jss.2011.08.031

Hemert, J., y Barker, A. (2008). Scientific Workflow: A Survey and Research Directions. *Parallel Processing and Applied Mathematics*, 746–753.

https://doi.org/10.1145/2213836.2213899

IEEE. (1990). *IEEE Standard Glossary of Software Engineering Terminology*. *Office* (Vol. 121990). IEEE. https://doi.org/10.1109/IEEESTD.1990.101064

Karastoyanova, D., y Andrikopoulos, V. (2013). EScienceSWaT - Towards an eScience software engineering methodology. In *Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC* (pp. 229–238). IEEE Computer Society. https://doi.org/10.1109/EDOCW.2013.32

Kelly, D. (2007). A software chasm: Software Engineering and Scientific Computing. *Computing Trends*, *7*(1), 1–3.

Kelly, D. (2015). Software Development Viewed as Knowledge Acquisition: Towards Understanding the Development of Scientific Software. *J. Systems and Software*, *106*, 50–61.

Kelly, D., y Sanders, R. (2008). Assessing the quality of scientific software. *In First International Workshop on Software Engineering for Computational Science and Engineering (May 2008)*, (May). Retrieved from http://www.cse.msstate.edu/~SECSE08/Papers/Kelly.pdf%5Cnhttp://secse08.cs.ua.edu/Papers/Kelly.pdf

Kelly, D., Thorsteinson, S., y Hook, D. (2011). Scientific Software Testing: Analysis with Four Dimensions. *Software, IEEE*, *28*(3), 84–90. https://doi.org/10.1109/MS.2010.88

Kim, D., France, R., Ghosh, S., y Song, E. (2003). A UML-Based Metamodeling Language to Specify Design Patterns. *Proceedings of the 2nd Workshop in Software Model Engineering*, 9. Retrieved from http://cs.baylor.edu/~song/publications/uml03-WiSME.pdf

Kim, S., y Carrington, D. (2004). Using integrated metamodeling to define OO design patterns with object-Z and UML. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, 257–264. https://doi.org/10.1109/APSEC.2004.108

Lee, W.-T., Hsu, K.-H., y Lee, J. (2012). Designing software architecture with use case blocks using the design structure matrix. *Proceedings - 2012 International Symposium on Computer, Consumer and Control, IS3C 2012*, 654–657. https://doi.org/10.1109/IS3C.2012.170

Li, Y., Guzman, E., Tsiamoura, K., Schneider, F., y Bruegge, B. (2015). Automated requirements extraction for scientific software. *Procedia Computer Science*, *51*(1), 582–591. https://doi.org/10.1016/j.procs.2015.05.326

Li, Y., Harutunian, M., Narayan, N., Bruegge, B., y Buse, G. (2011). Requirements Engineering for Scientific Computing: A Model-Based Approach. *E-Science Workshops (eScienceW), 2011 IEEE Seventh International Conference on*, 128–134. https://doi.org/10.1109/eScienceW.2011.30

Morris, C., y Segal, J. (2012). Lessons Learned from a Scientific Software Development Project. *Insights: Software Development*, *12*, 9–12. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6265073

Mustapha, M., y Nik Daud, N. G. (2011). UML diagram for design patterns. *Communications in Computer and Information Science*, *181 CCIS*(PART 3), 215–230. https://doi.org/10.1007/978-3-642-22203-0_19

Panagiotis, M., y Margaritis, K. (2014). Scientific computations on multi-score systems using different programming frameworks. *Applied Numerical Mathematics*, 3–7. Retrieved from http://dx.doi.org/10.1016/j.apnum.2014.12.008

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, *15*(12), 1053–1058. https://doi.org/10.1145/361598.361623

Pérez, J., Rodríguez, G., y Pomares, S. (2011). Pattern Object-Oriented Architecture for Multirate Integration Methods. In *Electrical Communications and Computers (CONIELECOMP)* (pp. 4–6). Puebla: IEEE.

