UNIVERSIDAD
REY JUAN CARLOS

Grado en Ingeniería de Computadores

Curso Académico 2023 / 2024

Trabajo Fin de Grado

Título del proyecto:
Puentes tecnológicos entre PCN y smart contracts

Autor: Álvaro Barrio Luquero

Tutores: Juan Manuel Vara Mesa,
Cristian Gómez Macías

RESUMEN

En los modelos de negocio existen procesos en los que se establece el funcionamiento de dichos negocios. En estos procesos se pueden distinguir las partes que intervienen, el desarrollo del producto o servicio que ofrecen y el valor que reciben a cambio junto a las diferentes actividades que dan lugar a estos intercambios de valor. Existen herramientas para definir y analizar estos procesos de negocio; una de ellas es PCN.

A la hora de definir estos procesos de negocio o servicio, se puede analizar y obtener información detallada sobre su flujo de funcionamiento, las interacciones que suceden en él y sus resultados. Toda esta información junto a la tecnología adecuada puede ser procesada y automatizada, permitiendo potenciar el rendimiento del negocio. Un ejemplo de esta tecnología son los smart contracts, capaces de integrarse en el negocio facilitando las interacciones entre las partes involucradas en un proceso, ya sean transacciones u otros acuerdos, al asegurar su cumplimento de manera automática y minimizar la necesidad de acción manual.

Los smart contracts son scripts alojados en blockchain que se ejecutan al cumplir condiciones o cláusulas acordadas entre los participantes del contrato a la hora de establecerlo, pudiendo efectuar transacciones u otras acciones de forma automática. Entre los múltiples beneficios que aporta su uso está la eficiencia al ser en esencia un script, transparencia al no contar con intermediarios y la seguridad que brinda el hecho de permanecer alojado en blockchain.

Sin embargo, aunque los smart contracts y la tecnología que gira en torno a blockchain está presente en diferentes sectores, esta tecnología no está al alcance de cualquier profesional. Esto da lugar a una brecha tecnológica entre personas con un perfil informático dedicadas a explorar y desarrollar esta tecnología y las personas dedicadas a otras tareas dentro de múltiples sectores, como es el del negocio, que desconocen lo relacionado con los smart contracts y no cuentan con los conocimientos para tratarlos, en especial los conocimientos en programación.

Bajo el contexto descrito, este proyecto pretende y logra acortar la brecha tecnológica existente para acercar los modelos de negocio y servicios a una de las tecnologías más actuales, permitiendo adoptar su uso para aprovechar todas las ventajas que ofrece y poder potenciar las áreas de negocio y servicio sin requerir de un conocimiento técnico por parte de los profesionales de dichas áreas.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

Capítulo 1 - Introducción

En este primer capítulo se comienza explicando el contexto que hay detrás de la realización de este Trabajo de Fin de Grado. Esto abarca un conjunto de espacios dedicados a explicar la **motivación** tras la idea del proyecto y el **objetivo** al que se pretende llegar gracias a la solución planteada. También se detalla la **metodología** seguida y los **medios utilizados** para abordar el objetivo mencionado, así como un resumen de la **estructura de contenidos** que sigue esta memoria.

1.1 Motivación

Los smart contracts junto con la tecnología blockchain han demostrado que tienen la capacidad de cambiar la forma que tienen los negocios de llevar a cabo sus actividades, particularmente en lo que a servicios se refiere. Estas tecnologías traen consigo una serie de beneficios que pueden ser aprovechados para el diseño de servicios, la servitización y los sistemas producto-servicio. Entre los beneficios mencionados, destacan la capacidad de automatización de procesos, una mejora en la eficacia con la que se realizan estos procesos y la transparencia en el tratamiento de las interacciones que tienen lugar entre las partes involucradas en los procesos y en la trazabilidad de tanto productos como servicios para obtener una ganancia de confianza a nivel de gestión y ejecución [1][2].

Sin embargo, el proceso de aprendizaje necesario para utilizar y sacar partido a esta tecnología requiere del conocimiento técnico propio de un perfil de desarrollador. Aunque existen herramientas diseñadas para facilitar el desarrollo de smart contracts por parte de los desarrolladores, no proporcionan el nivel de **accesibilidad** suficiente para acercar a los profesionales involucrados en negocio como analistas o diseñadores de servicios. En consecuencia, la base de conocimiento requerida sobre programación y blockchain excluye a estos profesionales de utilizar dichas herramientas en sus funciones, que ven compleja e improbable su adopción dado su desconocimiento técnico.

Esta situación hace evidente la existencia de un problema, una **brecha tecnológica** que dificulta la posibilidad de adoptar los smart contracts por parte de profesionales no técnicos. Dicha brecha impide a estos profesionales disponer de ventajas como la automatización de acuerdos y transacciones o el ahorro de costes en intermediarios. El hecho de reducir o cerrar esta brecha haría posible no solo que los responsables de negocio mejoren en el desempeño de sus tareas, sino que, en consecuencia, potenciaría el rendimiento de los negocios dados los beneficios mencionados al principio.

Motivado por estas causas, en este trabajo se aborda la identificación y explotación de las **relaciones** existentes entre la notación PCN (Process Chain Network), una herramienta dirigida a la representación de cadenas de procesos de negocio y centrada en la interacción de las partes que intervienen en estos procesos [3], y los smart contracts con el subsiguiente fin de construir un **puente tecnológico** que permita acercar

los smart contracts a los profesionales de negocio, ayudando a difundir esta tecnología entre más profesionales e innovando en el proceso de desarrollo de servicios.

1.2 Objetivos

El objetivo principal de este Trabajo de Fin de Grado, como sugiere el título del proyecto, es construir un **puente tecnológico** que conecte PCN y los smart contracts, dos herramientas con mucho potencial en múltiples sectores y sin ningún tipo de relación directa analizada y explotada, con el fin de acercar ambas.

A fin de que se puedan conectar ambas tecnologías de manera bidireccional, se han de construir en realidad dos puentes tecnológicos. Para ello se plantea el desarrollo de dos herramientas hechas con Xtend y Java en el entorno de desarrollo Eclipse, una que permita **transformar un PCN en un smart contract** funcional y otra con la que poder **transformar un smart contract en un PCN**. Estas transformaciones se basan en la previa identificación y definición de las correspondencias entre los elementos de ambas partes. Las herramientas a diseñar se apoyan en el uso de los denominados DSL (Domain Specific Language), lenguajes diseñados para resolver problemas dentro de campos o dominios concretos [4]. .Los DSL utilizados son los proporcionados por las herramientas INNoVaServ y SmaC, que definen la notación PCN y los smart contracts, respectivamente, como modelos manipulables bajo el entorno EMF (Eclipse Modeling Framework), un entorno diseñado para facilitar el desarrollo de aplicaciones en Eclipse basadas en el uso de modelos [5]. En la Figura 1 se puede observar de forma visual y simplificada el objetivo principal que se ha detallado hasta ahora, que describe a su vez la solución tecnológica planteada a alto nivel.

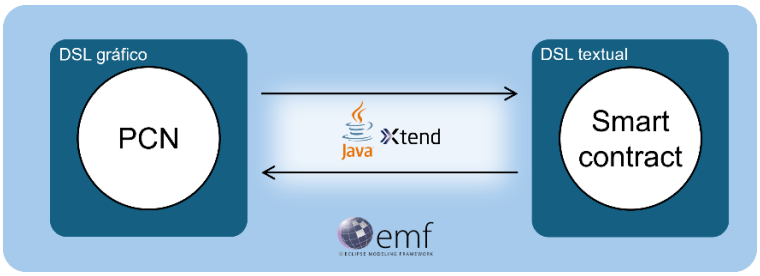


Figura 1. Solución tecnológica del proyecto.

Conseguir esto implica establecer una serie de **objetivos secundarios**:

- Desarrollar un mecanismo tecnológico en forma de plug-in para el entorno de desarrollo, con la finalidad de lanzar un proceso de ejecución semiautomático de la transformación bidireccional entre los modelos PCN y los modelos de smart contracts.
- Conseguir plasmar el flujo de funcionamiento de un diagrama PCN en Solidity. Para poder convertir un diagrama PCN en un smart contract de la manera más precisa, el smart contract resultante debe comportarse, dentro de las posibilidades de Solidity, como si de un diagrama PCN llevado a la práctica se tratara. Esto significa implementar lógicas de exclusión, espera, decisión, etc. junto a otros componentes que reflejen el comportamiento práctico de un diagrama PCN y transmitan la misma información.

- Conseguir el mayor porcentaje de código Solidity generado a partir de un PCN. La intención con el puente de PCN a smart contract es conseguir la máxima cantidad de código posible de forma automatizada. Sin embargo, para poder alcanzar un porcentaje elevado, es necesario el diseño e implementación de pequeñas interfaces de usuario a modo de guía, de forma que este soporte asista al usuario para poder recoger un mayor porcentaje de la lógica del smart contract, haciendo que este sea más funcional.

Es apreciable que la mayoría de los objetivos secundarios se sitúan en la línea de desarrollo del primer puente de PCN a smart contract. Esto se debe a que es el puente que mayor labor de conversión y generación de código supone. En cambio, el puente inverso de smart contract a PCN puede verse como deshacer los pasos dados con el primer puente.

1.3 Método de trabajo

Para lograr los objetivos propuestos, la metodología seguida se basa en la división del proyecto en varias **fases**, cada una con una pequeña meta o hito, con las que definir un orden de trabajo que además permite disponer puntos de control en los que revisar cada fase.

Se trata de una metodología similar a la conocida metodología Waterfall (en cascada), característica por un flujo descendente y lineal, que permite lograr una buena estructuración en proyectos con un objetivo claro desde el principio. De este modo y como se puede observar en la Figura 2, el desarrollo se establece de la siguiente forma:

- **Fase 1. Análisis:** Una fase inicial centrada en investigar las tecnologías relacionadas y las tecnologías principales, PCN y Solidity, subdividiendo la fase en varias tareas:
    - **Estudio de la notación PCN:** Tarea centrada en investigar PCN, la primera de las tecnologías de los puentes. Se estudia su origen, utilidad, funcionamiento y los diferentes elementos que lo componen junto a su significado. El objetivo es saber crear un diagrama PCN y saber interpretar un diagrama PCN previamente creado.
    - **Estudio del lenguaje Solidity:** Tarea cuyo propósito pasa por aprender a programar un smart contract, debiendo conocer para ello el lenguaje que utilizan, Solidity. El objetivo en esta tarea es conocer los elementos de un smart contract a través de los elementos de Solidity y saber cómo programar uno funcional siguiendo los estándares que dicta el lenguaje.
    - **Identificación de relaciones entre PCN y Solidity:** Tarea cuyo propósito pasa por situar sobre la mesa los elementos tanto de PCN como los de un smart contract (Solidity) previamente estudiados y establecer una correspondencia entre ellas para terminar por esbozar la forma de hacer un primer puente.
- **Fase 2. Solución tecnológica:** Una vez terminados los estudios previos y establecida la correspondencia entre PCN y Solidity, en esta fase comienza el desarrollo de ambos puentes tecnológicos:
    - **Transformación de PCN a smart contract:** En primer lugar, se desarrolla la herramienta para transformar un modelo PCN en código Solidity (smart contract)

siguiendo las correspondencias establecidas en la fase de análisis. Para ello se pone a punto el entorno de desarrollo, Eclipse, y se instala el plug-in de INNoVaServ con la funcionalidad para modelar diagramas PCN y poder consumir su contenido. Junto a ello, se instala en Eclipse el resto de plug-ins necesarios para hacer funcionar el modelo PCN y finalmente se implementa la funcionalidad.

- **Transformación de smart contract a PCN:** Una vez implementado el primer puente de PCN a smart contract, se desarrolla la segunda herramienta que permite transformar un smart contract obtenido con el primer puente en un modelo PCN. Para el desarrollo de este segundo puente se instala el plug-in del DSL textual SmaC para poder obtener el smart contract de origen en forma de modelo y poder consumir su contenido para poder crear su modelo PCN resultante.

- **Fase 3. Validación de la solución:** En esta fase se da por completado el desarrollo de las herramientas y se validan poniéndose a prueba con casos basados en distintos procesos de negocio reales, convirtiendo los modelos PCN de estos procesos en Smart Conctracts y convirtiendo los smart contracts de otros procesos de negocio en modelos PCN.
- **Fase 4. Memoria:** Finalmente, una vez completada y validada la solución tecnológica, se da por finalizada la parte funcional del proyecto y queda documentar en esta memoria el proceso completo. Aunque esta fase se sitúa al final, también se ha trabajado paralelamente en la misma redactando algunos fragmentos de este documento durante fases previas.

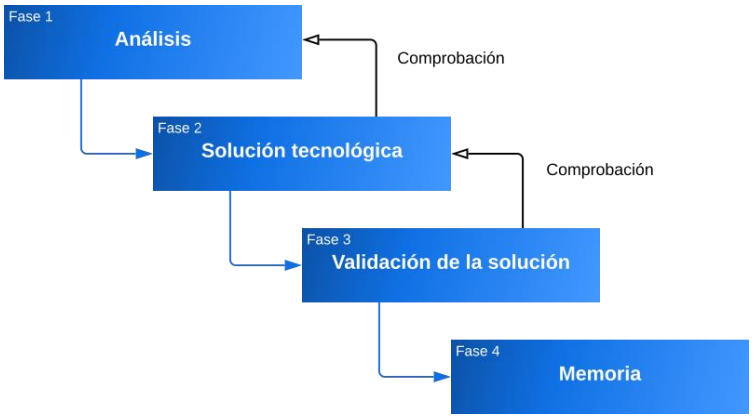


Figura 2. Proceso de desarrollo del proyecto.

1.4 Medios hardware y software

Este proyecto ha sido desarrollado principalmente en un equipo de sobremesa que cuenta con las siguientes especificaciones:

- Sistema Operativo: Windows 11 Home
- Procesador: Intel Core i7-14700KF 3.40 GHz
- Memoria: 32 GB DDR5
- Almacenamiento: 1 TB SSD + 1 TB SSD

También se ha utilizado como apoyo un segundo equipo, este portátil, como equipo de respaldo ante situaciones de indisponibilidad del equipo de sobremesa. Este segundo equipo cuenta con estas características:

- Sistema Operativo: Windows 10 Home
- Procesador: Intel Core i7-8750H 2.20 GHz
- Memoria: 16 GB DDR4
- Almacenamiento: 256 GB SSD + 1 TB HDD

Ambos equipos han utilizado exactamente el mismo software para el desarrollo de la herramienta, y es el que se muestra a continuación:

- JDK 17
- Eclipse IDE (for Java and DSL Developers) 2024-03
- Remix IDE
- INNoVaServ Toolkit PCN Business Model Plug-In
- SmaC DSL Plug-In
- Microsoft Word para Microsoft 365
- Microsoft PowerPoint para Microsoft 365

1.5 Estructura de la memoria

Esta memoria está estructurada de manera que pueda seguirse adecuadamente el proceso de construcción del proyecto. De esta forma, este documento cuenta con:

- El presente **Capítulo 1** a modo de introducción en el que se sitúa el contexto que da origen a la idea de este proyecto junto el objetivo a lograr, la metodología seguida para conseguirlo y las herramientas utilizadas.
- El **Capítulo 2** que sirve para presentar las bases tecnológicas sobre las que se construye la solución, detallando cada una de las tecnologías o herramientas utilizadas para su elaboración.
- El **Capítulo 3** con la solución tecnológica, donde se detallan los conocimientos previos adquiridos y el desarrollo de cada uno de los dos puentes tecnológicos construidos para conectar PCN y los smart contracts.
- El **Capítulo 4** que recoge varios casos de estudio para validar las herramientas implementadas, poniendo en evidencia su correcto funcionamiento.

  •Las conclusiones expuestas en el **Capítulo 5**, resultantes tras el proceso de desarrollo de las herramientas y su posterior validación en los dos capítulos previos y que dan por concluido este Trabajo de Fin de Grado. Además, se detallan posibles mejoras que incorporar en forma de trabajos futuros.

Capítulo 2 - Estado del Arte

Una vez situado un primer contexto, se da paso a la presentación de las tecnologías y herramientas que hacen de cimientos para este proyecto, las **bases** sobre las que se construye la solución tecnológica. Este capítulo pretende no solo introducir estas bases, sino poner en conocimiento el **estado** y **alcance** que tienen todas ellas en la **actualidad** y justificar el motivo de su selección como parte del proyecto frente a otras opciones.

También se ofrece una visión actual en la que se analizan **otras herramientas** o soluciones existentes que cubren parcial o totalmente la funcionalidad que soporta este proyecto.

2.1 Smart contracts y Solidity

Vivimos en un mundo cada vez más impulsado por la tecnología en el que no dejan de producirse cambios que transforman por completo la forma de realizar algunas actividades a medida que surgen avances tecnológicos. En el caso de los contratos, acuerdos elaborados por lo general de manera escrita que recogen una serie de condiciones a cumplir entre varias partes, tradicionalmente han necesitado una tercera parte de confianza encargada de verificar el cumplimiento del contrato. Pero eso cambió con la llegada de la tecnología **blockchain**, un registro o base de datos de transacciones compartida, descentralizada e inmutable [6], que posteriormente evolucionaría en un nuevo enfoque conocido como blockchain 2.0 con la llegada de un concepto alternativo a los contratos convencionales, los smart contracts.

2.1.1 Smart contracts

La web de Ethereum define los smart contracts como "**programas** informáticos almacenados en la **cadena de bloques** que siguen la lógica «si ocurre esto, entonces se produce aquello» y garantizan ejecutarse siguiendo las **reglas definidas** por su código, que no se puede cambiar una vez creado" [7].

En realidad, los smart contracts no surgieron directamente tras la creación de blockchain. Su origen se remonta a 1994, cuando el criptógrafo Nick Szabo ya hizo una aproximación al concepto de smart contract con unos objetivos que coinciden con los objetivos alcanzados por el concepto actual; "Necesitamos un protocolo que garantice que el producto se entregará si se realiza el pago, y viceversa" [8]. El protocolo buscado por Nick no llegaría hasta 2009 con la creación de la red blockchain de Bitcoin aunque fue la posterior creación de la red Ethereum en 2014 lo que impulsó la idea para llegar a ser lo que tenemos hoy en día [9].

El funcionamiento de los smart contract es el siguiente: una vez escrito el código del contrato, se aloja en la red blockchain a través de una transacción que contiene dicho código. Una vez alojado en la red, es desplegado automáticamente y se puede interactuar con él realizando transacciones a la dirección del contrato incluyendo los datos necesarios para ejecutar alguna de sus funciones.

Los principales motivos para la adopción de un smart contracts son los **beneficios** que presentan frente a un contrato convencional gracias a la tecnología blockchain, entre los cuales se encuentran:

- **Seguridad:** Los registros de blockchain están encriptados, distribuidos y son inmutables. Esto hace que los smart contracts alojados en la red sean seguros frente a ciberataques.
- **Confianza:** La distribución de los registros entre todos los miembros de la red blockchain y su inmutabilidad hacen que las transacciones en los smart contracts sean transparentes y fiables.
- **Ahorro:** En relación con el punto anterior, al no precisar de intermediarios ni terceros involucrados en el smart contract, se ahorra dinero y tiempo.
- **Eficiencia:** Los smart contracts no dejan de ser programas informáticos automatizados al cumplir ciertas condiciones, lo que supone una ganancia en eficiencia y velocidad [2].

Respecto a las **aplicaciones** y **usos** que pueden hacerse de los smart contracts, son muchos los sectores en los que tienen cabida. Educación, sanidad, finanzas, logística, inmobiliaria y empresa son varios de estos sectores. Veamos ejemplos para cada uno de ellos:

- **Educación:** Emisión de certificados académicos. La Universidad de Nicosia fue la primera en usar smart contracts para este fin [10].
- **Sanidad:** Incremento de agilidad en la gestión de datos de pacientes y acceso seguro y eficaz a registros sanitarios por parte de proveedores autorizados.
- **Finanzas:** Automatización de procesos financieros, pagos y seguimientos de reembolsos.
- **Logística:** Seguimiento de cadenas de suministro para garantizar la autenticidad de los productos y gestión eficaz de inventario.
- **Inmobiliaria:** Automatización en el cobro de alquileres y tanto garantía como eficacia en la transferencia de propiedades.
- **Empresa:** Automatización de procesos de contratación en Recursos Humanos y mejora de eficacia en el proceso de pago de nóminas y otros beneficios en empleados. Cabe destacar el proyecto Hyperledger Fabric, una plataforma blockchain de código abierto dedicada a uso empresarial en la que se pueden ejecutar smart contracts definidos por otros usuarios [11] [12].

Puesto que los smart contracts son en esencia scripts, estos han de escribirse en un **lenguaje de programación** para que pueda ser compilado y ejecutado como sucede con cualquier otro programa informático. Existen múltiples lenguajes que permiten programar smart contracts. Entre los lenguajes más utilizados se encuentran Solidity, Vyper, Rust y Haskell [13], pero a la hora de escoger uno hay que tener en cuenta **factores** como:

- La **red de blockchain** en la que se pretende alojar el smart contract, ya que cada red tiene lenguajes exclusivos o favoritos frente a otros [13] y el lenguaje utilizado deber ser compatible y estar preparado para funcionar con dicha red o plataforma.

- Las **características propias del lenguaje**, que van a permitir potenciar aspectos como la seguridad o el rendimiento a la hora de desarrollar smart contracts con dicho lenguaje.