Peterson, G., y Guthrie, S. E. (2013). *Encyclopedia of Sciences and Religions*. *Encyclopedia of Sciences and Religions*. https://doi.org/10.1007/978-1-4020-8265-8

Pree, W., y Sikora, H. (1995). Design Patterns for Object-Oriented Software Development. In *ICSE 97* (p. 268). Boston.

Rasmussen, L. V., Thompson, W. K., Pacheco, J. A., Kho, A. N., Carrell, D. S., Pathak, J., … Starren, J. B. (2014). Design patterns for the development of electronic health record-driven phenotype extraction algorithms. *Journal of Biomedical Informatics*, *51*, 280–286. https://doi.org/10.1016/j.jbi.2014.06.007

Rodriguez, G., Muñoz, J., y Fernandez, R. (2004). Scientific software design through scientific computing patterns. In *Iasted International Conference Modelling, Simulation and optimization* (pp. 2–5). Kauai, Hawaii, USA.

Roure, D. De, y Goble, C. (2009). Six Principles of Software Design to Empower Scientists Scientific Workflow Management Systems. *Software, IEEE*, *26*, 88–95. https://doi.org/10.1109/MS.2009.22

Sabatucci, L., Cossentino, M., y Susi, A. (2015). A goal-oriented approach for representing and using design patterns. *Journal of Systems and Software*, *110*, 136–154. https://doi.org/10.1016/j.jss.2015.07.040

Sanders, R., y Kelly, D. (2008). Developing scientific software Dealing with Risk in Scientific Software Development. *Focus: Developing Scientific Software*, *8*, 8.

Seffah, A. (2015). *The Patterns of HCI Design: Origin, Perceptions, and Misconceptions* (1st ed.). Switzerland: Springer International. https://doi.org/10.1007/978-3-319-15687-3

Segal, J. (2005). *When software engineers met research scientists: A case study*. *Empirical Software Engineering* (Vol. 10). Walton Hall. https://doi.org/10.1007/s10664-005-3865-y

Segal, J. (2007). Some Problems of Professional End User Developers. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)* (pp. 111–118). United Kingdom: IEEE Computer Society. https://doi.org/10.1109/VLHCC.2007.17

Segal, J. (2008). Scientists and software engineers: A tale of two cultures. *Proceedings of the Psychology of Programming Interest Group*, (September).

Segal, J. (2009). Some challenges facing software engineers developing software for scientists. In *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering* (pp. 9–14). Vancouver: IEEE Computer Society. https://doi.org/10.1109/SECSE.2009.5069156

Segal, J., y Morris, C. (2008). Developing Scientific Software. *IEEE Software*, 1–3.

Segal, J., y Morris, C. (2009). Developing Scientific Software, Part 2. *IEEE Software*, 1.

Shalloway, A., y Trott, J. R. (2004). Design patterns explained: a new perspective on object-oriented design. *AddisonWesley Publ Co*, 334. Retrieved from http://www.amazon.com/Design-Patterns-Explained-Perspective-Object-Oriented/dp/0201715945

Smith, J. M., y Stotts, D. (2002). Elemental design patterns: A link between architecture and object semantics. *Proceedings of OOPSLA*. Retrieved from http://www.cs.unc.edu/techreports/02-011.pdf

Stelting, S., y Maassen, O. (2002). *Applied Java Patterns* (1st ed.). Sun Java Series.

Stevens, W. P., Myers, G. J., y Constantine, L. L. (1974). Structured Design. *IBM Systems Journal*, 115–139. Retrieved from

http://download.springer.com.ezproxy.unal.edu.co/static/pdf/86/bok%253A978-3-8348-8675-0.pdf?originUrl=http%3A%2F%2Flink.springer.com%2Fbook%2F10.1007%2F978-3-8348-8675-0ytoken2=exp=1479117641~acl=%2Fstatic%2Fpdf%2F86%2Fbok%25253A978-3-8348-8675-0.pdf%3Fo

Streekmann, N. (2012). Software Architecture. In *Clustering-Based Support for Software Architecture Restructuring* (1st ed., pp. 9–22). Berlin: Vieweg+Teubner Verlag. https://doi.org/10.1007/978-3-8348-8675-0

The Object Management Group. (2005). Unified Modeling Language : Infrastructure. *The Object Management Group*, (July), 226. https://doi.org/10.1007/s00500-003-0307-x

Wagner, M. (2007). Evolution from a scientific application to an applicable product. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, 223–229. https://doi.org/10.1109/CSMR.2007.22

Wilson, G., Aruliah, D. A., Brown, C. T., Chue Hong, N. P., Davis, M., Guy, R. T., … Wilson, P. (2014). Best Practices for Scientific Computing. *PLoS Biology*, *12*(1). https://doi.org/10.1371/journal.pbio.1001745

Winetzhammer, S., y Westfechtel, B. (2015). Software Technologies. In *ICSOFT* (Vol. 555, pp. 331–348). Switzerland: Springer International. https://doi.org/10.1007/978-3-319-25579-8

Zapata, C. M. (2012). The UNC-Method Revisited: Elements of the New Approach Eliciting Software Requirements in a Complete, Consistent, and Correct Way. *LAP LAMBERT Academic Publishing GmbH y Co*, (5), 2013. Retrieved from https://www.lap-publishing.com/catalog/details/store/de/book/978-3-8484-0759-0/the-unc-method-revisited:-elements-of-the-new-approach?search=organic products

Zapata, C. M., y Arango, F. (2007). *Construcción automática de esquemas conceptuales a partir de lenguaje natural* (Primera ed). Medellín.

Zapata, C. M., Chaverra, J., y Vill, H. (2012). Un caso de estudio para la generación automática de código a partir de esquemas preconceptuales A study case for automatic code generation based on pre-conceptual schemas. *Cuaderno ACTIVA, Tecnológico de Antioquia*, *4*(4), 9–24.

Zapata, C. M., Gelbukh, A., y Isaza, F. A. (2006). Pre-conceptual Schema : a Conceptual-

Graph-like Knowledge Representation for Requirements Elicitation State-of-the-Art in KR-based Requirements Elicitation. *Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4–6. Retrieved from http://mail.gelbukh.com/CV/Publications/2006/MICAI-2006-Preconceptual.pdf

Zapata, C. M., Ruiz, L. M., y Pastor, O. (2010). Desde esquemas preconceptuales hacia OO-Method From pre-conceptual schemas to OO-Method. *Revista Facultad de Ingeniería Universidad de Antioquia*, *56*, 203–212.

Zhu, H. (2005). *Basic Concepts of Design*. *Software Design Methodology* (1st ed.). Elsevier Ltd.

Ackroyd, K., Kinder, S., Mant, G., Miller, M., Ramsdale, C., y Stephenson, P. (2008). Scientific Software Development at a Research Facility. *IEEE Software*, 8.

Ahmed, Z. (2015). Essential Design Modeling for Scientific Software Solutions Development. *Peerj*.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., y Angel, S. (1977). *A Pattern Language* (1st ed.). New York: Oxford University Press. Retrieved from http://library.uniteddiversity.coop/Ecological_Building/A_Pattern_Language.pdf

Aras, K., Cickovski, T., y Izaguirre, J. (2004). Empirical Evaluation of Design Patterns in Scientific Application. *Citeseer*. Retrieved from http://scholar.google.com/scholar?hl=enybtnG=Searchyq=intitle:Empirical+Evaluation+of+Design+Patterns+in+Scientific+Application#0

Barbieri, D., Cardellini, V., Filippone, S., y Rouson, D. (2012). Design Patterns for Scientific Computations on Sparse Matrices. *Lectures in Computer Science*. https://doi.org/10.1007/3-540-68339-9_34

Bashir, R. S., Lee, S. P., Khan, S. U. R., Chang, V., y Farid, S. (2016). UML models consistency management: Guidelines for software quality manager. *International Journal of Information Management*, *36*(6), 883–899. https://doi.org/10.1016/j.ijinfomgt.2016.05.024