Ejemplificando la importancia de estos factores a la hora de escoger lenguaje, Solidity y Vyper se ejecutan sobre EVM (Ethereum Virtual Machine), la máquina virtual que ejecuta los smart contracts en la red de Ethereum, lo que hace a estos lenguajes utilizables en Ethereum y en otras redes compatibles con EVM como lo son Polygon, Avalanche o BNB Chain [14]; sin embargo, Solidity destaca por su flexibilidad a la hora de desarrollar y Vyper se centra en la seguridad [13]. Por otro lado, Rust es utilizado preferiblemente en redes como Solana y Near Protocol enfocándose en el rendimiento, y Haskell en la red de Cardano ofreciendo más precisión y claridad [13].

2.1.2 Solidity

De entre todos los lenguajes para smart contracts, Solidity es el lenguaje **predominante** que destaca por varios motivos. En la sección 2.1.1 se ha comentado que se ejecuta sobre EVM, también su sintaxis está basada en lenguajes populares como C++, JavaScript y Python [15], lo que hace que sea sencillo de aprender y de utilizar por la gran mayoría de desarrolladores y además se considera el lenguaje estándar de Ethereum, la plataforma blockchain con mayor popularidad dentro del mundo de las aplicaciones descentralizadas [13]. Por estos motivos, el lenguaje Solidity se ha seleccionado para la realización de este trabajo.

Solidity está inspirado en varios lenguajes orientados a objetos y esto se puede apreciar en la estructura y componentes de un smart contract escrito en Solidity. Para empezar, cumple varios de los principios fundamentales de la programación orientada a objetos al contar con soporte de herencia o encapsulación al poder controlar el acceso a variables o funciones. Con Solidity, los contratos en sí actúan como **clases**. Estas clases poseen su propio **constructor** y pueden contener **funciones** actuando como cláusulas del contrato que contienen la lógica que seguirá el smart contract junto a **variables** o **estructuras** tanto locales como globales que dictan la lógica del contrato dentro de las funciones. A esta estructura familiar de los lenguajes orientados a objetos se le suman elementos propios de Solidity:

- **Modificadores:** Bloques de código predefinido que se pueden aplicar a las funciones para dotarlas de funcionalidades extra como si de un patrón decorador se tratara.
- **Eventos:** Código que actúa a modo de log (registro) para transmitir información sobre algún suceso durante la ejecución.
- **Tipos de datos:** Están presentes la mayoría de los tipos de datos vistos en la mayoría de los lenguajes de programación, tanto simples como estructurados, pero Solidity cuenta con tipos de dato destinado exclusivamente a almacenar direcciones de Ethereum, el tipo *address* junto con la posibilidad de enviar y/o recibir divisa entre variables de dicho tipo mediante funciones propias del lenguaje.

Así se termina de definir, en rasgos generales, lo que sería la estructura básica de un smart contract en Solidity.

2.2 Notaciones de negocio y proceso

Para que un negocio funcione de manera óptima es importante que los procesos que lo componen estén bien definidos. Una manera de conseguir esto es haciendo uso de las notaciones de negocio, herramientas que ayudan al modelado de los **procesos**, **flujos** de trabajo e **interacciones** que tienen lugar en un negocio, permitiendo **representarlos** y entenderlos de forma clara y visual. A través del uso de notaciones de negocio se pueden analizar los procesos de diversos tipos de modelo de negocio y en base a ello, tomar decisiones que potencien estos negocios [16].

Existen varios tipos de notaciones de negocio, algunas similares y otras muy diferentes entre sí según los elementos que las compongan. A continuación se introducen dos de ellas, PCN y e^{3}value.

2.2.1 Process Chain Network (PCN)

Process Chain Network (PCN) es una notación de negocio propuesta por Scott E. Sampson que "nos ayudará a visualizar procesos de negocio, redes y cuestiones de gestión" [17]. Nace con el objetivo de, una vez representado un proceso con esta notación, poder analizarlo e identificar mejoras que poder aplicar para optimizar dicho proceso haciéndolo más sólido, eficiente, ágil, factible y comprensible [18].

A continuación, en la Figura 3, se muestra un ejemplo de diagrama PCN sobre el servicio que ofrece un minorista de mobiliario en donde se pueden observar varios de los elementos de los que se compone la notación y cómo se relacionan entre sí.

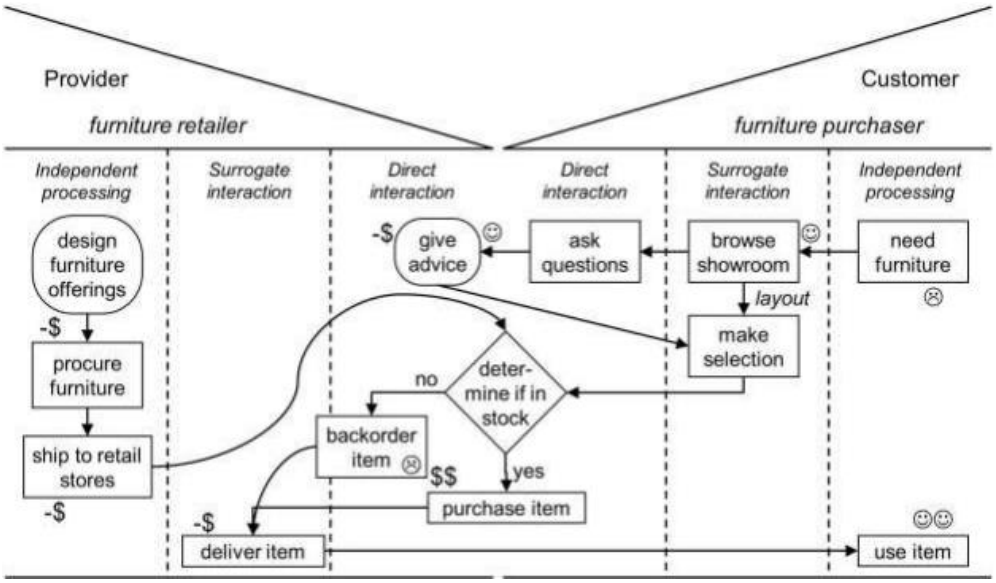


Figura 3. Diagrama PCN para un servicio de venta de mobiliario. Obtenido de [18].

PCN parte del concepto de **cadena de proceso**, una secuencia de **pasos** con un propósito específico como puede ser servir comida para un restaurante o, en el caso de la Figura 3, vender mobiliario a un cliente. Los pasos pueden ser de varios tipos y, en función de ello, son representados por cajas con distintas formas que contienen la acción parte del proceso (p.ej. enviar un producto). Para definir el flujo de las cadenas de proceso, los pasos están conectados por **dependencias**, flechas que conectan los pasos entre sí que marcan dicho flujo. Como complemento de los pasos existen las **etiquetas**, pequeños símbolos alrededor de las cajas que pueden usarse para expresar una ganancia o pérdida de valor vinculada en un paso en cuestión

(p.ej. ganancia económica muy positiva al vender un producto). Hasta ahora se han definido los elementos que forman las cadenas de procesos sin atender a los participantes que intervienen en el proceso; estos participantes son las **entidades**, representadas por estructuras cuya forma recuerdan a una casa. Y en relación con las entidades existe el **dominio**, la parte dividida en columnas que tiene cada entidad, en donde se sitúan los pasos que realiza y controla la entidad en cuyo dominio se encuentran estos pasos [17][18].

Algo más a resaltar sobre el dominio son las tres regiones en las que se divide:

- **Interacción directa:** Los pasos situados en esta región requieren de interacción de persona a persona con otra entidad.
- **Interacción subrogada:** Los pasos situados en esta región requieren de interacción con otra entidad pero no de persona a persona.
- **Proceso independiente:** Los pasos situados en esta región son realizados íntegramente por la propia entidad [18].

Para dejar más clara la diferencia entre las regiones del dominio, la Figura 4 pone el caso de un curso académico con un profesor y un estudiante como entidades protagonistas. En el caso del proceso independiente como acción propia y aislada, el profesor por su lado puede escribir un libro y el alumno puede leer otros libros por su cuenta. En la interacción subrogada, el profesor y el alumno interactúan de forma indirecta a través de una plataforma online. Por último, la interacción directa, compartida por profesor y alumno, reúne a ambos en una clase para revisar en persona el material académico.

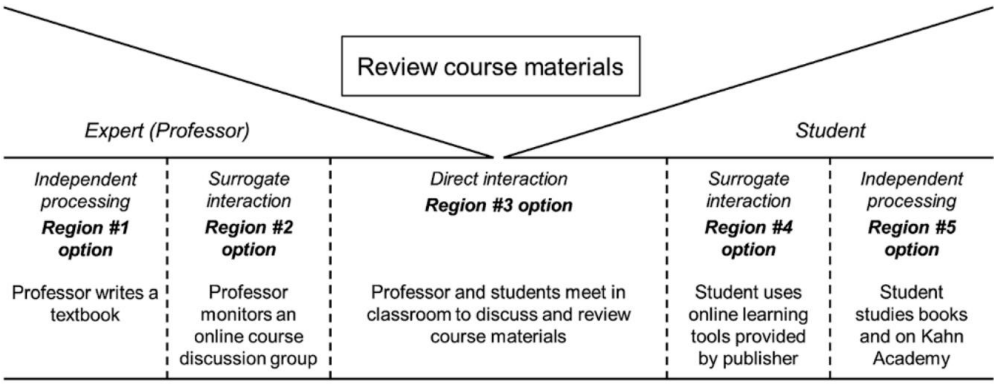


Figura 4. Regiones del dominio en un curso académico. Obtenido de [19].

Todos elementos vistos hasta ahora se recogen en la Figura 5 en forma de tabla, en la que además se pueden observar los diferentes tipos de pasos, dependencias y etiquetas que existen en la notación, además de la simbología que utiliza para representar cada elemento en sus diagramas y su significado.

Por último, se habla de usos de PCN en la actualidad. Como concluye Steve Pearce en su artículo (2020), puede ser aplicado en dispositivos portátiles, aplicaciones, Internet of Things, Inteligencia Artificial y robótica. También expone dos casos de uso: en un banco digital británico donde se hace uso de PCN para optimizar el proceso de apertura de una cuenta a través de una aplicación móvil y en educación donde se aplica PCN en un proceso educativo ayudando a preparar el programa formativo de un máster sobre negocio digital [20].

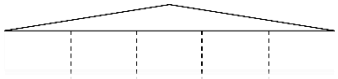
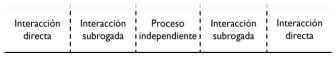
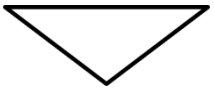
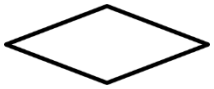
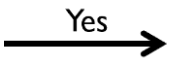
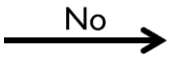
Elemento PCN		Símbolo PCN	Descripción
Entidad			Organismo o individuo independiente que interviene en un proceso.
Dominio			Conjunto de pasos del proceso realizados y controlados por una entidad. Divido en tres regiones: Interacción directa, interacción subrogada y proceso independiente.
Paso	Estandar		Paso que define la realización de una acción.
	Subcontrata		Paso estándar realizado por una fuente externa a la entidad.
	Innovación		Paso estándar que supone una innovación en el proceso.
	Acción y espera		Paso que define la realización de una acción seguida de un periodo de espera.
	Espera		Paso que define la realización de un periodo de espera.
	Decisión		Paso que define una toma de decisión condicional.
Dependencia	Estandar		Dependencia que define la precedencia del paso que la origina con el paso al que apunta.
	Con retardo		Dependencia estándar que define un periodo de espera entre los pasos que conecta.
	Positiva		Dependencia estándar que define una decisión positiva tomada en un paso de decisión y que conduce a otro paso.
	Negativa		Dependencia estándar que define una decisión negativa tomada en un paso de decisión y que conduce a otro paso.
Etiqueta	Compensación monetaria		Etiqueta que identifica un paso que supone un beneficio monetario para la entidad involucrada.
	Coste monetario		Etiqueta que identifica un paso que supone un coste monetario para la entidad involucrada.
	Beneficio no monetario		Etiqueta que identifica un paso que supone un beneficio no monetario para la entidad involucrada.
	Coste no monetario		Etiqueta que identifica un paso que supone un coste no monetario para la entidad involucrada.
	Textual	Text	Etiqueta de texto que señala algún aspecto adicional a destacar en un paso.

Figura 5. Tabla 1: Elementos de la notación PCN.

2.2.2 Notación e³value

E³value es una notación de negocio propuesta por Jaap Gordijn y Hans Akkermans centrada en los intercambios de valor para definir cómo tienen lugar estos en una red en la que intervienen varias partes. Esta aproximación nace de la idea de dejar atrás los modelos representados de manera informal o con representaciones gráficas tan específicas que no son generalizables ni claras para, en su lugar, poder representar modelos de negocio de manera gráfica y ligera sin dejar atrás la calidad y la cantidad [21][22].

La Figura 6 muestra un ejemplo de diagrama e³value que define el servicio que ofrece una biblioteca. En él, se pueden observar la mayoría de los elementos que componen esta notación y que se exponen a continuación.

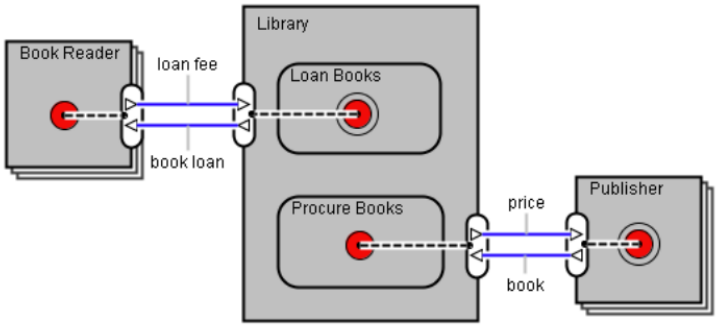


Figura 6. Diagrama e³value para un servicio de biblioteca. Obtenido de [23].

Ya se ha explicado que e³value mantiene el foco en los intercambios de valor. Estos intercambios se dan entre los llamados **actores** (p.ej. una biblioteca o un restaurante), representados por un rectángulo si se trata de una entidad individual, o **segmento de mercado** si se trata de un grupo de actores que comparten una misma función (p.ej. los lectores de una biblioteca), representado por varios rectángulos apilados. Cada actor o segmento de mercado puede contener las llamadas **actividades de valor** con las que poder detallar la forma de ofrecer y recibir valor, representadas con rectángulos con las esquinas redondeadas. Para realizar los intercambios de valor, cada actor o segmento de mercado dispone de **interfaces de valor**, elementos con forma ovalada donde tienen lugar el envío y recepción de valor. Cada interfaz de valor, a su vez, está formada por **puertos de valor**, puntas de flecha cuya dirección indica si dichos puertos envían o reciben valor. Por otra parte, están los **estímulos iniciales**, eventos que desencadenan una **ruta de escenario,** un camino en forma de línea discontinua que pueden conectar interfaces de valor u otras rutas de escenario mediante **dependencias AND** u **OR**, y que finalizará con un **estímulo final**; ambos estímulos representados con círculos rojos con un punto diferenciando el estímulo final con un círculo externo extra. Por último, los **intercambios de valor** en sí son representados con una línea azul que conecta los puertos de dos actores o segmentos de mercado [21][22].

Los elementos explicados se agrupan a continuación en la tabla de la Figura 7.

Elemento e^{3}value		Símbolo e^{3}value	Descripción
Actor			Organismo o individuo independiente que interviene en una red.
Segmento de mercado			Grupo de actores con la misma función dentro de una red.
Actividad de valor			Actividad realizada por un actor que produce un beneficio de valor.
Interfaz de valor			Elemento donde tiene lugar un intercambio de valor. Contiene puertos de valor.
Puerto de valor			Componente de una interfaz de valor que define puntos de entrada y salida de valor.
Intercambio de valor			Elemento que conecta dos puertos de valor para definir un intercambio de valor.
Estímulo	Inicial		Evento por parte de una entidad que supone el inicio de una ruta de escenario.
	Final		Elemento que indica el final de una ruta de escenario.
Ruta de escenario			Elemento que une estímulos con interfaces de valor. Pueden bifurcarse y unirse mediante dependencias AND u OR.
Dependencia	AND		Dependencia que permite bifurcar o unir dos rutas de escenario siguiendo una lógica de conjunción.
	OR		Dependencia que permite bifurcar o unir dos rutas de escenario siguiendo una lógica de disyunción.

Figura 7. Tabla 2: Elementos de la notación e^{3}value.

Dejando de lado los elementos, hay que destacar algunos beneficios que trae el uso de e^{3}value:

- Mejora en la **comunicación** y **toma de decisiones** de las partes interesadas sobre los aspectos principales de un modelo de negocio.
- Permite una **comprensión** clara de las operaciones y requisitos de los modelos de negocio por medio del análisis de los escenarios [22].

Para finalizar, se menciona un caso de estudio recogido por Gordijn, Petit y Wieringa (2006) que muestra el uso que puede darse de la notación. Este caso trata sobre una editorial de periódicos independientes, a los que facilita medios de impresión, logística y personal, entre otros. Gordijn ejerció de consultor para esta editorial y trazó un modelo e^{3}value inicial según los objetivos fijados. Dado que posteriormente entraron nuevos valores, objetivos y actores, se revisó el modelo y se adaptó a las nuevas necesidades. Esta iteración en el proceso de modelado haciendo uso de e^{3}value permitió perfeccionar el negocio y adaptarlo a nuevos alcances [24].

2.3 Model Driven Engineering (MDE)

La ingeniería de software siempre ha arrastrado problemas en el desarrollo de software a lo largo de su historia, problemas ya señalados por Dijkstra en 1972 en relación con tiempo, coste y alcance. Con la finalidad de solventar parte de estos problemas nace la Ingeniería Dirigida por Modelos (en inglés Model Driven Engineering bajo las siglas MDE), una aproximación para el desarrollo de software a través del uso de modelos [4].

MDE introduce concepto principal el **modelo**, una simplificación de un sistema en el mundo real centrada en aspectos relevantes de dicho sistema en un contexto concreto. Para definir un modelo se utiliza un **lenguaje de modelado** que a su vez está definido por el conceto de **metamodelo**, el modelo que define la estructura y sintaxis de un lenguaje de modelado [25][26]. La Figura 8 muestra las relaciones entre estos conceptos principales.

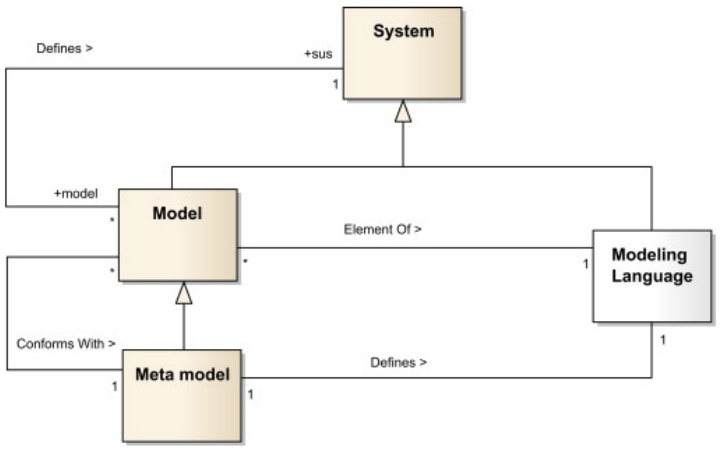


Figura 8. Relación entre los conceptos principales de MDE. Obtenido de [25].

A estos conceptos principales se les suma la **transformación entre modelos** como parte del desarrollo software bajo MDE. Existen dos tipos de transformaciones:

- **Model-To-Model (M2M):** Transformación de un modelo en otro modelo.
- **Model-To-Text (M2T):** Generación automática de código a partir de un modelo [25].

Esta aproximación trae consigo beneficios tales como un elevado nivel de **abstracción** de los lenguajes a través del uso de modelos para acercarlos a dominios concretos y la **automatización** y **reducción de errores** en el desarrollo de software gracias a la generación automática de código a partir de modelos [4].

Dentro del paradigma MDE existen diversas tecnologías y herramientas que utilizan los conceptos ya explicados, pudiendo dividirse generalmente en las siguientes categorías acompañadas con ejemplos:

- **Tecnologías base** como la proporcionada por el entorno de desarrollo Eclipse con su proyecto EMF (Eclipse Modeling Framework) para construir soluciones basadas en MDE.
- **Entornos de desarrollo de lenguajes** como Xtext para el modelado de DSLs textuales como SmaC.
- **Entornos de gestión de modelos** como la herramienta INNoVaServ, un entorno dirigido al diseño de servicios [27].

2.3.1 Eclipse Modeling Framework (EMF)

Eclipse Modeling Framework es un proyecto de Eclipse Foundation basado en MDE que ofrece un entorno de modelado para construir aplicaciones y herramientas basadas en modelos de datos estructurados descritos en XMI (XML Metadata Interchange) [29], un formato pensado para el intercambio de información entre modelos y herramientas [30].

Entre las características principales de este entorno se encuentran:

- La creación y manipulación de metamodelos, denominados **Ecore**, para definir modelos y ejecutar instancias del modelo definido.
- Un subentorno que incluye un conjunto de clases para construir **editores** de modelos EMF.
- Capacidad de generación de código a partir de modelos EMF, incluyendo la implementación de interfaces y clases Java del modelo, adaptadores para la edición y visualización de las partes del modelo y un editor personalizable del modelo [29].

Los usos más destacados de EMF se dan en herramientas, en su mayoría, desarrolladas por la propia Eclipse Foundation. Este es el caso de Sirius, un entorno de modelado gráfico para editar y visualizar modelos EMF, y de Acceleo, una tecnología basada en plantillas para construir generadores de código a partir de modelos EMF [31][32].

2.3.2 Xtext y Xtend

Xtext es un **entorno** para la creación de Lenguajes de Dominio Específico (DSLs) desarrollado por itemis y mantenido por Eclipse. Brinda lo necesario para **facilitar el desarrollo** de DSLs, sobre todo pensando en el paradigma MDE, donde es necesario el desarrollar estos lenguajes de forma eficaz [33].

Para sus DSLs, Xtext ofrece **validación** de semántica para evitar errores, **coloreado** de sintaxis como suele verse con lenguajes convencionales, **personalización** del formato mediante el manejo de tokens, **vista** jerárquica de los lenguajes, "quick fixes" para **solucionar errores** comunes, entre otros. La tarea más importante a la hora de crear un DSL es definir su **gramática**, el conjunto de **reglas** y **expresiones** que definen la escritura del DSL una vez puesto en marcha, para lo que Xtext trae un lenguaje específico. Una vez definida la gramática de un DSL, Xtext genera de forma automática los componentes necesarios para poner en uso el lenguaje [34].

A pesar de estar vinculado a Eclipse, Xtext no se limita solo al entorno de desarrollo Eclipse, sino que se puede ejecutar en entornos más populares como Visual Studio Code y otros entornos basados en cloud [33]. Cabe resaltar su uso en cualquier ámbito dada la flexibilidad de uso de los DSLs para integrarlos en prácticamente cualquier dominio, como es el caso de los smart contracts con el DSL SmaC.

Por otra parte está Xtend, un **lenguaje** de programación basado en el lenguaje Java y autoconsiderado un dialecto de este. Xtend busca mejorar muchos de los aspectos de Java **ampliando** su funcionalidad para ser más conciso, expresivo y legible [35].

A pesar de las mejoras que plantea respecto a Java, Xtend garantiza una **interoperabilidad** plena con el lenguaje; además, el compilador se encarga, por defecto y de forma automática, de **generar código** Java traducido a partir de código Xtend.

Sin embargo, sobre todas las características de Xtend, cabe resaltar una en concreto por su utilidad e importancia en este proyecto, las **plantillas**. Dentro de todas las expresiones soportadas por Xtend existen las plantillas, expresiones definidas como si fueran cadenas de texto pero con triples comillas simples (''') que además de texto, saltos de línea y tabulaciones, pueden contener variables, estructuras condicionales y bucles que se evalúan y sustituyen por el texto correspondiente [35]. Esta característica hace que Xtend sea una opción muy atractiva bajo el paradigma MDE para construir generadores de código como el que forma parte de la solución expuesta en el siguiente capítulo.

2.3.3 INNoVaServ

INNoVaServ es un **entorno de modelado** desarrollado por Francisco Javier Pérez para el diseño de modelos de negocio y servicios. Para ello, la herramienta se compone de un conjunto de **DSLs gráficos** creados sobre el entorno EMF y mediante Sirius y Eclipse GMF (Graphical Modeling Framework), un entorno que ofrece componentes para la creación de editores gráficos basados en EMF, que definen **diferentes notaciones de negocio** (PCN, e^{3}value, BPMN, Service Blueprint y Canvas) en forma de **modelo** y un panel de control que permite realizar **transformaciones** parciales entre estos modelos mediante puentes tecnológicos implementados entre algunas de las notaciones (ver Figura 9) [36].

Figura 9. Puentes tecnológicos soportados por INNoVaServ. Obtenido de [36].

También incluye reglas de **validación** para los modelos creados de cada notación soportada

El hecho de que INNoVaServ esté construido sobre una distribución del entorno de desarrollo Eclipse y mantenga un diseño modular con el reparto de sus componentes en distintos **plug-ins** hace que sea fácil de integrar completa o parcialmente con otros proyectos. Así sucede en el caso de este proyecto haciendo uso del plug-in que contiene el modelo PCN que incluye INNoVaServ.

2.3.4 SmaC

SmaC es un **DSL textual** creado con Xtext y desarrollado por Cristian Gómez Macías para el modelado de smart contracts escritos en lenguaje Solidity.

La gramática definida para SmaC está basada, como es lógico, en la sintaxis del lenguaje Solidity. Como sucede con los DSL creados con Xtext, cuenta con **validación** de su semántica siguiendo unas reglas de validación establecidas para evitar errores durante su escritura, **coloreado** de la sintaxis para un aspecto visual más claro y trabajado y soporte de "quick fixes" para facilitar la **resolución de errores** de sintaxis, entre otras funciones [37].

Este acercamiento entre modelo y smart contract hace posible poder tratar un smart contract como una clase para el consumo y manejo de su información para diversos usos como transformaciones en otros modelos y también para facilitar la escritura de smart contracts.

Una vez presentados todos los aspectos clave, se habla de alternativas actuales que comparten funcionalidad con la expuesta en este trabajo.

Anteriormente se ha presentado INNoVaServ, una herramienta que incorpora una serie de puentes tecnológicos entre varias notaciones de negocio seleccionadas (entre las que se incluye PCN). También estudia la posibilidad de generar smart contracts, tal y como se propone en el artículo [38]. El objetivo de establecer un puente entre PCN y smart contract sería un hecho posible de lograr con INNoVaServ, pero trae consigo un inconveniente. Con INNoVaServ, transformar un modelo PCN en un smart contract de la manera más directa pasa por hacer una serie de transformaciones consecutivas por dos vías posibles:

- Transformar primero el modelo PCN en un modelo SBP (Service Blueprint), una notación de negocio enfocada en la interacción cliente-proveedor de un servicio, y transformar este segundo modelo SBP obtenido en un tercer modelo e^{3}value para, finalmente, transformar de forma directa el modelo e^{3}value obtenido en smart contract.
- Transformar el modelo PCN en un modelo BPMN (Business Process Modeling Notation) y transformar este modelo BPMN directamente en smart contract [38].

Estos procedimientos no serían los más convenientes para la integridad del modelo PCN inicial, puesto que en la transformación de PCN a SBP se puede perder aproximadamente un 2,29% de información, una cantidad muy asumible, sin embargo en la transformación de SBP a e^{3}value se puede perder el valor aproximado de un 96,77% de información, un valor que sí supone una pérdida de gran peso. En la transformación de PCN a BPMN se perdería en torno a un 32,54% de información, que, aunque no es un porcentaje alto, podría tener impacto significativo según el contexto y el valor de la información perdida [39]. Esto no sería un problema contando con un mecanismo directo de transformación entre PCN y smart contract como el que aborda este trabajo y que se expone en el siguiente capítulo. En la Figura 10 se puede observar una comparativa de las posibles soluciones, las dos pertenecientes a INNoVaServ resaltando las pérdidas de información mencionadas y la perteneciente a la solución tecnológica propuesta.

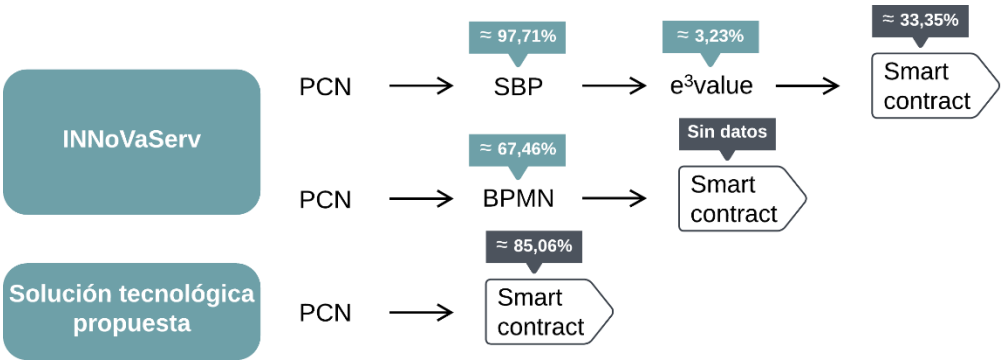


Figura 10. Comparativa de soluciones para la transformación de PCN a smart contract.

También, en lo que respecta a notaciones de negocio, se ha presentado e^{3}value, una notación con transformación directa a smart contract y viceversa por parte de INNoVaServ, y PCN, la notación elegida para conectar con los smart contract. Esto puede llevar a plantear si merece la pena escoger PCN para convertir en smart contract frente a e^{3}value; la respuesta es sí. Esto se justifica comparando ambas notaciones: la Figura 11 muestra una comparativa visual entre ambas notaciones para la representación del servicio que ofrece la empresa de reparto de comida a domicilio Deliveroo; un diagrama PCN en el lado

izquierdo y un diagrama e^{3}value en el lado derecho. La equivalencia más clara se da entre las entidades de PCN y los actores de e^{3}value, resaltando en color azul el actor o entidad principal, Deliveroo, en cada uno de ellos. Por otro lado, están los elementos en los que se enfoca cada notación, los intercambios de valor en e^{3}value y las cadenas de proceso en PCN. La notación e^{3}value se limita a representar estos intercambios de valor de forma sencilla y sin detalle mientras que en PCN la información de estos intercambios está contenida en el diagrama en forma de procesos más detallados. Para ejemplificarlo, se resalta el intercambio de valor entre Deliveroo y los clientes por el que un cliente hace un pedido (resaltado en color verde) y a cambio recibe su pedido en su casa (resaltado en color amarillo). Este intercambio es representado en el diagrama e^{3}value sin más detalle que una breve descripción que define los valores del intercambio, sin embargo, en el diagrama PCN a través de sus cadenas de procesos se pueden ver detallados estos intercambios mediante los diferentes tipos de pasos y dependencias que permiten detallar a nivel funcional el proceso de intercambio (toma de decisiones, esperas, etc.). Esto sin contar las capas de detalle adicionales que ofrece PCN como el nivel de interacción que se da en cada parte de los procesos o las etiquetas que aportan más detalles si cabe.

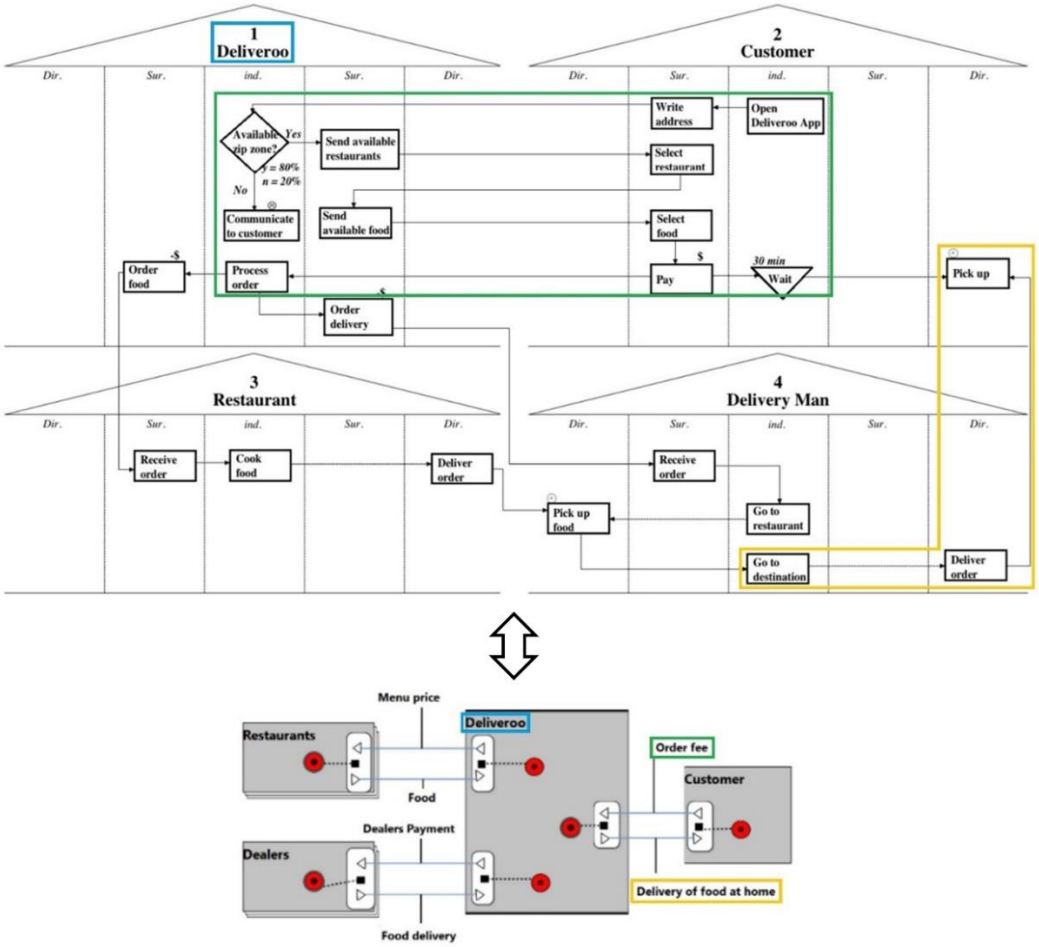


Figura 11. Comparativa visual entre PCN (arriba) y e^{3}value (abajo).

En definitiva, el hecho de que PCN ofrezca más información y niveles de detalle que e^{3}value se traduce en una mejor definición de servicios en forma de smart contract con mayor detalle funcional si se hace a partir de un diagrama PCN.

Capítulo 3 - Solución Tecnológica

En este apartado se presentan tanto los puentes tecnológicos propuestos como solución para lograr conectar la notación de negocio PCN y los smart contracts. Para ello, en primer lugar, se detalla la **arquitectura** de la solución mostrando su estructura junto a los elementos que la forman, además de otros aspectos como los medios y herramientas utilizados. Después, se hace un acercamiento más en detalle del proceso de construcción desde el **análisis previo** a la especificación de la solución hasta el **diseño** y **desarrollo** de las herramientas creadas. Estas herramientas están alojadas en un **repositorio** dedicado [40].

3.1 Arquitectura de la solución

La solución propuesta pasa por diseñar y desarrollar un programa en Java con apoyo de Xtend que permita convertir un diagrama PCN en smart contract. Desde ambos extremos del puente se cuenta con un punto de partida: por un lado se dispone del plug-in de INNoVaServ con el que poder modelar y manipular diagramas PCN como objetos, y por otro, el DSL SmaC con el que poder escribir smart contracts en código Solidity y, de igual forma, manipularlos a través de clases. Estos hechos sitúan en las bases del proyecto a EMF y Xtext, y en consecuencia, a Eclipse como IDE indispensable en el desarrollo.

De esta manera, y dado que el puente tecnológico ha de ser bidireccional para formar una unión sólida entre ambas partes, la solución se divide en dos herramientas independientes pero complementarias, una encargada del proceso de transformación de un modelo PCN en smart contract escrito en lenguaje Solidity y otra encargada del proceso inverso, transformar un smart contract en un modelo PCN.

Ambas herramientas comparten una estructura de clases similar. Esta estructura, que recuerda al patrón Mediador, separa la funcionalidad en distintas clases aisladas entre sí, cada una dedicada a un grupo de tareas concreto, que son coordinadas por clases mediadoras que gestionan su actuación. Esto dota de modularidad a los dos programas y facilita el mantenimiento del código, además de mejorar su legibilidad y la escalabilidad ante la posible incorporación de nueva funcionalidad. Así, los grupos de tareas que existen en ambos programas quedan denominados de la siguiente manera:

- **Gestor de ficheros:** Clase Java que contiene la funcionalidad con relación a la creación del directorio y/o el fichero que necesitan generarse.
- **Gestor de interfaces:** Clase Java que contiene la funcionalidad encargada de las interfaces con las que puede interactuar el usuario.
- **Gestor de plantillas:** Clase Xtend que contiene la funcionalidad relacionada con las plantillas que permiten construir el código generado.

Estos grupos, como se explicaba anteriormente, son orquestados por clases Java mediadoras, un **manejador del generador** como mediador principal y un **generador de código** como mediador secundario

al servicio del mediador principal. Así pues, y tal y como se puede ver en la Figura 12, se define de forma simplificada la arquitectura que sigue cada una de las direcciones del puente.

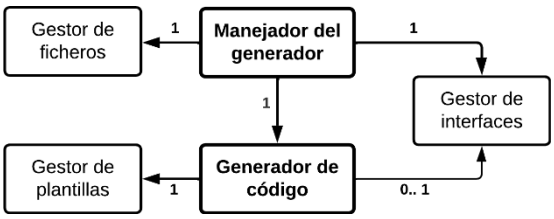


Figura 12. Arquitectura simplificada de cada puente tecnológico.

Estas herramientas, a su vez, son construidas como componentes integrados como parte de los proyectos plug-in de los que se sirven para su funcionamiento, el modelo PCN de INNoVaServ en el caso del puente de PCN a smart contract y el DSL SmaC en el caso del puente de smart contract a PCN. Esto es así debido a que el uso de las herramientas sin acceso a los modelos carecería de sentido y presuponer su instalación a parte podría traer problemas de estabilidad al tratar de usar las herramientas.

Por último y no menos importante, queda comentar acerca del punto de entrada a las herramientas. Por cada una de ellas, se habilita un punto de entrada en forma de botón seleccionable desde el menú contextual dentro de la pestaña del explorador de paquetes para cada tipo de fichero (*.sce* o *.pcn*). Según el fichero modelo seleccionado, el botón que aparezca será el de una herramienta u otra.

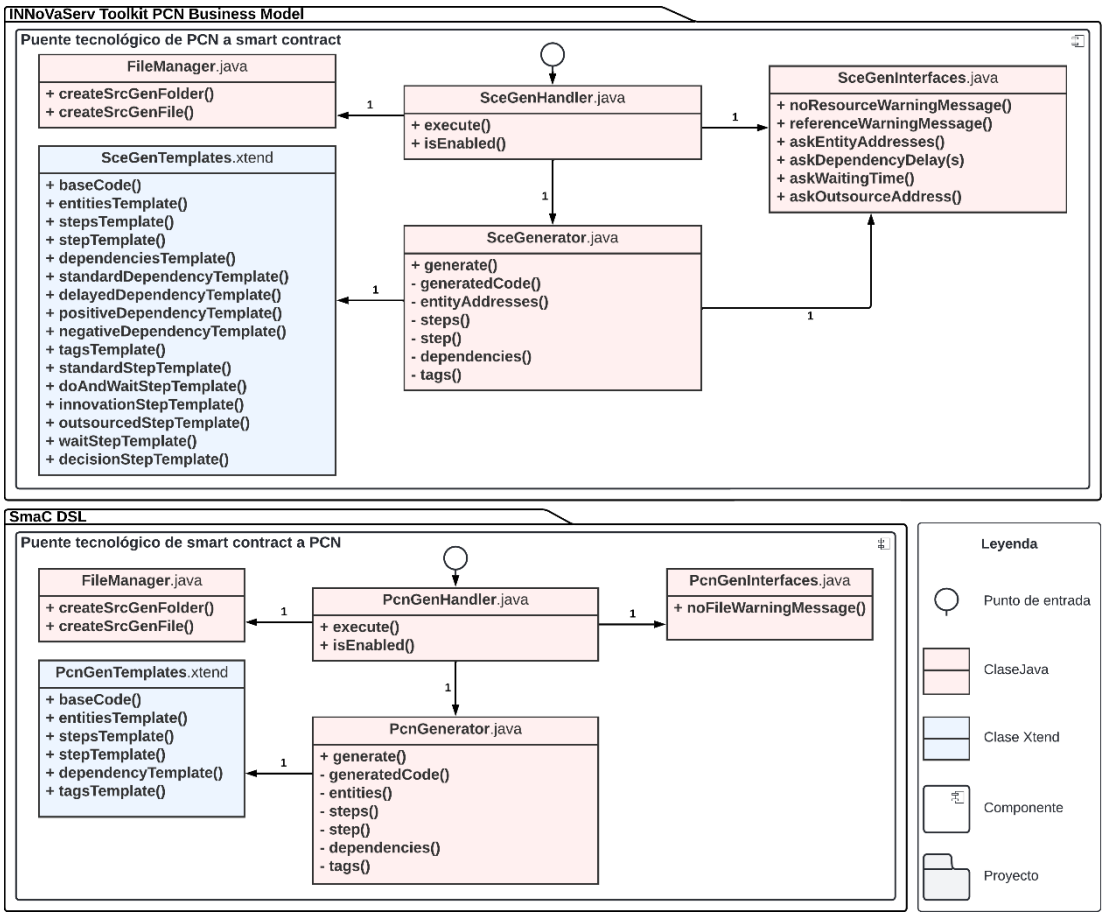


Figura 13. Arquitectura completa del puente tecnológico.

En la Figura 13 se puede observar el diagrama de clases que ubica y relaciona de forma detallada todas las partes mencionadas que componen la arquitectura completa de la solución tecnológica.