Bass, L., Clements, P., y Kazman, R. (2003). *Software Architecture in Practice*. *Vasa* (Third, Vol. 1st). Boston: Pearson. https://doi.org/10.1024/0301-1526.32.1.54

Bonfè, M., Fantuzzi, C., y Secchi, C. (2013). Design patterns for model-based automation software design and implementation. *Control Engineering Practice*, *21*(11), 1608–1619. https://doi.org/10.1016/j.conengprac.2012.03.017

Carver, J., Kendall, R., Squires, S., y Post, D. (2007). Software Development

Environments for Scientific and Engineering Software: A Series of Case Studies. In *29th International conference on Software Engineering (ICSE'07)* (p. 10). Mississipi: IEEE.

Charles, B. (2002). *Patterns in scientific software: An Introduction*. *Feature software patterns* (2nd ed.). Computing In Science y Engineering.

Cickovski, T., Matthey, T., y Izaguirre, J. (2004). Design Patterns for Generic Object-Oriented Scientific Software. Bergen.

Clune, R., Connor, J. J., Ochsendorf, J. A., y Kelliher, D. (2012). An object-oriented architecture for extensible structural design software. *Computers and Structures*, *100–101*, 1–17. https://doi.org/10.1016/j.compstruc.2012.02.002

Cuéllar, E., Muñoz, G. R., y Gómez, J. (2005). Aplicación de Patrones de Software en el Dominio de los Simuladores de Procesos Dinámicos. Aguascalientes.

Davis, C. (2012). Software engineering for computational science and engineering. *Computing in Science y Engineering*, *14*(2), 8–11. https://doi.org/10.1109/MCSE.2012.31

De Oliveira, K. S., y Soares, M. S. (2013). A systematic review on aspects in software architecture design. *Proceedings - International Conference of the Chilean Computer Science Society, SCCC*, 21–28. https://doi.org/10.1109/SCCC.2012.10

Decyk, V., y Gardner, H. (2006). Object-Oriented Design Patterns in Scientific Codes using Fortran95. *Elsevier Science2*, 4–6.

Decyk, V., y Gardner, H. (2007). Object-Oriented Design Patterns in Fortran. *Elsevier Science*, (July 2007).

Dijkstra, E. W. (1972). Selected Writings on Computing: A Personal Perspective. *Ewd*, *477*, 60–66. https://doi.org/10.1007/978-1-4612-5695-3

Dodig-crnkovic, G. (2002). Scientific Methods in Computer Science. *Computer (Long. Beach. Calif).*, 126–130. Retrieved from http://www.mrtc.mdh.se/~gdc/work/cs_method.pdf

Dong, J. (2002). UML extensions for design pattern compositions. *Journal of Object Technology*, *1*(5), 149–161. Retrieved from http://www.jot.fm/issues/issue_2002_11/article3.pdf

Dong, J. (2003). Representing the applications and compositions of design patterns in UML. *SAC '03: Proceedings of the 2003 ACM Symposium on Applied Computing*, 1092–1098 PU–ACM Press. Retrieved from

http://dx.doi.org/10.1145/952532.952746

Dong, J., y Yang, S. (2003). Visualizing design patterns with a UML profile. *Proceedings - 2003 IEEE Symposium on Human Centric Computing Languages and Environments, HCC 2003*, 123–125. https://doi.org/10.1109/HCC.2003.1260215

Dong, J., Yang, S., y Zhang, K. (2007). Visualizing design patterns in their applications and compositions. *IEEE Transactions on Software Engineering*, *33*(7), 433–453. https://doi.org/10.1109/TSE.2007.1012

Fowler, M. (2004). *UML Distilled: A brief guide to the Standard Object Modeling Language* (3rd ed.). Canada: Addison Wesley. Retrieved from http://www.agentgroup.unimore.it/~nicola/courses/IngegneriaDelSoftware/uml/books/UMLDistilled.pdf

Fowler, M., y Kendall, S. (1999). *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (2nd ed.). Canada: Addison Wesley. Retrieved from www.awl.com/cseng/

Gamma, E., Helm, R., Johnson, R., y Vissides, J. (1994). *Design Patterns: Elements of reussable oriented-objected software*. (Ke. Zhang, Ed.) (6th ed.). Sidney.