3.2 Análisis previo

Antes de comenzar con el desarrollo de los puentes dentro de Eclipse, ha sido necesario hacer una investigación en profundidad sobre la notación PCN y Solidity, el lenguaje usado para escribir smart contracts. Estas tareas abarcan lo que sería el marco teórico que da vida al proyecto, de ahí la importancia de indagar y obtener toda la información posible para descubrir puntos o características compatibles que permitan relacionar ambos campos para el desempeño de las tareas siguientes.

Para el caso de la notación PCN, su estudio ha abarcado los orígenes de esta herramienta, sus aplicaciones pero sobre todo, haciendo un especial hincapié, los elementos que lo forman y su funcionamiento. Esta investigación se da por finalizada al comprender y diferenciar los conceptos dentro de la notación como los distintos tipos de pasos, dependencias y etiquetas, entre otros conceptos; lo que da la capacidad de crear y entender diagramas PCN.

En el caso de Solidity, se procede de forma similar. La diferencia es que, en contraste con PCN, donde el punto de partida sin experiencia previa con notaciones de negocio intensificaba la tarea, el punto de partida con Solidity ha sido más aclimatado gracias a la experiencia previa en lenguajes orientados a objetos. Echar un vistazo con detenimiento a la documentación oficial para conocer los elementos característicos del lenguaje, su patrón de construcción y otros formalismos junto a los numerosos ejemplos que acompañan a los apartados han sido suficiente para pasar a la práctica. Se ha utilizado el IDE online Remix, que no solo es accesible al estar en formato web sino que cuenta con entornos de simulación en los que poder desplegar y probar los smart contracts trabajados con acceso a una colección de direcciones y límite de gas. El uso de Remix ha permitido perfeccionar lo aprendido y probar los límites del lenguaje.

Una vez investigadas las herramientas protagonistas, el siguiente objetivo pasa por establecer una relación de correspondencia entre sus elementos. La forma más sencilla de hacerlo es partiendo de los elementos de PCN y pensando cómo estos podrían representarse en código Solidity haciendo posibles las características funcionales que define la notación. El caso que se hace más evidente es el de las **entidades** de PCN, que al ser elementos que representan a una persona u organización, se relacionan directamente con direcciones de Ethereum con acceso al contrato alojadas en objetos de tipo ***address***. Cada **paso** PCN puede representarse con una **función** dentro del smart contract, ya que cada paso define una acción o conjunto de acciones que pueden traducirse como instrucciones dentro de una función que representa dicho paso. El **dominio** y sus **regiones** en PCN pueden representarse como **modificadores** aplicados a las funciones para dar exclusividad de ejecución a las direcciones (entidades) elegidas; esto además en combinación con el uso de **eventos** en dichas funciones para hacer eco de la información. Las **dependencias** en PCN, de la misma forma que sucede con el concepto de dominio, también se pueden representar como **modificadores** aplicados a las funciones (pasos) para determinar su orden de ejecución, tal y como hacen con los pasos de un diagrama PCN. Las **etiquetas** de PCN, dada su función meramente informativa, puede representarse con **eventos** que transmitan y reflejen la misma información. Por último, están los **recursos** que participan dentro de un proceso de negocio representado en un diagrama PCN, que se pueden corresponder con las **variables** utilizadas para construir la lógica funcional de cada paso dentro de las funciones del smart contract.

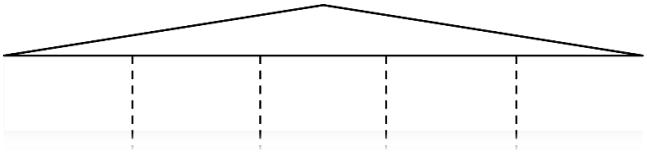
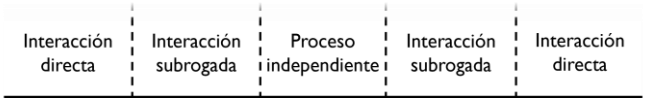
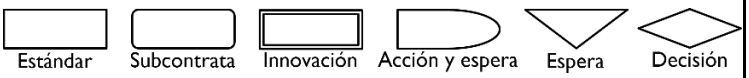
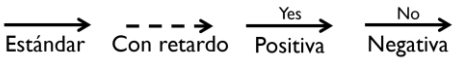
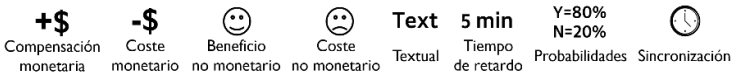
Elemento Solidity	Elemento PCN	Símbolo gráfico PCN
Objeto *address*	Entidad	
Modificadores Eventos	Dominio	
Funciones	Pasos	
Modificadores	Dependencias	
Eventos	Etiquetas	
Variables	Recursos	

Figura 14. Tabla 3: Correspondencias entre smart contracts y diagramas PCN.

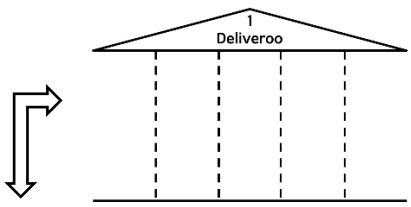
Como resultado, el ejercicio de asociación entre los elementos de PCN y Solidity puede verse sintetizado en la tabla de la Figura 14.

3.3 Formato de un smart contract basado en PCN

Una vez definidas las relaciones entre los conceptos de Solidity y los elementos de PCN, hay que argumentar dichas relaciones poniéndolas en práctica para corroborar su robustez. Para ello, se idea un **formato** muy específico que establece cómo representar cada elemento PCN en Solidity siguiendo las correspondencias establecidas para conseguir dar vida a un diagrama PCN en forma de smart contract. A continuación se explica cómo representar cada elemento:

- **Entidad:** Se utiliza una **estructura** para representar a cada entidad como un objeto formado por una dirección Ethereum vinculada a la entidad, y el orden y nombre de la entidad dentro del diagrama PCN. Luego se usa un **diccionario** que indexa el objeto de cada entidad dándole un identificador con el que referirse a cada entidad.

```
struct Entity{
    uint order;
    string name;
    address addr;
}

mapping(uint => Entity) private entity;
```



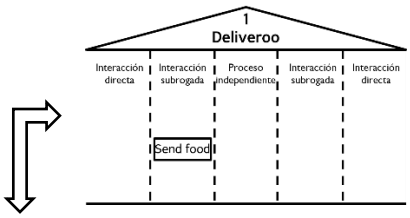

```
constructor() {
    entity[0] = Entity(1, "Deliveroo", 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4);
}
```

Figura 15. Representación de una entidad en un smart contract.

Dentro del constructor del smart contract se declara el **objeto** de cada una de las entidades y se asigna a una **entrada** vacía del diccionario, como se señala la Figura 15.

- **Dominio:** Se utiliza un **modificador** aplicado a la función de cada paso tras el modificador de acceso para permitir su ejecución solo a la entidad especificada por medio de su identificador y un **evento** informativo que aporta cierta información sobre cada paso, entre la que se incluye su región del dominio y su identificador dentro de esa región.

```
modifier domain(uint _entity) {
    require(msg.sender == entity[_entity].addr,
    "This step isn't part of your domain");
    _;
}

event StepInfo(string _class, string _domainRegion,
               int _domainRegionId, string _action);
```




```
function step6() public domain(0)
standardDependency(0, 6)
{
    emit StepInfo("PCNStandardStep", "leftSurrogateInteractions", 0, "Send food");

    //
    // Functional logic
    //

    emit StepCompleted(msg.sender, "Done: Send food");
    completedStep[6] = true;
}
```

Figura 16. Representación del dominio y sus regiones en un smart contract.

Como se comentaba, en la función de cada paso se aplica el modificador del dominio especificando el **identificador** de la entidad a cuyo dominio pertenece dicho paso. Por otro lado, como primera instrucción dentro del cuerpo de la función, se lanza el evento informativo en el que se introduce la **región** del dominio (*leftDirectInteractions*, *leftSurrogateInteractions*, *centerIndependentInteractionsActivities*, *rightSurrogateInteractions* o *rightDirectInteractions*) y el **identificador** del paso dentro de la región de la entidad, como se ve en la Figura 16.

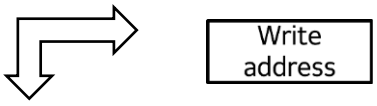
- **Pasos:** Se usa el **evento** informativo visto anteriormente con la información que quedaba por explicar, el tipo de paso y la acción de dicho paso, y también se usa un segundo **evento** de finalización que anuncia que un paso ha sido completado, indicando la acción y la entidad responsable de completar el paso.

  Cada paso es etiquetado con un identificador numérico general a partir de 1 y representado con una función cuyo nombre sigue el formato "*step*[identificador general]". Su cabecera utiliza el modificador de acceso *public* para tener acceso al paso. En su interior se lanza el evento informativo con el **tipo** de paso que es y la **acción** de dicho paso, un espacio dedicado a la implementación de la **lógica funcional** que desempeñe la acción del paso, y el evento de finalización como señal de **fin** del paso.

  Estos elementos son comunes en todos los pasos, independientemente del tipo de paso que sea; sin embargo, la disposición dentro del cuerpo de la función varía según el tipo de paso:

- **Estándar:** Al tratarse de un paso sin ninguna característica especial, cumple la estructura básica explicada anteriormente y señalada en la Figura 17, formada por el evento informativo con el **tipo** de paso (*PCNStandardStep*) y su **acción**, seguido del espacio para la **lógica funcional** del paso y después el evento de **finalización**.

```
event StepInfo(string _class, string _domainRegion,
               int _domainRegionId, string _action);

event StepCompleted(address _entity, string _name);
```




```
function step9() public domain(1)
standardDependency(0, 9)
{
    emit StepInfo("PCNStandardStep", "leftSurrogateInteractions", 1, "Write address");

    //
    // Functional logic
    //

    emit StepCompleted(msg.sender, "Done: Write address");
    completedStep[9] = true;
}
```

Figura 17. Representación de un paso Estándar en un smart contract.

- **Subcontrata:** Este paso se representa de forma similar a un paso Estandar. Se especifica su **tipo** (*PCNOutsourcedStep*) pero también se declara la **dirección** de la entidad externa que actúa como subcontrata después de la lógica funcional y antes del evento de finalización del paso, en donde se sitúa a la **subcontrata** como autora del paso. Esto se puede ver en la Figura 18.

```
event StepInfo(string _class, string _domainRegion,
               int _domainRegionId, string _action);

event StepCompleted(address _entity, string _name);
```


```
function step21() public domain(2)
standardDependency(0, 21)
{
    emit StepInfo("PCNOutsourcedStep", "leftDirectInteractions", 0, "Give advice");

    //
    // Functional logic
    //

    address outsource = 0xdD870fA1b7C4700F2BD7f44238821C26f7392148;
    emit StepCompleted(outsource, "Done: Give advice");
    completedStep[21] = true;
}
```

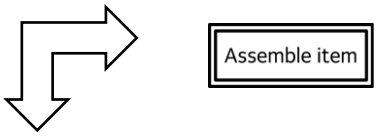
Figura 18. Representación de un paso Subcontrata en un smart contract.

- **Innovación:** En este paso, al igual que en el paso Subcontrata, no existe gran diferencia con un paso Estándar. En este caso se utiliza también un **evento** para representar el cambio o novedad que supone la realización del paso.

  En la función de un paso Innovación se especifica su **tipo** (*PCNInnovationStep*) y se lanza el evento que representa la **innovación** después de la lógica funcional y

antes del evento de finalización del paso como se observa en la Figura 19, finalizando el paso de la forma vista hasta ahora.

```
event StepInfo(string _class, string _domainRegion,
               int _domainRegionId, string _action);

event StepCompleted(address _entity, string _name);

event Innovation();
```

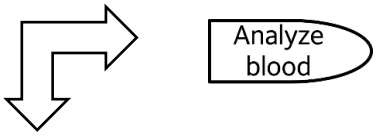



```
function step1() public domain(0)
standardDependency(0, 1)
{
    emit StepInfo("PCNInnovationStep", "rightDirectInteractions", 0, "Assemble item");

    //
    // Functional logic
    //

    emit Innovation();
    emit StepCompleted(msg.sender, "Done: Assemble item");
    completedStep[1] = true;
}
```

Figura 19. Representación de un paso Innovación en un smart contract.

- **Acción y Espera:** Los tipos de paso que deben de implementar un mecanismo de espera como es en este caso, utilizan también un **diccionario** que guarda el instante de tiempo en el que se inicia el paso y, con ello, la espera. Cada tiempo del diccionario se asocia a una clave que coincide con el identificador general del paso al que corresponden dichos instantes de tiempo.

```
mapping(uint => uint) private waitInit;

event StepInfo(string _class, string _domainRegion,
               int _domainRegionId, string _action);

event StepCompleted(address _entity, string _name);
```



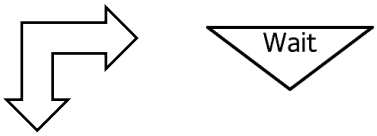

```
function step2() public domain(0)
standardDependency(0, 2)
{
    emit StepInfo("PCNDoAndWaitStep", "rightSurrogateInteractions", 0, "Analyze blood");

    if (waitInit[2]==0) {

        //
        // Functional logic
        //

        waitInit[2] = block.timestamp;
    } else {
        require(block.timestamp >= waitInit[2] + 300,
        "Wait not finished yet");

        emit StepCompleted(msg.sender, "Done: Analyze blood");
        completedStep[2] = true;
    }
}
```

Figura 20. Representación de un paso Acción y Espera en un smart contract.

De esta forma y como se ve en la Figura 20, dentro de la función se especifica el **tipo** (*PCNDoAndWaitStep*) y se implementa una estructura condicional que en una primera ejecución pasa por la lógica funcional del paso y guarda el **tiempo**, para en posteriores ejecuciones comprobar si ha pasado el tiempo de espera determinado (en segundos) dentro de una **cláusula** *require* que dictamina si se finaliza el paso de forma incompleta o de forma completa mediante el evento de finalización del paso como viene sucediendo en los tipos de paso anteriores.

- **Espera:** Al tratarse de un paso que, al igual que se vio con el paso Acción y Espera, debe implementar un mecanismo de espera, cumple con prácticamente la misma estructura con los pasos Acción y Espera. La diferencia radica, como se observa en la Figura 21, en el cambio de **tipo** (*PCNWaitStep*) y que la función se construye **sin lógica funcional** porque el paso solo precisa espera sin una acción previa.

```
mapping(uint => uint) private waitInit;

event StepInfo(string _class, string _domainRegion,
               int _domainRegionId, string _action);

event StepCompleted(address _entity, string _name);
```



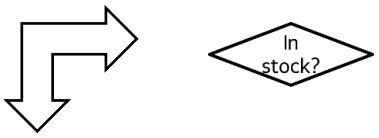

```
function step4() public domain(0)
standardDependency(0, 4)
{
    emit StepInfo("PCNWaitStep", "leftSurrogateInteractions", 1, "Wait");

    if (waitInit[4]==0) {
        waitInit[4] = block.timestamp;
    } else {
        require(block.timestamp >= waitInit[4] + 1800,
        "Wait not finished yet");

        emit StepCompleted(msg.sender, "Done: Wait");
        completedStep[4] = true;
    }
}
```

Figura 21. Representación de un paso Espera en un smart contract.

- **Decisión:** Se utiliza un **diccionario** para guardar la decisión tomada en cada paso Decisión. La entrada de cada decisión se asocia con un número identificativo que coincide con el identificador de la función del paso en el que se toma dicha decisión.

  En el cuerpo de la función del paso, además de especificar el **tipo** (*PCNDecisionStep*), se utiliza una **variable booleana** pensada para ser utilizada como parte de la lógica funcional del paso y que posteriormente el valor de dicha variable determine qué **decisión**, afirmativa o negativa, guardar en el diccionario pensado para ello. Después se termina con el evento de finalización del paso al igual que en pasos anteriores y como se aclara en la Figura 22.

```
mapping(uint => bool) private decision;

event StepInfo(string _class, string _domainRegion,
               int _domainRegionId, string _action);

event StepCompleted(address _entity, string _name);
```




```
function step3() public domain(1)
standardDependency(0, 3)
{
    emit StepInfo("PCNDecisionStep", "centerIndependentInteractionsActivities", 0, "In stock?");

    bool selection;

    //
    // Functional logic using 'selection' variable to decide between
    // the affirmative dependency (variable set to 'true')
    // or the negative Dependency (variable set to 'false')
    //

    if (!selection) {
        decision[3] = false;
    } else {
        decision[3] = true;
    }

    emit StepCompleted(msg.sender, "Done: In stock?");
    completedStep[3] = true;
}
```

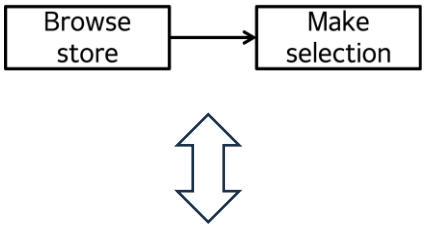
Figura 22. Representación de un paso Decisión en un smart contract.

- **Dependencias:** Se utiliza un **modificador** aplicado tras el modificador de dominio en las funciones de los pasos a los que se llega con dicha dependencia e incluso de los pasos sin ninguna dependencia entrante. Como funcionalidad base, este modificador recibe el identificador general del paso que **origina** la dependencia y el identificador general del paso que **recibe** la dependencia que coincide con el del propio paso en el que se aplica. Cabe resaltar la posibilidad de tener pasos con más de una dependencia entrante y, por tanto, tener más de un modificador de dependencia en una sola función de paso. También se usa un **diccionario** como estructura de estado situado justo después del evento de finalización de cada paso que sirve para hacer que se cumpla el flujo de ejecución que dictan las dependencias.

  Según las características propias de cada dependencia, el modificador mencionado varía ligeramente, por lo que hay un tipo de modificador por cada tipo de dependencia:

  - **Estándar:** Ya que se trata de una dependencia con la funcionalidad base, cumple con las características mencionadas anteriormente. La función de un paso que precede a esta dependencia incorpora el **modificador** correspondiente (*standardDependency*). Como **caso especial**, los pasos que no dependen de ningún otro representan esta característica con una dependencia estándar en la que el origen es un paso raíz (**paso 0**) completado en el constructor del smart contract.

```
mapping(uint => bool) private completedStep;

modifier standardDependency(uint _source, uint _target) {
    require(completedStep[_source],
    "Dependencies not complete for this step");

    require(!completedStep[_target],
    "This step has already been done");
    _;
}
```




```
constructor() {
    completedStep[0] = true;
}
```

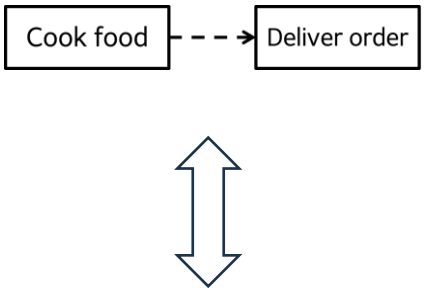
```
function step26() public domain(3)
standardDependency(0, 26)
{
    emit StepInfo("PCNStandardStep",
    "leftSurrogateInteractions", 0,
    "Browse store");

    //
    // Functional logic
    //

    emit StepCompleted(msg.sender,
    "Done: Browse store");
    completedStep[26] = true;
}
```

```
function step15() public domain(3)
standardDependency(26, 15)
{
    emit StepInfo("PCNStandardStep",
    "leftDirectInteractions", 0,
    "Make selection");

    //
    // Functional logic
    //

    emit StepCompleted(msg.sender,
    "Done: Make selection");
    completedStep[15] = true;
}
```

Figura 23. Representación de una dependencia Estándar en un smart contract.

- **Con retardo:** Utiliza un **modificador** similar al de una dependencia Estándar**,** pero incluyendo un mecanismo de espera antes de permitir ejecutar el paso de destino. Este tipo de dependencia implica el uso de valores de espera, por lo que se usa un **diccionario** que guarda el instante de tiempo en el que se termina el paso que origina esta dependencia. Cada entrada del diccionario es asociada con una clave que coincide con el identificador general del paso que da pie a la dependencia, como se venía explicando. Por otra parte, el modificador podrá comprobar si los tiempos guardados han sido cumplidos para permitir la ejecución del paso que precede. Esto se puede entender mejor observando la Figura 24.

```
mapping(uint => bool) private completedStep;

mapping(uint => uint) private delayedDependencyInit;

modifier delayedDependency(uint _source, uint _target,
                           uint _duration) {
    require(completedStep[_source],
    "Dependencies not complete for this step");

    require(!completedStep[_target],
    "This step has already been done");

    require(block.timestamp >=
    delayedDependencyInit[_source] + duration,
    "You must wait a while before executing this step");
    _;
}
```



```
constructor() {
    completedStep[0] = true;
}
```

```
function step7() public domain(2)
standardDependency(0, 7)
{
    emit StepInfo("PCNStandardStep",
    "rightSurrogateInteractions", 0,
    "Cook food");

    //
    // Functional logic
    //

    emit StepCompleted(msg.sender,
    "Done: Cook food");
    completedStep[7] = true;

    delayedDependencyInit[7] = block.timestamp;
}
```

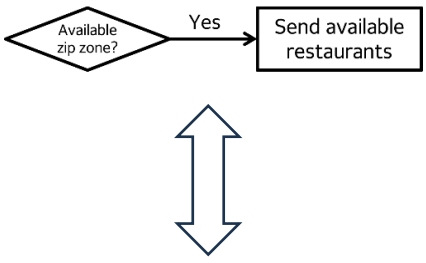
```
function step8() public domain(2)
delayedDependency(7, 8, 900)
{
    emit StepInfo("PCNStandardStep",
    "rightDirectInteractions", 0,
    "Deliver order");

    //
    // Functional logic
    //

    emit StepCompleted(msg.sender,
    "Done: Deliver order");
    completedStep[8] = true;
}
```

Figura 24. Representación de una dependencia Con retardo en un smart contract.