Gardner, H. (2004). *Design Patterns in Scientific Software*. *Design patterns in scientific software* (4th ed.). Berlin: Springer-Verlag.

Gardner, H., y Manduchi, G. (2007). Design Patterns for e-Science. *Elsevier Science*, 64–81.

Garlan, D., Shaw, M., Company, W. S. P., y Institute, C. M. U. S. E. (1994). An Introduction to Software Architecture. *Cmu/Sei-94-Tr-21*, (January).

Garlan, D., T. Monroe, R., y Wile, D. (2000). ACME: An architectural description of component-based systems. *Foundations of Component-Based Systems*, 47–68. Retrieved from http://www.cs.cmu.edu/afs/cs/project/able/ftp/acme-fcbs/acme-fcbs.pdf

Hannay, E., MacLeod, C., Singer, J., Langtangen, H. P., Pfahl, D., y Wilson, G. (2009). How Do Scientist Develop and Use Scientific Software? In *SECSE'09* (p. 8). Vancouver: IEEE Computer Society.

Hartmanis, J., y Leeuwen, J. Van. (2001). *Large-Scale Scientific Computing* (1st ed.). Sozopol: Springer.

Hasheminejad, S. M. H., y Jalili, S. (2012). Design patterns selection: An automatic two-phase method. *Journal of Systems and Software*, *85*(2), 408–424.

https://doi.org/10.1016/j.jss.2011.08.031

Hemert, J., y Barker, A. (2008). Scientific Workflow: A Survey and Research Directions. *Parallel Processing and Applied Mathematics*, 746–753. https://doi.org/10.1145/2213836.2213899

IEEE. (1990). *IEEE Standard Glossary of Software Engineering Terminology*. *Office* (Vol. 121990). IEEE. https://doi.org/10.1109/IEEESTD.1990.101064

Karastoyanova, D., y Andrikopoulos, V. (2013). EScienceSWaT - Towards an eScience software engineering methodology. In *Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC* (pp. 229–238). IEEE Computer Society. https://doi.org/10.1109/EDOCW.2013.32

Kelly, D. (2007). A software chasm: Software Engineering and Scientific Computing. *Computing Trends*, *7*(1), 1–3.

Kelly, D. (2015). Software Development Viewed as Knowledge Acquisition: Towards Understanding the Development of Scientific Software. *J. Systems and Software*, *106*, 50–61.

Kelly, D., y Sanders, R. (2008). Assessing the quality of scientific software. *In First International Workshop on Software Engineering for Computational Science and Engineering (May 2008)*, (May). Retrieved from http://www.cse.msstate.edu/~SECSE08/Papers/Kelly.pdf%5Cnhttp://secse08.cs.ua.edu/Papers/Kelly.pdf

Kelly, D., Thorsteinson, S., y Hook, D. (2011). Scientific Software Testing: Analysis with Four Dimensions. *Software, IEEE*, *28*(3), 84–90. https://doi.org/10.1109/MS.2010.88

Kim, D., France, R., Ghosh, S., y Song, E. (2003). A UML-Based Metamodeling Language to Specify Design Patterns. *Proceedings of the 2nd Workshop in Software Model Engineering*, 9. Retrieved from http://cs.baylor.edu/~song/publications/uml03-WiSME.pdf

Kim, S., y Carrington, D. (2004). Using integrated metamodeling to define OO design patterns with object-Z and UML. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, 257–264. https://doi.org/10.1109/APSEC.2004.108

Lee, W.-T., Hsu, K.-H., y Lee, J. (2012). Designing software architecture with use case blocks using the design structure matrix. *Proceedings - 2012 International Symposium on Computer, Consumer and Control, IS3C 2012*, 654–657. https://doi.org/10.1109/IS3C.2012.170

Li, Y., Guzman, E., Tsiamoura, K., Schneider, F., y Bruegge, B. (2015). Automated requirements extraction for scientific software. *Procedia Computer Science*, *51*(1), 582–591. https://doi.org/10.1016/j.procs.2015.05.326

Li, Y., Harutunian, M., Narayan, N., Bruegge, B., y Buse, G. (2011). Requirements Engineering for Scientific Computing: A Model-Based Approach. *E-Science Workshops (eScienceW), 2011 IEEE Seventh International Conference on*, 128–134. https://doi.org/10.1109/eScienceW.2011.30

Morris, C., y Segal, J. (2012). Lessons Learned from a Scientific Software Development Project. *Insights: Software Development*, *12*, 9–12. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6265073

Mustapha, M., y Nik Daud, N. G. (2011). UML diagram for design patterns. *Communications in Computer and Information Science*, *181 CCIS*(PART 3), 215–230. https://doi.org/10.1007/978-3-642-22203-0_19

Panagiotis, M., y Margaritis, K. (2014). Scientific computations on multi-score systems using different programming frameworks. *Applied Numerical Mathematics*, 3–7. Retrieved from http://dx.doi.org/10.1016/j.apnum.2014.12.008

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, *15*(12), 1053–1058. https://doi.org/10.1145/361598.361623

Pérez, J., Rodríguez, G., y Pomares, S. (2011). Pattern Object-Oriented Architecture for Multirate Integration Methods. In *Electrical Communications and Computers (CONIELECOMP)* (pp. 4–6). Puebla: IEEE.

Peterson, G., y Guthrie, S. E. (2013). *Encyclopedia of Sciences and Religions*. *Encyclopedia of Sciences and Religions*. https://doi.org/10.1007/978-1-4020-8265-8

Pree, W., y Sikora, H. (1995). Design Patterns for Object-Oriented Software Development. In *ICSE 97* (p. 268). Boston.

Rasmussen, L. V., Thompson, W. K., Pacheco, J. A., Kho, A. N., Carrell, D. S., Pathak, J., … Starren, J. B. (2014). Design patterns for the development of electronic health record-driven phenotype extraction algorithms. *Journal of Biomedical Informatics*, *51*, 280–286. https://doi.org/10.1016/j.jbi.2014.06.007

Rodriguez, G., Muñoz, J., y Fernandez, R. (2004). Scientific software design through scientific computing patterns. In *Iasted International Conference Modelling, Simulation and optimization* (pp. 2–5). Kauai, Hawaii, USA.

Roure, D. De, y Goble, C. (2009). Six Principles of Software Design to Empower Scientists Scientific Workflow Management Systems. *Software, IEEE*, *26*, 88–95. https://doi.org/10.1109/MS.2009.22

Sabatucci, L., Cossentino, M., y Susi, A. (2015). A goal-oriented approach for representing and using design patterns. *Journal of Systems and Software*, *110*, 136–154. https://doi.org/10.1016/j.jss.2015.07.040

Sanders, R., y Kelly, D. (2008). Developing scientific software Dealing with Risk in Scientific Software Development. *Focus: Developing Scientific Software*, *8*, 8.

Seffah, A. (2015). *The Patterns of HCI Design: Origin, Perceptions, and Misconceptions* (1st ed.). Switzerland: Springer International. https://doi.org/10.1007/978-3-319-15687-3

Segal, J. (2005). *When software engineers met research scientists: A case study*. *Empirical Software Engineering* (Vol. 10). Walton Hall. https://doi.org/10.1007/s10664-005-3865-y

Segal, J. (2007). Some Problems of Professional End User Developers. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)* (pp. 111–118). United Kingdom: IEEE Computer Society. https://doi.org/10.1109/VLHCC.2007.17

Segal, J. (2008). Scientists and software engineers: A tale of two cultures. *Proceedings of the Psychology of Programming Interest Group*, (September).

Segal, J. (2009). Some challenges facing software engineers developing software for scientists. In *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering* (pp. 9–14). Vancouver: IEEE Computer Society. https://doi.org/10.1109/SECSE.2009.5069156

Segal, J., y Morris, C. (2008). Developing Scientific Software. *IEEE Software*, 1–3.