La función del paso que origina este tipo de dependencia incorpora detrás de la estructura de estado una instrucción que guarda el **tiempo** (en segundos) en el que finaliza el paso origen para en una posterior ejecución del paso de destino poder comprobar si se ha completado el tiempo de espera. Por otro lado, el paso destino utiliza el **modificador** correspondiente (*delayedDependency*) que incluye el tiempo de espera que establece la dependencia antes de permitir la ejecución de ese paso.

- **Positiva:** Su existencia implica que el paso predecesor es de tipo Decisión, lo que también implica haber guardado de antemano la decisión tomada en dicho paso. Situado este contexto, el **modificador** de una dependencia Positiva (*positiveDependency*) funciona como el de una dependencia Estándar pero incorporando una cláusula *require* adicional que verifica si esa ha sido la opción elegida en el paso Decisión anterior, permitiendo o no la ejecución del paso al que se aplica. La Figura 25 ilustra cómo se aplica en la práctica.

```
mapping(uint => bool) private completedStep;

modifier positiveDependency(uint _source, uint _target) {
    require(completedStep[_source],
    "Dependencies not complete for this step");

    require(!completedStep[_target],
    "This step has already been done");

    require(decision[_source]==true,
    "The decision taken is negative");
    _;
}
```



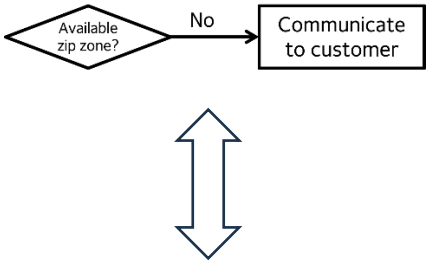

```
constructor() {
    completedStep[0] = true;
}
```

```
function step11() public domain(2)
positiveDependency(10, 11)
{
    emit StepInfo("PCNStandardStep", "leftSurrogateInteractions", 1, "Send available restaurants");

    //
    // Functional logic
    //

    emit StepCompleted(msg.sender, "Done: Send available restaurants");
    completedStep[11] = true;
}
```

Figura 25. Representación de una dependencia Positiva en un smart contract.

- **Negativa:** La forma de operar es exactamente igual al de la dependencia Positiva recién revisada, como se observa en la Figura 26.

```
mapping(uint => bool) private completedStep;

modifier negativeDependency(uint _source, uint _target) {
    require(completedStep[_source],
    "Dependencies not complete for this step");

    require(!completedStep[_target],
    "This step has already been done");

    require(decision[_source]==false,
    "The decision taken is positive");
    _;
}
```




```
constructor() {
    completedStep[0] = true;
}
```

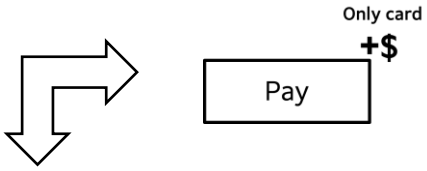
```
function step12() public domain(2)
negativeDependency(10, 12)
{
    emit StepInfo("PCNStandardStep", "leftDirectInteractions", 1, "Communicate to customer");

    //
    // Functional logic
    //

    emit StepCompleted(msg.sender, "Done: Communicate to customer");
    completedStep[12] = true;
}
```

Figura 26. Representación de una dependencia Negativa en un smart contract.

La única diferencia es que este **modificador** (*negativeDependency*) invierte la lógica encargada de determinar que el paso en el que se aplica es la decisión tomada, siendo en este caso la decisión negativa. La única diferencia es que este modificador (*negativeDependency*) invierte la lógica encargada de determinar que el paso en el que se aplica es la decisión tomada, siendo en este caso la decisión negativa.

- **Etiquetas:** Simplemente se usa una serie de **eventos**, uno para cada tipo de etiqueta.

```
event MonetaryCompensationsTag();
event MonetaryCostsTag();
event NonmonetaryBenefitsTag();
event NonmonetaryCostsTag();
event TextTag(string _text);
event DelayTimeTag(string _time);
event ProbabilitiesTag(string _yes, string _no);
event SynchronizationTag();
```




```
function step11() public domain(1)
standardDependency(0, 11)
{
    emit StepInfo("PCNStandardStep", "leftSurrogateInteractions", 0, "Pay");
    emit TextTag("Only card");
    emit MonetaryCompensationsTag();

    //
    // Functional logic
    //

    emit StepCompleted(msg.sender, "Done: Pay");
    completedStep[11] = true;
}
```

Figura 27. Representación de etiquetas en un smart contract.

La Figura 27 explica cómo simplemente se lanza el **evento** o eventos de las etiquetas que tenga el paso en el cuerpo de la función de dicho paso, situándolos justo después del evento informativo que encabeza el cuerpo de cada una de las funciones.

3.4 Puente tecnológico de PCN a smart contract

Una vez diseñado un formato sólido de smart contract basado en PCN, se da inicio al desarrollo de una primera herramienta para la transformación de modelos PCN en smart contracts que sigan las reglas de dicho formato. Como se explica en el apartado 3.1, esta herramienta se construye como parte del plug-in de INNoVaServ que contiene el modelo PCN. Por ello, se crea en su interior un paquete dedicado a albergar la funcionalidad que abarca el puente como un componente aislado. Cabe recordar la arquitectura de este componente vista anteriormente en la Figura 12.

El primer punto pasa por la implementación del **manejador del generador** (*SceGenHandler.java*). Pero para ello, antes se implementa un nuevo botón en Eclipse. Este botón se incorpora como extensión del plug-in situándose en el menú contextual en el explorador de paquetes del IDE y sirviendo como **punto de entrada** a la clase del manejador para que sea este quien obtenga el recurso del modelo original y dé inicio al proceso de conversión de modelos PCN a smart contract.

Como parte del proceso y antes de poner en marcha la generación de código, el manejador gestiona la creación, en caso de necesitarlo, del directorio de destino para el smart contract generado y el fichero vacío que se convertirá en el smart contract final una vez sea generado el código Solidity y volcado en dicho fichero posteriormente. Esto lo hace mediante el uso de otra clase java implementada que agrupa la funcionalidad relacionada con ficheros, apodada como **gestor de ficheros** (*FileManager.java*).

Una vez creado el directorio y ficheros necesarios, el manejador pone en marcha el generador de código, alojado en otra clase. Llegado este punto, ya habrán actuado varios mecanismos de validación aplicados al modelo PCN seleccionado que serán detallados más adelante en un apartado dedicado.

Por otro lado y dado que un smart contract basado en PCN precisa de datos que no puede aportar el diagrama PCN en el que se basa, se considera necesaria la existencia de un conjunto de interfaces guiadas que obtengan dichos datos a través del usuario. Entre estos datos se encuentran las direcciones Ethereum de las entidades implicadas y los tiempos que se desee establecer en pasos con espera o dependencias con retardo. Por ello, se engloba toda esta funcionalidad y se implementa una clase java que actúa como **gestor de interfaces** (*SceGenInterfaces.java*) al servicio del generador de código para que este pueda lanzarlas cuando sea necesario. Además de interfaces relacionadas con la obteción de datos del diagrama, también abarca interfaces relacionadas con los mecanismos de validación ya mencionados.

En otra línea, el uso de plantillas es un aspecto fundamental a destacar antes de hablar del proceso de generación. Siguiendo los ejemplos vistos en el apartado 3.3, se diseñan varias plantillas estáticas de código Solidity personalizables mediante el uso de *placeholders* y se implementa una clase Xtend (*SceGenTemplates.xtend*) con métodos dedicados a la obtención y manejo de estas plantillas, apodado como **gestor de plantillas**. Esta clase queda al control exclusivo del generador detallado a continuación.

El núcleo funcional principal de la herramienta es el **generador de código** (*SceGenerator.java*), por ello se pretende hacer un especial énfasis en el proceso de generación de código y en la labor de coordinación que desempeña esta clase junto con otros módulos del programa para llevarlo a cabo. Este proceso sigue un procedimiento ilustrado en la Figura 28 por el que los diferentes métodos del generador de código por una parte pueden obtener (si aplica) datos necesarios del usuario llamando a la interfaz adecuada en cada momento por medio de la clase gestora de interfaces, y por otra solicitan plantillas con fragmentos de código a la clase gestora de plantillas. De esta forma, y en combinación con los recursos proporcionados por el propio modelo PCN accedidos por el generador a través de las clases del modelo que permiten el acceso a sus elementos y atributos, se llega a componer el código del smart contract final.

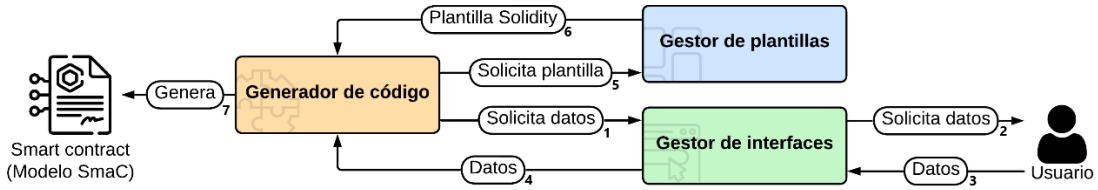


Figura 28. Proceso sencillo de generación de código del modelo SmaC.

Sin embargo, esto va más allá: algunos de estos fragmentos de código generados como plantillas, a su vez son solicitados por otros métodos del generador que generan con ellos fragmentos cada vez más grandes de código. Es decir, la generación de código sigue en realidad un proceso **recursivo** en el que el código final se va generando desde la obtención de las últimas plantillas hasta la primera plantilla que contiene el código básico. El motivo de seguir este procedimiento recursivo es porque la propia estructura de un smart contract basado en PCN se puede ver como un árbol cuyos nodos padres contienen a sus nodos hijos, o visto de otra forma, como una matrioska de código conteniendo a la vez más código. Siendo así, se parte del código base que se correspondería con un diagrama PCN vacío, que puede contener a su vez un espacio para las entidades y otro para el conjunto total de pasos; el conjunto de pasos, por su parte puede estar formado por pasos individuales; y cada paso, al mismo tiempo puede contener un conjunto de dependencias y etiquetas. Con este enfoque, el proceso recursivo de la generación de código es el expuesto en la Figura 29, donde se ve cómo operan los métodos (recuerde Figura 13) de las distintas clases entre sí.

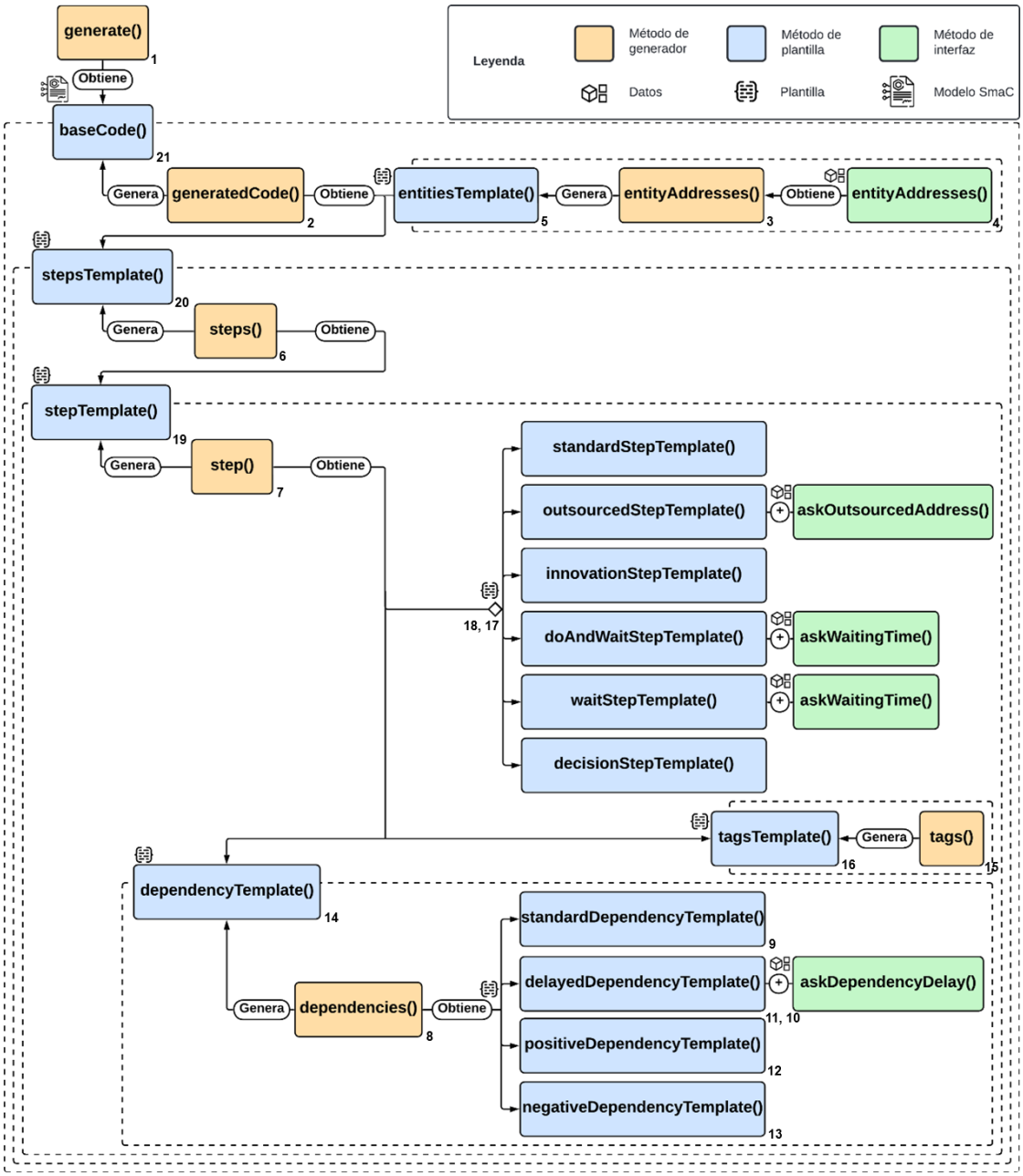


Figura 29. Proceso recursivo de generación de código del modelo SmaC.

3.5 Puente tecnológico de smart contract a PCN

Tras la construcción de un primer puente capaz de convertir modelos PCN en smart contracts, se desarrolla la segunda herramienta complementaria para la transformación de smart contracts en modelos PCN. De manera análoga al desarrollo del primer puente en el apartado anterior, la herramienta se construye dentro de un paquete dedicado como parte del plug-in que incorpora el DSL textual SmaC para la programación de smart contracts en Solidity.

Se comenta de antemano que, como ya se vio en el apartado 3.1, la arquitectura de esta segunda herramienta es muy similar a la primera, compartiendo a grandes rasgos el mismo método. Las diferencias se dan en aspectos que giran en torno al empleo de un enfoque distinto centrado en deconstruir la primera herramienta intercambiando los puntos de partida y de llegada para obtener esta vez el resultado inverso.

Se incorpora un botón en Eclipse como extensión del plug-in. Este botón se ubica, al igual que el de la primera herramienta, en el menú contextual del explorador de paquetes del IDE funcionando como **punto de entrada** al manejador. Este **manejador** se implementa en una clase java (*PcnGenHandler.java*) que obtiene el recurso del smart contract como modelo SmaC y se realizan las labores relacionadas con ficheros.

Al igual que en la primera herramienta, antes de la generación se ponen a punto el directorio en el que se guardará el modelo PCN generado y fichero vacío en el que volcar posteriormente el contenido generado. Para ello se usa una clase java **gestora de ficheros** (*FileManager.java*) utilizada por el manejador con exactamente la misma funcionalidad que la utilizada en la primera herramienta para cubrir este aspecto.

Un asunto a diferenciar con su herramienta complementaria es la implementación de un mecanismo de validación que actúa durante todo el proceso de generación en lugar de hacerlo antes. Este mecanismo se detallará más adelante en su apartado dedicado.

También es relevante mencionar que, un smart contract basado en PCN implica por definición que contenga todos los datos necesarios para poder construir el diagrama PCN en el que se basa, por lo que no es necesario el uso de ninguna interfaz con la que el usuario deba interactuar para obtener datos como parte del proceso. Dado este aspecto, se obtiene una conversión inmediata respecto a la que se obtiene con el puente de PCN a smart contract. Aun así, se implementa una clase **gestora de interfaces** (*PcnGenInterfaces.java*) pero disponiendo de una única interfaz para uso informativo.

Por otra parte, y una vez más, las plantillas adquieren protagonismo en el proceso de generación de código. A pesar de que puede parecer complejo generar un modelo PCN a partir de plantillas de texto, hay que saber que por debajo de un modelo PCN en realidad no hay más que código XMI, por lo que se dedica un periodo de tiempo al estudio y descomposición de un diagrama PCN en formato XMI. Tras esto se diseñan las plantillas estáticas de código XMI necesarias para representar cada elemento PCN por separado y se agrupan en una clase Xtend **gestora de plantillas** (*PcnGenTemplates.xtend*); esta clase queda al mando del generador para aplicar las plantillas durante la generación de código cuando sea necesario.

Como parte clave en el proceso, la implementación del **generador de código** (*PcnGenerator.java*) sigue la misma metodología vista en el apartado anterior. El proceso sencillo se da de la siguiente manera: los distintos métodos del generador obtienen plantillas de código XMI a partir de los métodos del gestor de plantillas, que en combinación con los recursos obtenidos del modelo SmaC permiten ir construyendo el código XMI completo. El código final será volcado en el fichero vacío inicial y de esta forma se obtiene el fichero XMI final, lo que es a fin de cuentas el modelo PCN generado a partir del smart contract. Este proceso se ilustra a continuación en la Figura 30.

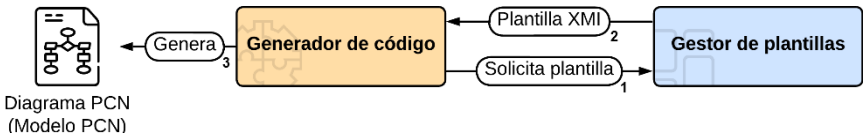


Figura 30. Proceso sencillo de generación de código del modelo PCN.

Como ya se reveló en la primera herramienta, este proceso se ve alargado de forma recursiva mediante la obtención y de plantillas con fragmentos de código aplicadas a su vez a otras plantillas hasta obtener una plantilla final con el contenido XMI íntegro. Al igual que sucede con un smart contract basado

en PCN, el código XMI de un modelo PCN también se expresa en forma de árbol y esto permite ir generando fragmentos de código desde las hojas hasta la raíz de este árbol. El proceso recursivo se muestra en la Figura 31.

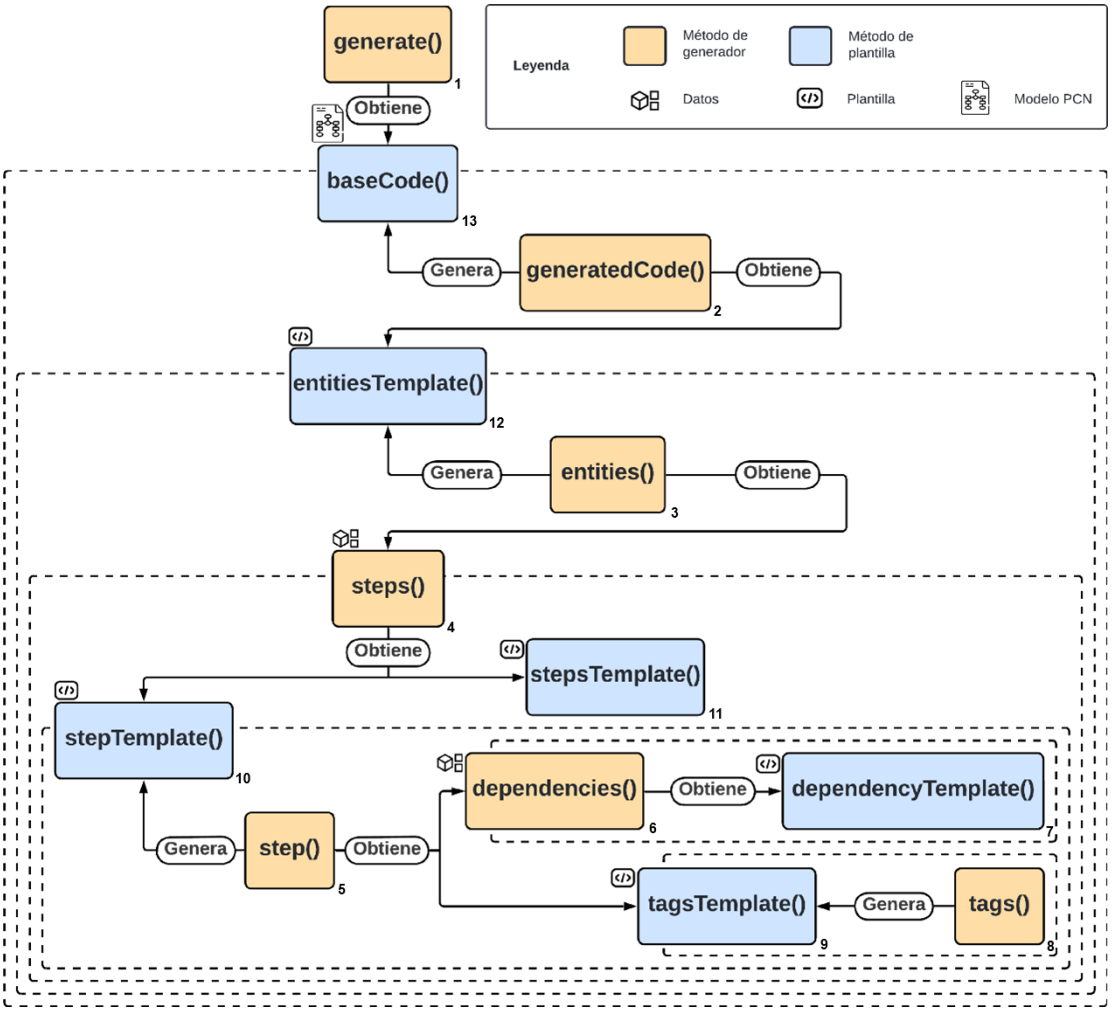


Figura 31. Proceso recursivo de generación de código del modelo PCN.

3.6 Mecanismos de validación de modelos

Con la intención de hacer las herramientas lo más sólidas y estables posible, se incorporan una serie de procesos o mecanismos de verificación con los que dar una experiencia robusta en la medida de lo viable.

Estos mecanismos se localizan en distintos puntos previos al comienzo de los procesos de transformación en sí mismos: **antes** de entrar a los manejadores y **dentro** de los manejadores de las herramientas. La razón de esto es **evitar errores** o **situaciones inesperadas** que puedan resultar en procesos abortados o resultados incorrectos. A continuación se detallan estos mecanismos:

- **Verificación de formato:** Este mecanismo tiene lugar antes de acceder a los manejadores de las herramientas y a través de las extensiones definidas en los plug-ins donde estas se alojan. Estas extensiones incorporan reglas que impiden mostrar los botones que inician las herramientas a no ser que la extensión del fichero seleccionado se corresponda con la de un modelo PCN (*.pcn*) o un modelo SmaC (*.sce*). Esta verificación supone un primer filtro donde se evita el uso de las herramientas con multitud de ficheros inconexos con el tema.

- **Verificación de contenido:** Este mecanismo sucede dentro de los manejadores de las herramientas, donde es implementado de maneras diferentes según la herramienta para comprobar si el modelo a convertir es válido:
    - **Modelo PCN:** En el manejador de la herramienta para la transformación de un modelo PCN en smart contract, la estrategia pasa por obtener el recurso del modelo. En caso de error tratando de obtener el modelo, se muestra un mensaje de error a través de una interfaz de usuario (ver Figura 32) y se termina la ejecución. Adicionalmente, en caso de obtener el recurso con éxito, se analiza dicho recurso en busca de elementos Referencia, dependencias "inalámbricas" existentes en la notación PCN cuyo sentido es puramente estético para permitir trazar una dependencia entre dos pasos distantes entre sí en la composición de un diagrama PCN. Debido a la existencia de discordancia entre la solución y la forma en que están implementadas las Referencias en la definición del metamodelo, se consideran incompatibles los modelos que hagan uso de Referencias. Por ello, en caso de encontrar una Referencia, se muestra un mensaje de error explicando la incompatibilidad (ver de nuevo Figura 32) y se pone fin a la ejecución.

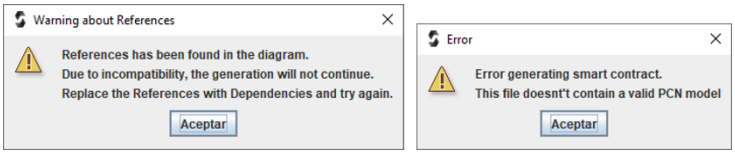


Figura 32. Errores de verificación de contenido en un modelo PCN.

- **Modelo SmaC:** Por otro lado, en el manejador de la herramienta para la transformación de un modelo SmaC (smart contract) a modelo PCN se obtiene el recurso y se procede con normalidad con la generación. Sin embargo, cualquier excepción producida en el generador será capturada abortando la generación del modelo PCN y mostrando un mensaje de error mediante una interfaz de usuario (ver Figura 33).

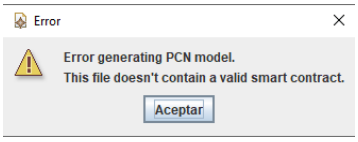


Figura 33. Error de verificación de contenido en un modelo SmaC.

Capítulo 4 - Validación de la Solución

Una vez implementada la solucion que ha hecho posible la construcción de los dos puentes, se pone a prueba frente a varios casos de estudio con la intención de verificar el correcto funcionamiento de las herramientas. Con la realización de estas pruebas se pretende, además, evaluar aspectos como la **eficiencia** y, por supuesto, el **grado de cumplimiento** de los resultados obtenidos en cada caso.

4.1 Criterios de validación

Con el fin de estandarizar el proceso de validación, se establecen una serie de criterios bajo los que evaluar los casos de estudio expuestos a lo largo de este apartado. Estos criterios buscan valorar las herramientas implementada centrándose en aspectos importantes para cualquier aplicación software:

- **Eficiencia:** Este criterio se basa en la **medición del tiempo** que se tarda en generar un modelo SmaC y un modelo PCN para un caso dado en manos de un usuario experimentado. Esta medición se da desde el momento en el que se pulsa el botón de las herramientas hasta que termina el proceso y aparece el fichero generado.

  La incógnita a resolver que surge con este aspecto es puntualizar lo que se consideraría poco eficiente o muy eficiente. Para ello se plantean tiempos de referencia basados en la realización manual de los procesos, el único método alternativo que permitiría hacer cualquiera de los procesos y su reverso. Una vez obtenido un tiempo de referencia en segundos, se considera "**eficiente**" si el tiempo medido es más bajo que el tiempo de referencia; será "**muy eficiente**" si el tiempo medido es menor que la mitad del tiempo de referencia, es decir, más del doble de rápido; y por último, será "**altamente eficiente**" si el tiempo tardado es menor que la tercera parte de la referencia, es decir, más del triple de rápido. Cualquier tiempo situado por encima de la referencia se califica de "**ineficiente**".

- **Grado de cumplimiento:** Relacionado con la calidad de los resultados obtenibles con las herramientas, se trata de un indicador que se corresponde con el **porcentaje de código generado** de smart contract o de modelo PCN. Para poder cuantificar el porcentaje de la manera más objetiva posible, se abstraen los elementos PCN que conforman el 100% del modelo generado y se les da un **valor** unitario. De este modo y mediante la suma de estos valores se puede ponderar el contenido existente en el modelo y estimar el porcentaje de código generado frente a lo que sería el total absoluto del código generable.

  Dadas las diferencias entre el código de un smart contract y el de un modelo PCN, la forma de calcular este indicador es ligeramente diferente según el tipo de modelo generado:

- **Modelo SmaC:** Dada la naturaleza funcional de un smart contract, la lógica y el conjunto de mecanismos que implementan en él las características funcionales de un diagrama PCN se ponderan como unidades mínimas de valor, además de la propia representación de cada elemento.

  De esta forma, la ponderación de los elementos PCN en un smart contract se establece como se indica en la tabla de la Figura 34.

Abstracción		**Ponderación**
Representación		1
Lógica funcional		1
Mecanismo de decisión		1
Mecanismo de espera		1
Mecanismo de dependencia		1
Mecanismo de dominio		1
Elemento PCN		**Ponderación total**
Diagrama (vacío)		Representación = **1**
Entidad		Representación = **1**
Dominio		(Representación + Mecanismo de dominio) * nº pasos = **2 * nº pasos**
Paso	Estándar	Representación + Lógica funcional = **2**
	Subcontrata	Representación + Lógica funcional = **2**
	Innovación	Representación + Lógica funcional = **2**
	Espera y Acción	Representación + Lógica funcional + Mecanismo de espera = **3**
	Espera	Representación + Mecanismo de espera = **2**
	Decisión	Representación + Lógica funcional + Mecanismo de decisión = **3**
Dependencia	Estándar	Representación + Mecanismo de dependencia = **2**
	Con retardo	Representación + Mecanismo de dependencia + Mecanismo de espera = **3**
	Positiva	Representación + Mecanismo de dependencia + Mecanismo de decisión = **3**
	Negativa	Representación + Mecanismo de dependencia + Mecanismo de decisión = **3**
Etiqueta		Representación = **1**

Figura 34. Tabla 4: Ponderaciones para el grado de cumplimiento de un modelo SmaC.

- **Modelo PCN:** De forma mucho más simplificada respecto al caso de un smart contract, el valor de cada elemento PCN en el propio modelo PCN se reduce a tan solo su representación. Esto es porque el código XMI de un modelo PCN es lo suficientemente simple como para que cada elemento pondere por igual.

  De este modo, la ponderación de los elementos en el modelo PCN queda como se plantea en la tabla de la Figura 35.

Abstracción	Ponderación
Representación	1
Elemento PCN	**Ponderación total**
Diagrama (vacío)	Representación = **1**
Entidad	Representación = **1**
Dominio	Representación * nº pasos = **nº pasos**
Paso	Representación = **1**
Dependencia	Representación = **1**
Etiqueta	Representación = **1**

Figura 35. Tabla 5: Ponderaciones para el grado de cumplimiento de un modelo PCN.

4.2 Caso de estudio 1: Deliveroo

El primer caso de estudio lo protagoniza una de las compañías de entrega de comida más populares, Deliveroo. Partiendo del servicio que ofrece la empresa a sus clientes modelado con PCN, se obtendrá un smart contract con el que después se obtendrá de nuevo un modelo PCN.

4.2.1 Definición

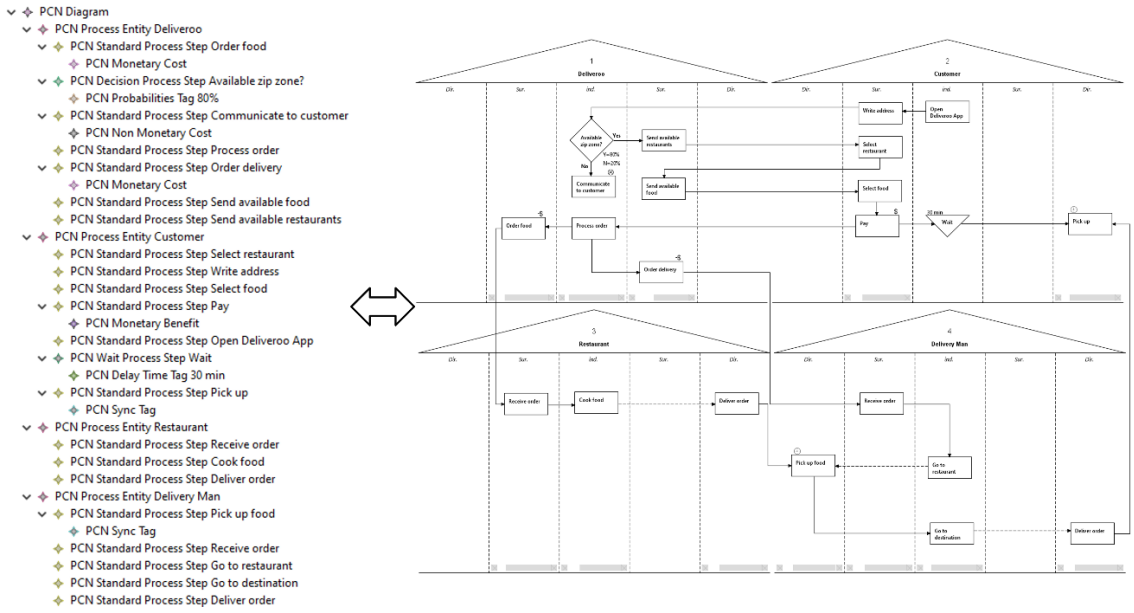


Figura 36. Caso de estudio 1: Modelo PCN (izda.) y diagrama PCN (dcha.).

El servicio sigue el siguiente proceso. Un cliente abre la aplicación de Deliveroo y manda la dirección de su domicilio. Con esta dirección, Deliveroo comprueba la zona postal y en caso de entrar en su área de actividad, hecho que sucede el 80% de las veces, envía al cliente los restaurantes disponibles a los que pedir comida; de lo contrario, notifica al cliente con un sentimiento de disculpa y se termina el proceso. En caso de seguir el proceso, el cliente selecciona un restaurante y Deliveroo envía la comida disponible para pedir en el restaurante seleccionado. Una vez el cliente selecciona la comida, paga y esperará unos 30 minutos hasta que pueda recoger su pedido de manos del repartidor. Deliveroo obtiene

ingresos del pago del cliente y procesa el pedido, por un lado pidiendo la comida al restaurante seleccionado y emitiendo la orden de reparto al repartidor. Por su parte, el restaurante recibe el pedido, prepara la comida y la entrega al repartidor. Por otro lado, el repartidor recibe la orden de entrega, va al restaurante y recoge la comida para, posteriormente, ir al domicilio del cliente y hacer entrega del pedido.

El proceso descrito se modela con PCN creando su respectivo modelo PCN en INNoVaServ. Para facilitar su creación se utiliza el diagramador que incluye el plug-in. El contenido de ambos ficheros, modelo y diagrama, se muestran en la Figura 36.

4.2.2 Ejecución

A partir del modelo PCN original basado en el caso a tratar, se utiliza el puente de PCN a smart contract para generar un smart contract. El proceso de generación atraviesa distintas interfaces, mostradas en la Figura 37, con las que establecer las direcciones de las entidades y dar valor a los tiempos de pasos Espera y dependencias Con retardo. Una vez finalizado este proceso y habiendo medido el tiempo tardado, se obtiene el **smart contract generado** en forma de modelo SmaC.

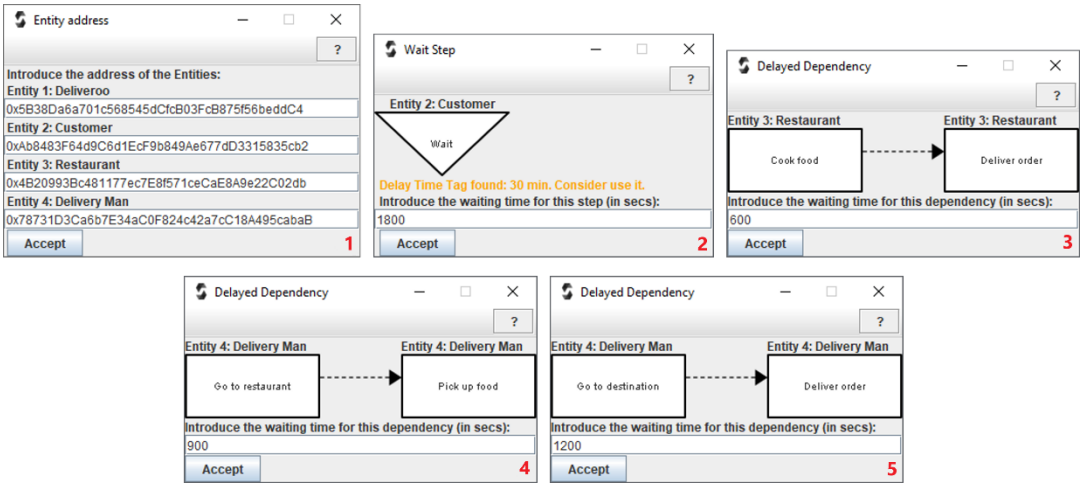


Figura 37. Caso de estudio 1: Secuencia de interfaces para la generación del smart contract.

Después, el smart contract recién generado se utiliza para generar un modelo PCN. Para ello se utiliza el puente de smart contract a PCN, con el que de manera completamente automática y midiendo el tiempo tardado se obtiene el **modelo PCN generado**.

4.2.3 Resultados

Como resultado, se puede observar parcialmente el smart contract generado en la Figura 38 y el modelo PCN generado en la Figura 39. Tras la ejecución del caso, se aplican los criterios establecidos a cada proceso de transformación:

- **Eficiencia:** Habiendo calculado de antemano el tiempo en hacer manualmente el mismo smart contract y el mismo modelo PCN que los generados durante la ejecución del caso, se toman estas medidas como tiempos de referencia para evaluar la eficiencia.
    - **Smart contract:** El tiempo manual se sitúa en 3720,525s y dado que el tiempo de ejecución logrado es de 24,816s, se cataloga como **altamente eficiente**.

- **PCN:** El tiempo manual es de 410,419s y ya que el tiempo de ejecución obtenido es de 0,024s, es etiquetado como **altamente eficiente**.

- **Grado de cumplimiento:** Siguiendo las tablas de ponderación vistas en la Figura 34 y la Figura 35, se calcula la puntuación total posible y la puntuación obtenida para cada modelo generado.

  - **Smart contract:**

    Diagrama (vacío): 1
    Entidades: 1 + 1 + 1 + 1 = 4
    *Dominio: 22*2 = 44*
    *Pasos: 20*2(Estándar) + 3(Decisión) + 2(Espera) = 45*
    *Dependencias: 18*2(Estandar) + 3*3(Con retardo) + 3(Positiva) + 3(Negativa) = 51*
    Etiquetas: 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 8
    Total absoluto (100%):** 1 + 4 + 44 + 45 + 51 + 8 = **153

    Diagrama (vacío): 1
    Entidades: 1 + 1 + 1 + 1 = 4
    *Dominio: 22*2 = 44*
    *Pasos: 20*1(Estándar) + 2(Decisión) + 1(Espera) = 23*
    *Dependencias: 18*2(Estandar) + 3*3(Con retardo) + 3(Positiva) + 3(Negativa) = 51*
    Etiquetas: 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 8
    Total generado:** 1 + 4 + 44 + 23 + 51 + 8 = **131

    Siendo el smart contract completo una suma de 153, el total generado de 131 implica la obtención de un **85,62%** de código generado.

  - **PCN:**

    Diagrama (vacío): 1
    Entidades: 1 + 1 + 1 + 1 = 4
    Dominio: 22
    Pasos: 20(Estándar) + 1(Decisión) + 1(Espera) = 22
    Dependencias: 18(Estandar) + 3(Con retardo) + 1(Positiva) + 1(Negativa) = 23
    Etiquetas: 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 8
    Total absoluto (100%):** 1 + 4 + 22 + 22 + 23 + 8 = **80

    Diagrama (vacío): 1
    Entidades: 1 + 1 + 1 + 1 = 4
    Dominio: 22
    Pasos: 20(Estándar) + 1(Decisión) + 1(Espera) = 22
    Dependencias: 18(Estandar) + 3(Con retardo) + 1(Positiva) + 1(Negativa) = 23
    Etiquetas: 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 8
    Total generado:** 1 + 4 + 22 + 22 + 23 + 8 = **80

    Siendo el PCN completo una suma de 80, la generación del modelo PCN concluye con un **100%** de código generado.

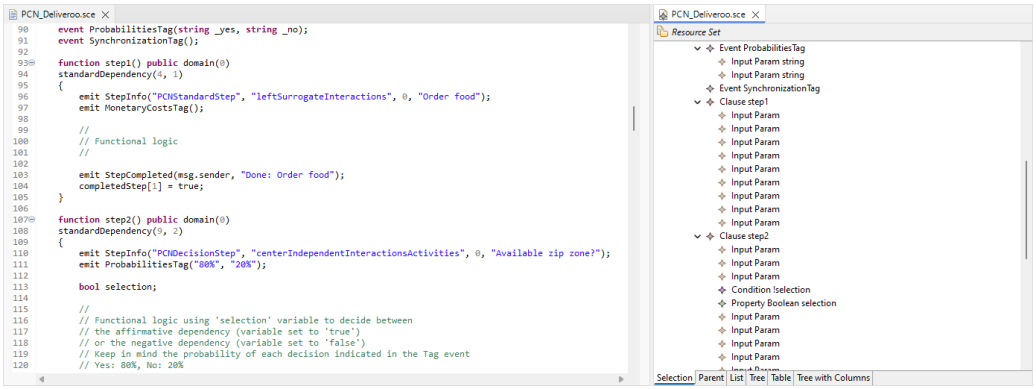


Figura 38. Caso de estudio 1: Modelo SmaC generado en formato texto (izda.) y árbol (dcha.).

Figura 39. Caso de estudio 1: Modelo PCN generado en formato texto (izda.) y árbol (dcha.).

4.3 Caso de estudio 2: Centro de salud

Este caso gira en torno al servicio de un centro de salud y su interacción con el paciente y otras entidades involucradas. Nuevamente, partiendo de un modelo PCN del servicio, se obtendrá un smart contract que posteriormente será convertido otra vez en modelo PCN.

4.3.1 Definición

El proceso del servicio comienza cuando un paciente se encuentra mal y acude al centro de salud. Allí hace el check-in y pasa a la sala de espera durante un tiempo indeterminado. Cuando llega el turno del paciente, comparte sus síntomas con el centro y este le toma una muestra de sangre. El centro, por su lado, ha suministrado las herramientas de laboratorio necesarias esterilizándolas y enseñando al personal a utilizarlas para realizar un análisis de sangre del paciente. Tras el análisis, el centro receta medicación al paciente, envía la receta a la farmacia y presenta una solicitud de pago a la compañía de seguros. La compañía de seguros recibe esta solicitud y paga la cantidad cubierta con la previa definición de un cronograma de pagos por parte de un tercero. Luego, la compañía de seguros establece un acuerdo de cobertura. Por otra parte, el paciente va a una farmacia enseñando su identificación. Después, la farmacia comprueba la identificación del paciente, prepara la receta y comprueba el acuerdo de cobertura. Tras esto, la farmacia envía el reclamo del pago a la compañía de seguros para que esta procese el pago y solicita al paciente la cantidad de copago. Por último, el cliente aporta dicho pago, toma la medicación recetada y finalmente pasa a sentirse mejor.