Segal, J., y Morris, C. (2009). Developing Scientific Software, Part 2. *IEEE Software*, 1.

Shalloway, A., y Trott, J. R. (2004). Design patterns explained: a new perspective on object-oriented design. *AddisonWesley Publ Co*, 334. Retrieved from http://www.amazon.com/Design-Patterns-Explained-Perspective-Object-Oriented/dp/0201715945

Smith, J. M., y Stotts, D. (2002). Elemental design patterns: A link between architecture and object semantics. *Proceedings of OOPSLA*. Retrieved from http://www.cs.unc.edu/techreports/02-011.pdf

Stelting, S., y Maassen, O. (2002). *Applied Java Patterns* (1st ed.). Sun Java Series.

Stevens, W. P., Myers, G. J., y Constantine, L. L. (1974). Structured Design. *IBM Systems Journal*, 115–139. Retrieved from http://download.springer.com.ezproxy.unal.edu.co/static/pdf/86/bok%253A978-3-8348-8675-0.pdf?originUrl=http%3A%2F%2Flink.springer.com%2Fbook%2F10.1007%2F978-3-8348-8675-0ytoken2=exp=1479117641~acl=%2Fstatic%2Fpdf%2F86%2Fbok%25253A978-3-8348-8675-0.pdf%3Fo

Streekmann, N. (2012). Software Architecture. In *Clustering-Based Support for Software Architecture Restructuring* (1st ed., pp. 9–22). Berlin: Vieweg+Teubner Verlag. https://doi.org/10.1007/978-3-8348-8675-0

The Object Management Group. (2005). Unified Modeling Language : Infrastructure. *The Object Management Group*, (July), 226. https://doi.org/10.1007/s00500-003-0307-x

Wagner, M. (2007). Evolution from a scientific application to an applicable product. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, 223–229. https://doi.org/10.1109/CSMR.2007.22

Wilson, G., Aruliah, D. A., Brown, C. T., Chue Hong, N. P., Davis, M., Guy, R. T., … Wilson, P. (2014). Best Practices for Scientific Computing. *PLoS Biology*, *12*(1). https://doi.org/10.1371/journal.pbio.1001745

Winetzhammer, S., y Westfechtel, B. (2015). Software Technologies. In *ICSOFT* (Vol. 555, pp. 331–348). Switzerland: Springer International. https://doi.org/10.1007/978-3-319-25579-8

Zapata, C. M. (2012). The UNC-Method Revisited: Elements of the New Approach Eliciting Software Requirements in a Complete, Consistent, and Correct Way. *LAP LAMBERT Academic Publishing GmbH y Co*, (5), 2013. Retrieved from https://www.lap-publishing.com/catalog/details/store/de/book/978-3-8484-0759-0/the-unc-method-revisited:-elements-of-the-new-approach?search=organic products

Zapata, C. M., y Arango, F. (2007). *Construcción automática de esquemas conceptuales a partir de lenguaje natural* (Primera ed). Medellín.

Zapata, C. M., Chaverra, J., y Vill, H. (2012). Un caso de estudio para la generación automática de código a partir de esquemas preconceptuales A study case for

automatic code generation based on pre-conceptual schemas. *Cuaderno ACTIVA, Tecnológico de Antioquia*, *4*(4), 9–24.

Zapata, C. M., Gelbukh, A., y Isaza, F. A. (2006). Pre-conceptual Schema : a Conceptual-Graph-like Knowledge Representation for Requirements Elicitation State-of-the-Art in KR-based Requirements Elicitation. *Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4–6. Retrieved from http://mail.gelbukh.com/CV/Publications/2006/MICAI-2006-Preconceptual.pdf

Zapata, C. M., Ruiz, L. M., y Pastor, O. (2010). Desde esquemas preconceptuales hacia OO-Method From pre-conceptual schemas to OO-Method. *Revista Facultad de Ingeniería Universidad de Antioquia*, *56*, 203–212.

Zhu, H. (2005). *Basic Concepts of Design*. *Software Design Methodology* (1st ed.). Elsevier Ltd.