Como punto de partida, el proceso es definido en PCN con INNoVaServ haciendo uso del diagramador que incorpora. En la Figura 40 se puede observar el modelo PCN obtenido y el diagrama empleado para construirlo.

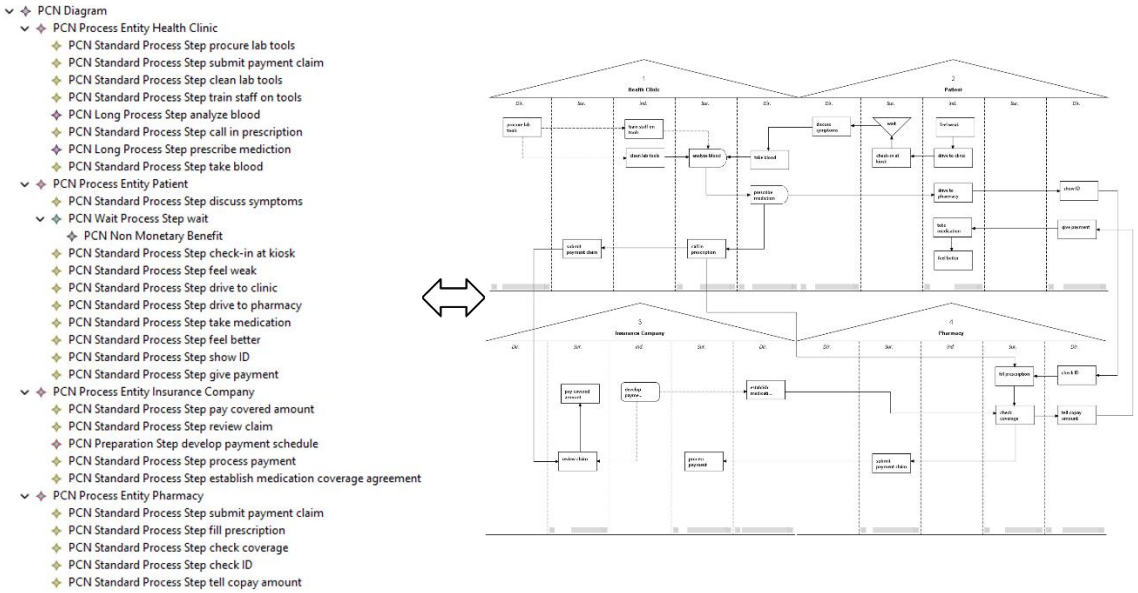


Figura 40. Caso de estudio 2: Modelo PCN (izda.) y diagrama PCN (dcha.).

4.3.2 Ejecución

Una vez creado un modelo PCN inicial, se lanza la herramienta puente en dicho fichero para comenzar el proceso de generación de un smart contract basado en el PCN del modelo. El proceso atraviesa diferentes interfaces, vistas en la Figura 41, con las que fijar las direcciones de entidades tanto principales como terceras junto a los tiempos de dependencias Con retardo y de pasos que involucran a un tiempo de espera. Tras completar cada etapa del proceso y obtener el tiempo de ejecución, se **genera** el modelo SmaC que contiene el **smart contract**.

A continuación se lanza la segunda herramienta puente en el fichero que contiene el modelo SmaC recién generado para transformarlo en modelo PCN. De la misma forma y tras finalizar el proceso, se mide el tiempo tardado y se obtiene el fichero con el **modelo PCN generado** igual al modelo PCN original.

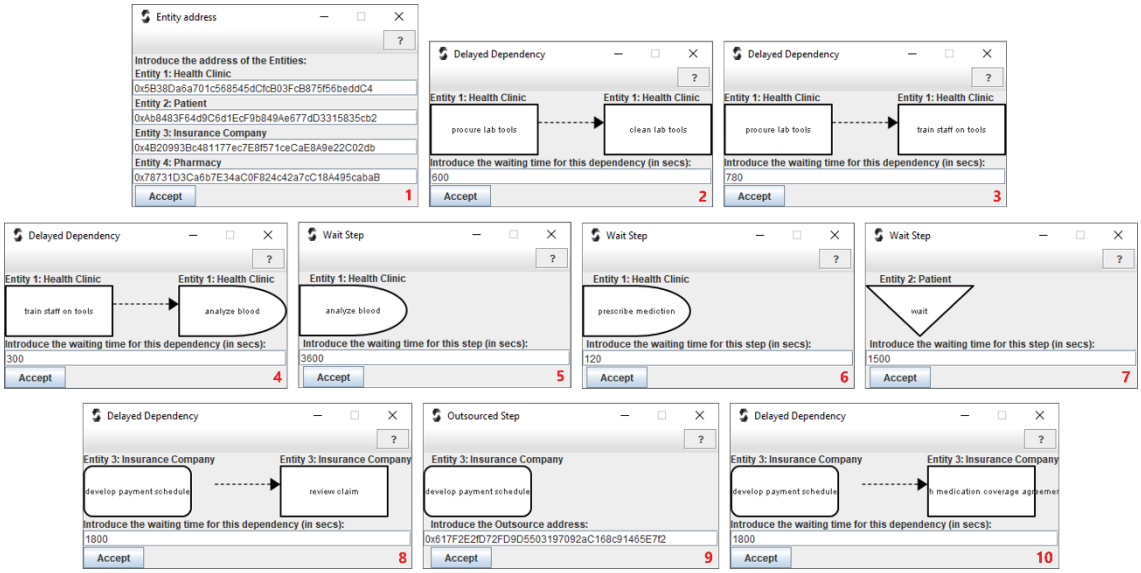


Figura 41. Caso de estudio 2: Secuencia de interfaces para la generación del smart contract.

4.3.3 Resultados

Como resultado de la ejecución, se muestra una parte del smart generado en la Figura 42 y del modelo PCN generado en la Figura 43. Después, se aplican los criterios definidos a cada proceso realizado:

- **Eficiencia:** Tras calcular el tiempo estimado para hacer manualmente el smart contract y el modelo PCN generados, se toman estos tiempos como referencia para evaluar el aspecto eficiente de las herramientas.
    - **Smart contract:** Con un tiempo de referencia de 4461,573s y puesto que el tiempo de ejecución obtenido es de 41,499s, se considera **altamente eficiente**.
    - **PCN:** De igual manera, con una referencia de 435,254s, el tiempo logrado se sitúa en 0,038s. Por tanto, se califica como **altamente eficiente**.
- **Grado de cumplimiento:** Conforme a las tablas de ponderación mostradas en la Figura 34 y la Figura 35, se calcula la puntuación total posible y la puntuación lograda en cada modelo generado.
    - **Smart contract:**

      Diagrama (vacío): 1
      Entidades: 1 + 1 + 1 + 1 = 4
      *Dominio: 28*2 = 56*
      *Pasos: 24*2(Estándar) + 2*3(Acción y Espera) + 2(Espera) + 2(Subcontrata) = 58*
      *Dependencias: 25*2(Estandar) + 5*3(Con retardo) = 65*
      Etiquetas: 0
      Total absoluto (100%):** 1 + 4 + 56 + 58 + 65 + 0 = **184

      Diagrama (vacío): 1
      Entidades: 1 + 1 + 1 + 1 = 4
      *Dominio: 28*2 = 56*
      *Pasos: 24(Estándar) + 2*2(Acción y Espera) + 1(Espera) + 1(Subcontrata) = 30*
      *Dependencias: 25*2(Estandar) + 5*3(Con retardo) = 65*
      Etiquetas: 0
      Total generado:** 1 + 4 + 56 + 30 + 65 + 0 = **156

      Según los cálculos realizados y puesto que el smart contract completo tiene un valor de 184, el total generado de 156 significa la obtención de un **84,78%** del código generado.
    - **PCN:**

      Diagrama (vacío): 1
      Entidades: 1 + 1 + 1 + 1 = 4
      Dominio: 28
      Pasos: 24(Estándar) + 2(Acción y Espera) + 1(Espera) + 1(Subcontrata) = 28
      Dependencias: 25(Estandar) + 3(Con retardo) = 30
      Etiquetas: 0
      Total absoluto (100%):** 1 + 4 + 28 + 28 + 30 + 0 = **91

      Diagrama (vacío): 1
      Entidades: 1 + 1 + 1 + 1 = 4
      Dominio: 28
      Pasos: 24(Estándar) + 2(Acción y Espera) + 1(Espera) + 1(Subcontrata) = 28
      Dependencias: 25(Estandar) + 3(Con retardo) = 30
      Etiquetas: 0
      Total generado:** 1 + 4 + 28 + 28 + 30 + 0 = **91

      Con una suma de 91 en el PCN completo, la generación del modelo PCN termina el **100%** del código generado.

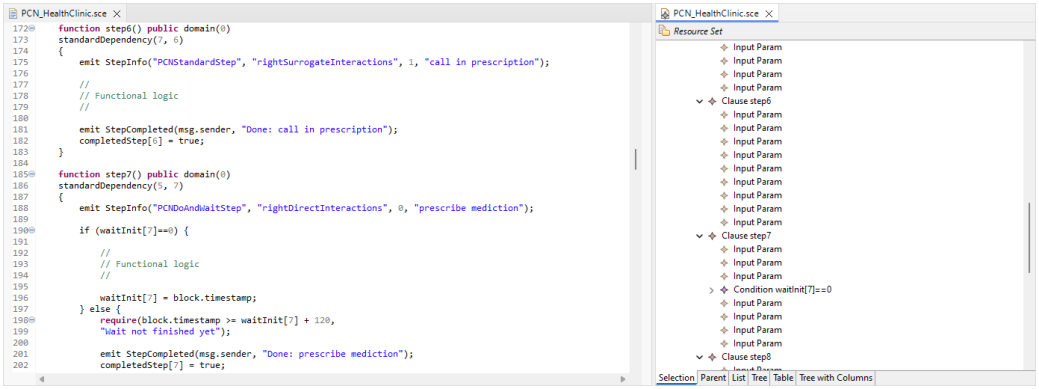


Figura 42. Caso de estudio 2: Modelo SmaC generado en formato texto (izda.) y árbol (dcha.).

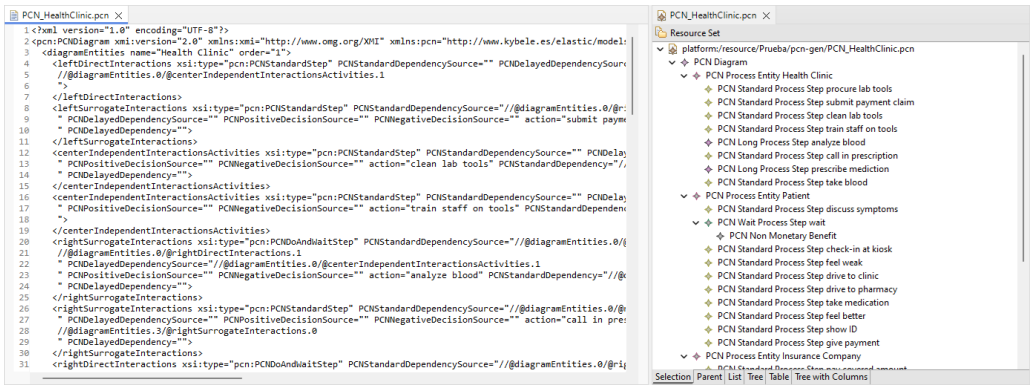


Figura 43. Caso de estudio 2: Modelo PCN generado en formato texto (izda.) y árbol (dcha.).

4.4 Caso de estudio 3: Restaurante

Este caso de estudio trata sobre el servicio que ofrece un restaurante especializado en pizza. Utilizando como punto de partida el modelo PCN que define dicho servicio, se generará un smart contract, que a su vez, será utilizado para generar de nuevo un modelo PCN.

4.4.1 Definición

El restaurante pizzería sigue el siguiente procedimiento. Por una parte, a un cliente hambriento se le antoja una pizza, lo que le genera un sentimiento negativo por no poder hacerlo en ese momento. Esto da lugar a que haga un pedido a la pizzería vía web. Este pedido es procesado por la pizzería y conduce a la preparación de la pizza bajo la previa compra de los ingredientes necesarios y el suministro del material de cocina con la inversión económica que eso conlleva. En paralelo, el cliente viaja al restaurante manteniendo su descontento. Cuando la pizza está lista, es entregada por el restaurante y pagada por el cliente, generando ingresos económicos para el restaurante y satisfacción en el cliente, que puede volver a su casa, aún con hambre, y comer la pizza en su casa plenamente feliz.

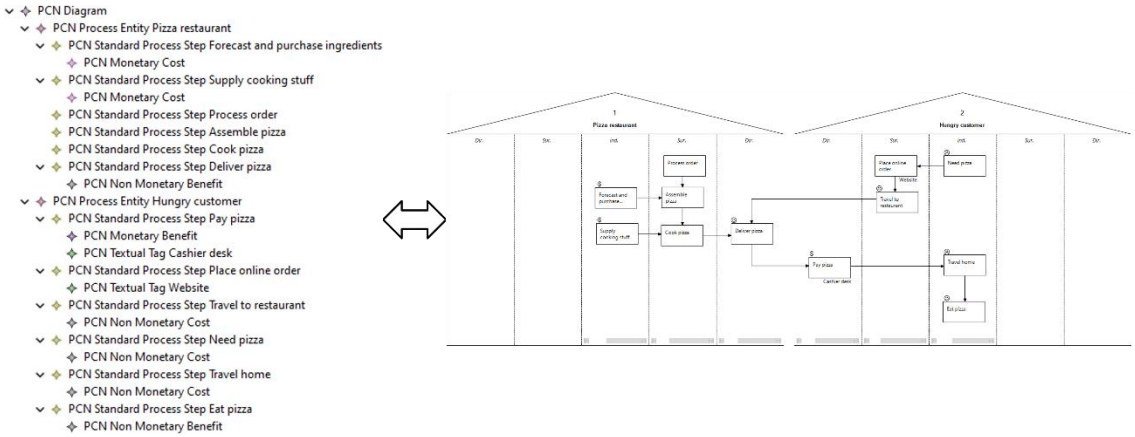


Figura 44. Caso de estudio 3: Modelo PCN (izda.) y diagrama PCN (dcha.).

Este procedimiento, una vez más, representado mediante PCN en INNoVaServ da lugar a un modelo PCN obtenido a partir de un diagrama de este. Ambos ficheros, modelo y diagrama, se muestran en la Figura 44.

4.4.2 Ejecución

Tras la creación del modelo PCN mencionado, se hace uso de la herramienta implementada para generar un smart contract a partir de este. Dada la naturaleza del PCN recogido en este caso de estudio, el proceso es más corto que en otros casos al solo pasar por la fase de introducción de direcciones para las entidades (véase Figura 45). Al acabar el procedimiento con la medición del tiempo tardado, se obtiene el fichero con el **smart contract generado** en forma de modelo SmaC.

Figura 45. Caso de estudio 3: Interfaz para la generación del smart contract.

Después de la obtención del modelo SmaC, se hace uso de la segunda herramienta implementada para generar un modelo PCN igual al modelo PCN de partida a partir del smart contract recién generado. Tras completar este proceso automático, incluyendo la medición del tiempo empleado en la tarea, se consigue el **modelo PCN generado**.

4.4.3 Resultados

En la Figura 46 se observa parte del smart contract generado mientras que en la Figura 47 se observa el modelo PCN generado. Con la obtención de los modelos resultantes, se evalúa cada proceso bajo los criterios definidos:

- **Eficiencia:** Se utilizan medidas de tiempo de la creación manual del mismo modelo PCN y smart contract generados como tiempos de referencia en la evaluación de este aspecto.
    - **Smart contract:** Siendo 2323,514s el tiempo de referencia de y 7,632s el tiempo logrado con la ejecución de la herramienta, se califica de **altamente eficiente**.

- **PCN:** En este caso, el tiempo de referencia es de 175,203s y el tiempo obtenido con la ejecución es de 0,017s, luego se considera **altamente eficiente**.

- **Grado de cumplimiento:** Se utilizan las tablas de ponderación de la Figura 34 y la Figura 35 para calcular las puntuaciones totales absolutas y obtenidas para los modelos generados.

  - **Smart contract:**

    Diagrama (vacío): 1
    Entidades: 1 + 1 = 2
    *Dominio: 12*2 = 24*
    *Pasos: 12*2(Estándar) = 24*
    *Dependencias: 11*2(Estandar) = 22*
    Etiquetas: 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 9
    Total absoluto (100%):** 1 + 2 + 24 +24 + 22 + 9 = **82

    Diagrama (vacío): 1
    Entidades: 1 + 1 = 2
    *Dominio: 12*2 = 24*
    Pasos: 12(Estándar) = 12
    *Dependencias: 11*2(Estandar) = 22*
    Etiquetas: 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 9
    Total generado:** 1 + 2 + 24 +12 + 22 + 9 = **70

    Puesto que el 100% del smart contract se alcanza con una suma de 82, el porcentaje de código generado con la suma de 70 es del **85,36%**.

  - **PCN:**

    Diagrama (vacío): 1
    Entidades: 1 + 1 = 2
    Dominio: 12
    Pasos: 12(Estándar) = 12
    Dependencias: 11(Estandar) = 11
    Etiquetas: 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 9
    Total absoluto (100%):** 1 + 2 + 12 +12 + 11 + 9 = **47

    Diagrama (vacío): 1
    Entidades: 1 + 1 = 2
    Dominio: 12
    Pasos: 12(Estándar) = 12
    Dependencias: 11(Estandar) = 11
    Etiquetas: 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 9
    Total generado:** 1 + 2 + 12 +12 + 11 + 9 = **47

    Obteniendo el total absoluto del modelo con un valor de 47, el valor generado de 47 implica la obtención de la totalidad del **100%** de código generado.

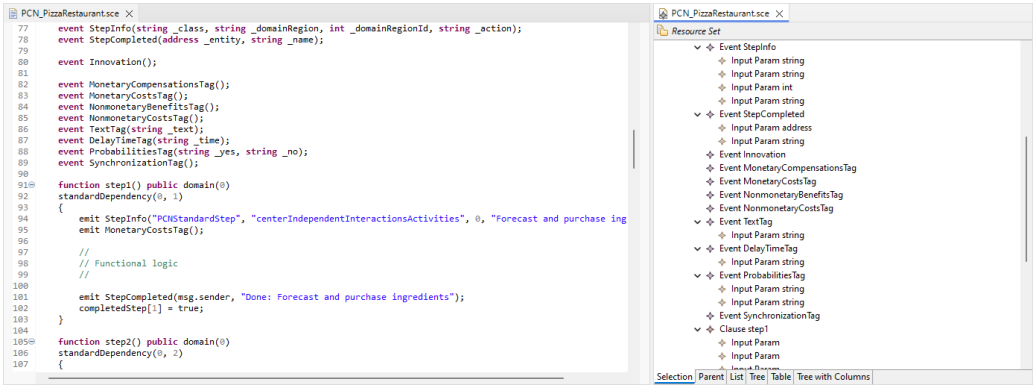


Figura 46. Caso de estudio 3: Modelo SmaC generado en formato texto (izda.) y árbol (dcha.).

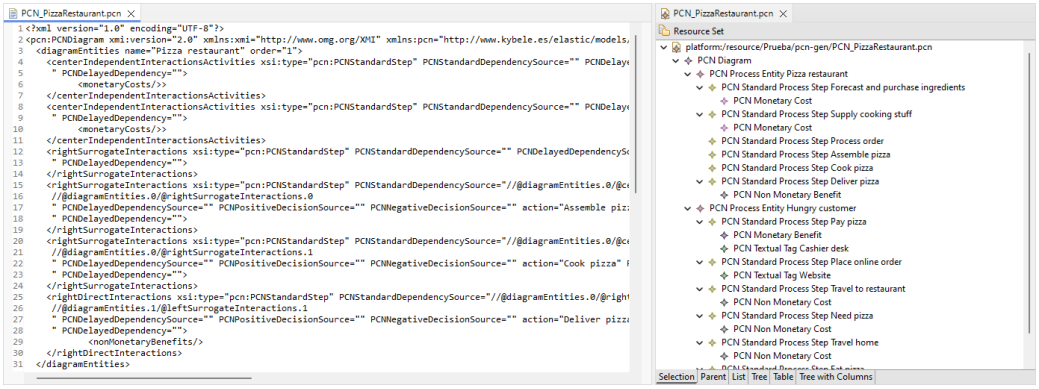


Figura 47. Caso de estudio 3: Modelo PCN generado en formato texto (izda.) y árbol (dcha.).

4.5 Caso de estudio 4: Clínica de atención primaria

Como último caso de estudio, se plantea el servicio que da una clínica de atención primaria. Una vez más, definiendo el proceso de este servicio en PCN, se generará un smart contract con el que después generar un modelo PCN.

4.5.1 Definición

El proceso que sigue el servicio es el siguiente. Un paciente presenta algún problema de salud y acude a la clínica, en donde comenta sus síntomas con un responsable de la clínica. En caso no de requerir de un especialista se tramita una receta y termina el proceso; en caso contrario se proporciona y procesa una remisión al especialista, para lo que la clínica notificará al coordinador de atención. Después de que el coordinador revise la remisión, procede a programar la cita del paciente con el especialista y revisa con el paciente su condición para saber si precisa de transporte. En caso de necesitar transporte, el coordinador programa un servicio de transporte para el paciente, para que después el proveedor de transporte recoja al paciente y lo lleve hasta el especialista. Una vez reunidos, el paciente y el especialista hablan de los síntomas. Luego el paciente proporciona muestras para analizar que son recogidas y analizadas por el especialista para terminar de proporcionar el tratamiento a recibir por el paciente. Tras esto, el especialista notifica al coordinador del caso para llevar un seguimiento. Adicionalmente, en caso de precisar seguimiento también por parte de la clínica, el coordinador programa una cita de control con la clínica para seguir de cerca la evolución del paciente.

El proceso descrito es representado en PCN en INNoVaServ, originando un modelo PCN a partir de un diagrama. En la Figura 48 se puede observar tanto el diagrama como el modelo PCN que será utilizado como punto de partida en la ejecución del caso.

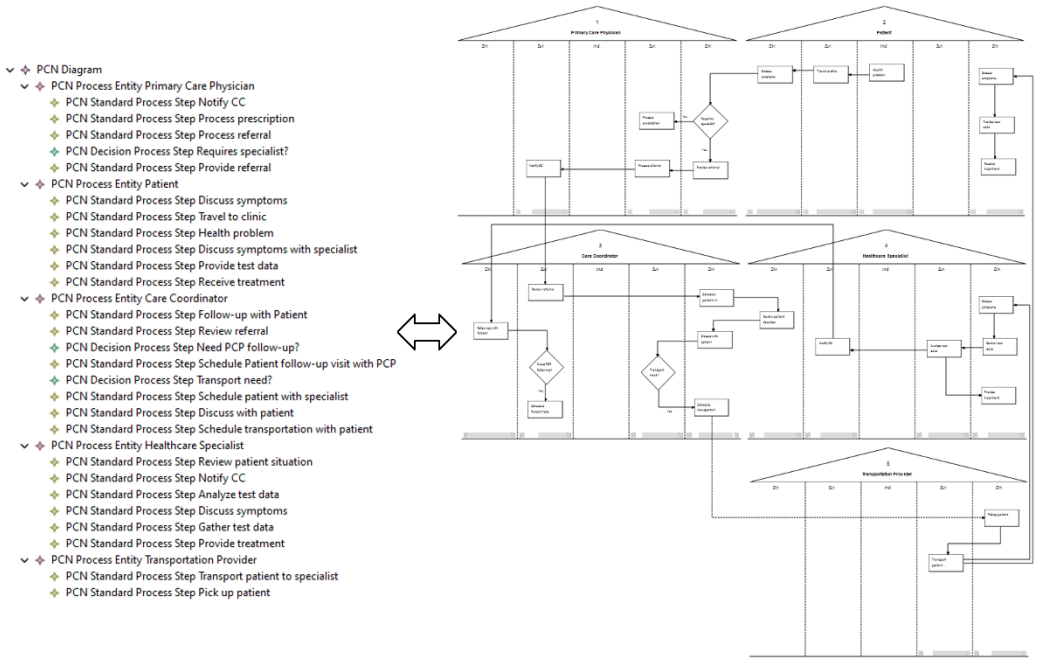


Figura 48. Caso de estudio 4: Modelo PCN (izda.) y diagrama PCN (dcha.).

4.5.2 Ejecución

Se hace uso de la herramienta con el modelo PCN mencionado, dando lugar a la creación de un smart contract basado en dicho PCN. El proceso de creación pasa por la secuencia de interfaces que se aprecia en la Figura 49. Terminado el proceso y tomada la medición del tiempo tardado, se obtiene el modelo SmaC que contiene el **smart contract generado**.

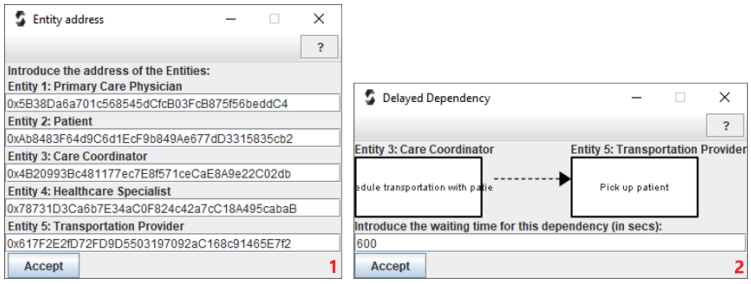


Figura 49. Caso de estudio 4: Secuencia de interfaces para la generación del smart contract.

Tras esto, el smart contract obtenido se usa con la segunda herramienta. El resultado tras finalizar el proceso y medir el tiempo tardado es un **modelo PCN generado** idéntico al primer modelo PCN utilizado.

4.5.3 Resultados

Tras la ejecución, la Figura 50 muestra una porción del smart contract generado y la Figura 51 muestra el modelo PCN generado. Ahora se evalúa cada proceso de generación bajo los criterios definidos:

- **Eficiencia:** Se comprueba la eficiencia de las herramientas comparando los tiempos logrados en la ejecución del caso con tiempos de referencia resultantes de replicar los procesos a mano.
    - **Smart contract:** Con un tiempo de referencia de 4314,115s y un tiempo de ejecución de 20,221s, el proceso se califica como **altamente eficiente**.

- **PCN:** Dado que el tiempo de referencia se sitúa en 446,0,86s y el tiempo obtenido es de 0,053s, el proceso es calificado como **altamente eficiente**.

- **Grado de cumplimiento:** Se da valor al código generado calculando el valor total posible y el valor conseguido en cada modelo de acuerdo a las tablas de ponderación mostradas en el apartado 4.1.

  - **Smart contract:**

    Diagrama (vacío): 1
    Entidades: 1 + 1 +1 +1 +1 = 5
    *Dominio: 27*2 = 54*
    *Pasos: 24*2(Estándar) + 3*3(Decisión) = 57*
    *Dependencias: 21*2(Estandar) + 3(Con retardo) + 3*3(Positiva) + 3(Negativa)= 57*
    Etiquetas: 0
    Total absoluto (100%):** 1 + 5 + 54 +57 + 57 + 0 = **174

    Diagrama (vacío): 1
    Entidades: 1 + 1 +1 +1 +1 = 5
    *Dominio: 27*2 = 54*
    *Pasos: 24(Estándar) + 3*2(Decisión) = 30*
    *Dependencias: 21*2(Estandar) + 3(Con retardo) + 3*3(Positiva) + 3(Negativa)= 57*
    Etiquetas: 0
    Total generado:** 1 + 5 + 54 +30 + 57 + 0 = **147

    Considerando que el valor del smart contract íntegro es de 174 y el valor generado es de 147, el porcentaje de código generado se sitúa en el **84,48%**.

  - **PCN:**

    Diagrama (vacío): 1
    Entidades: 1 + 1 +1 +1 +1 = 5
    Dominio: 27
    Pasos: 24(Estándar) + 3(Decisión) = 27
    Dependencias: 21(Estandar) + 1(Con retardo) + 3(Positiva) + 1(Negativa)= 26
    Etiquetas: 0
    Total absoluto (100%):** 1 + 5 + 27 +27 + 26 + 0 = **86

    Diagrama (vacío): 1
    Entidades: 1 + 1 +1 +1 +1 = 5
    Dominio: 27
    Pasos: 24(Estándar) + 3(Decisión) = 27
    Dependencias: 21(Estandar) + 1(Con retardo) + 3(Positiva) + 1(Negativa)= 26
    Etiquetas: 0
    Total generado:** 1 + 5 + 27 +27 + 26 + 0 = **86

    El modelo PCN en su totalidad se puntúa con 86. Ya que el valor del modelo generado es 86, se puede confirmar la obtención del **100%** del código generado.

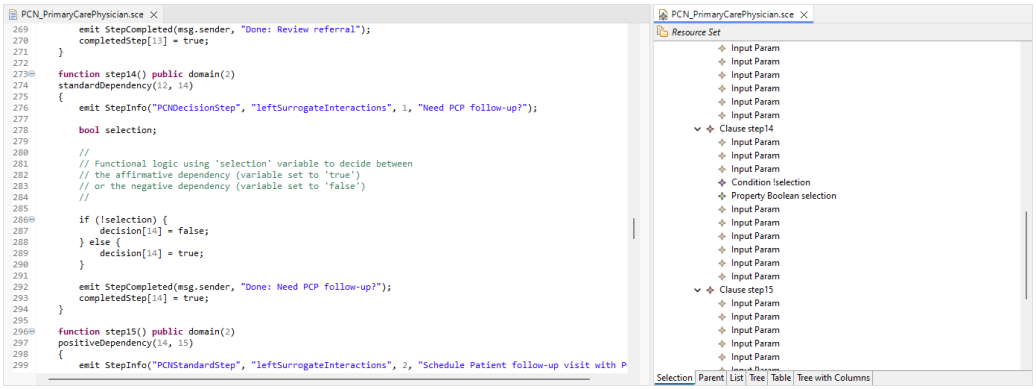


Figura 50. Caso de estudio 4: Modelo SmaC generado en formato texto (izda.) y árbol (dcha.).

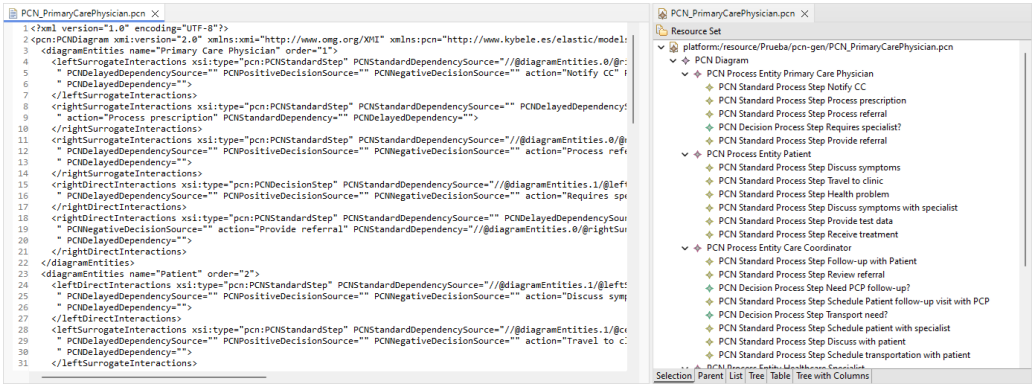


Figura 51. Caso de estudio 4: Modelo SmaC generado en formato texto (izda.) y árbol (dcha.).

4.6 Análisis de los resultados

Recapitulando, se recogen los resultados obtenidos en cada uno de los casos de estudio vistos en los apartados anteriores para concluir este capítulo de validación con una visión en conjunto.

Por cada criterio establecido en el apartado 4.1 se valora el proceso de generación de PCN y el proceso de generación de smart contract individualmente. Esto permite sacar conclusiones sobre cada herramienta en los aspectos evaluados. Los resultados en cuestión se reúnen en la tabla de la Figura 52.

Caso de estudio	Eficiencia		Grado de cumplimiento	
	PCN	Smart contract	PCN	Smart contract
Caso de estudio 1	Altamente eficiente (0,024s)	Altamente eficiente (24,816s)	100 %	85,62%
Caso de estudio 2	Altamente eficiente (0,038s)	Altamente eficiente (41,499s)	100%	84,78%
Caso de estudio 3	Altamente eficiente (0,017s)	Altamente eficiente (7,632s)	100%	85,36%
Caso de estudio 4	Altamente eficiente (0.053s)	Altamente eficiente (20,221s)	100%	84,48%
Media	**Altamente eficiente (0,033s)**	**Altamente eficiente (23,542s)**	**100%**	**85,06%**

Figura 52. Tabla 6: Resultados generales de la validación.

El análisis de los resultados vistos de manera conjunta lleva a las siguientes conclusiones:

- **Eficiencia:** Los tiempos obtenidos en todos los casos son concluyentes, situando a las herramientas como los procesos **más eficientes** frente a procedimientos manuales, destacando los tiempos de fracciones de segundo en la generación de modelos PCN.

  Otro aspecto a mencionar es la relación proporcional que existe entre el tiempo necesario para la generación de smart contracts y los datos aportados por el usuario, siendo mayor el tiempo que se tarda en generar un smart contract cuanta más información y atención deba aportar el usuario a través de interfaces.

- **Grado de cumplimiento:** Los porcentajes de código calculados evidencian que el puente de smart contract a PCN **cumple plenamente** con su objetivo al generar la totalidad de los modelos PCN. No se puede decir lo mismo del puente de PCN a smart contract, donde el porcentaje de smart contract que no es posible generar se corresponde con la lógica funcional que describe la acción de cada paso (p.ej. “preparar pedido”) debido a la imposibilidad de considerar un sinfín de acciones posibles; sin embargo, se considera que el porcentaje medio logrado del 85,06% **cumple con creces** dentro de lo factible.

También, dado que en cada caso de estudio el objeto de partida es un modelo PCN, se garantiza una **interoperabilidad** entre las herramientas, puesto que al generar un smart contract a partir de este PCN se puede generar en todos los casos una copia exacta del PCN original a partir del smart contract generado.

Capítulo 5 - Conclusiones y Trabajos futuros

Dando por finalizada la memoria de este proyecto, este apartado expone las **conclusiones** personales en términos de desarrollo, aprendizaje, resultados obtenidos y aplicación acerca del trabajo realizado, así como una **evaluación del esfuerzo** dedicado y un vistazo a posibles líneas de **trabajo futuras**.

5.1 Conclusiones y valoración

Al hablar de **desarrollo**, el hecho de hacer posible la idea planteada ha supuesto un éxito en el cumplimiento de los objetivos que ha repercutido positivamente en la trayectoria del proyecto. Sin embargo, como sucede en cualquier desarrollo de software, pueden surgir diversos problemas que retrasen el tiempo de desarrollo, que deriven en un desvío de la hoja de ruta establecida o que precisen de la búsqueda de soluciones alternativas a las pensadas en un principio. Como en este proyecto no ha sido diferente, se exponen los problemas principales afrontados durante el desarrollo de la solución:

- Al comienzo del desarrollo de ambas herramientas surgieron dificultades para instalar parte de las dependencias necesarias en Eclipse para hacer funcionar el modelo PCN. Esto fue debido a que estas dependencias pertenecen al proyecto GMF Tooling, descontinuado hoy en día; no obstante, pudieron ser obtenidas de un repositorio de terceros.
- La unión de la clase manejadora con la clase generadora en ambas herramientas pretendía realizarse con un conjunto de clases cuyos métodos facilitaban el paso de los recursos de los modelos. Sin embargo, la existencia de múltiples problemas relacionados con la inyección automática de objetos empujó a buscar una solución alternativa pero idéntica bajo el uso de clases que no dependieran de inyecciones automáticas.
- Durante el desarrollo del puente de PCN a smart contract surgió la necesidad de invalidar el uso de elementos Referencia como parte de los diagramas PCN admitidos por la herramienta. Esto fue debido a la imposibilidad de interpretar correctamente dichos elementos dada la estructuración interna del DSL gráfico que modela los diagramas, permitiendo definir Referencias salientes pero no Referencias entrantes. La medida fue tomada teniendo presente la alternativa equivalente de poder sustituir Referencias por dependencias Estándar.
- En el desarrollo del puente de smart contract a PCN hubo que modificar ligeramente la gramática original de SmaC para contemplar casos de sintaxis presentes en el formato de smart contract basado en PCN que son válidos pero que el DSL no contemplaba.

Respecto a los **conocimientos depositados** en programación, dada su importancia en el proyecto, cabe dejar una breve apreciación por cada uno de los lenguajes tratados a lo largo del trabajo realizado:

- **Java:** La alta familiarización con este lenguaje tras varios años de experiencia ha supuesto que la mayor parte del desarrollo de la solución sea en un entorno reconocido y cómodo.
- **Solidity:** Tras no haber tenido experiencia previa con smart contracts ni con tecnología blockchain, uno de los retos que proponía el proyecto fue aprender Solidity de cero durante la fase previa al desarrollo de las herramientas. Esta tarea se abordó siguiendo la documentación oficial y antes de lo esperado dada su similitud con Java.
- **Xtext:** Aunque la participación de Xtext en el proyecto se limita a su uso en SmaC, sin cierto conocimiento de su sintaxis aprendido durante el proceso no habría sido posible modificar la gramática de SmaC para ajustar inconvenientes y evitar retrasos en el desarrollo de las herramientas.
- **Xtend:** A pesar de que la funcionalidad requerida de Xtend se reduce tan solo a una de sus características, las plantillas, ha desempeñado un gran papel facilitando la labor de generar código. La documentación oficial y su estrecha afinidad con Java permitieron usar el lenguaje con comodidad.

Por otro lado, los **resultados** obtenidos evidencian la posibilidad real de unir procesos de negocio y smart contracts. Los casos de estudio evaluados han permitido cuantificar la calidad de las herramientas a la hora de transformar modelos PCN en smart contracts y viceversa, logrando resultados favorablemente positivos bajo los criterios utilizados en cuestiones de eficacia y cumplimiento.

Respecto a **aplicación** y relevancia en su área, el proyecto logra, no solo acercar los smart contracts a profesionales de negocio reduciendo la brecha existente, sino contribuir a la transformación digital con el planteamiento de una aproximación a la automatización de procesos y servicios, en especial los que más se consumen; es la razón por la que los casos de estudio planteados se basan en servicios que muchas personas suelen solicitar con cierta frecuencia.

5.2 Estimación de esfuerzos

Otro punto sobre el que reflexionar es el esfuerzo y el tiempo dedicados durante la elaboración del proyecto. Aunque medir o llevar un registro exacto de estos factores, sobre todo el esfuerzo empeñado, es bastante complejo, se puede llegar a representar una aproximación con ayuda de un **cronograma**.

A continuación, en la Figura 53, se muestra el cronograma del proyecto en formato tabular. A través de esta tabla se puede apreciar el tiempo y, de cierta manera, el esfuerzo empleado en el proyecto a lo largo del proceso en distintas etapas de este. Para ello, se han situado las tareas principales realizadas y se han marcado las semanas en las que se ha trabajado cada una de ellas.

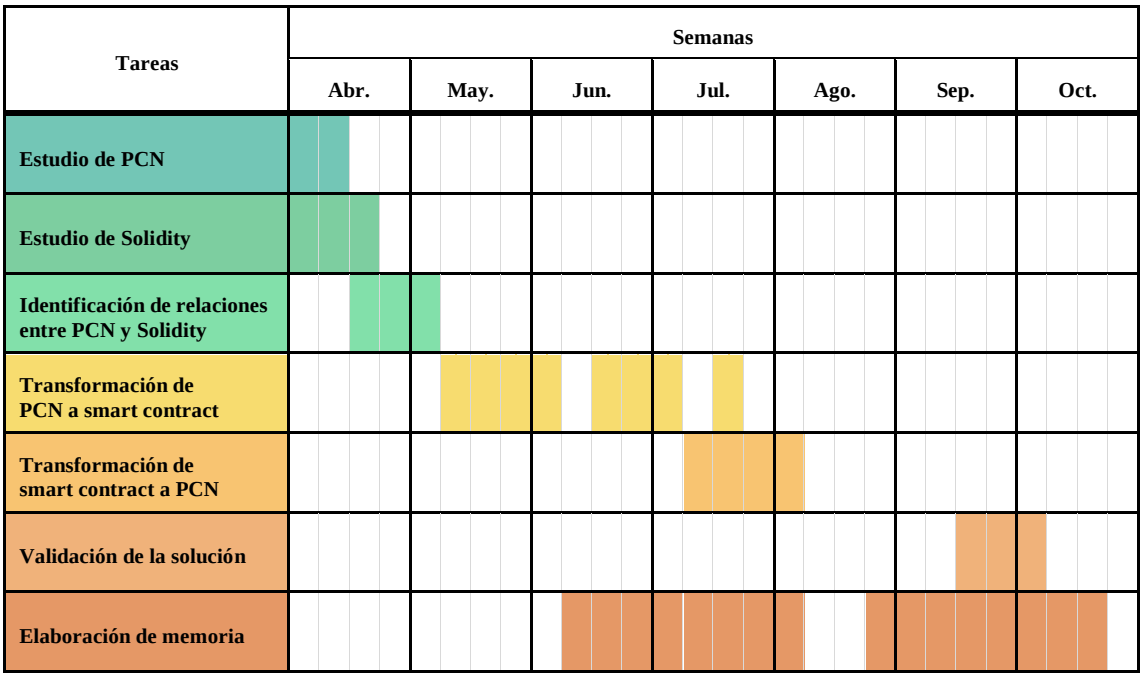


Figura 53. Tabla 7: Cronograma del proyecto.

Con las tareas ordenadas secuencialmente de arriba a abajo, la visión que se obtiene con el cronograma es similar a la de una escalera que desciende de manera irregular a medida que avanzan las semanas y que permite saber de un vistazo qué fases han supuesto un mayor esfuerzo y tiempo respecto a otras, como ha sido el caso del desarrollo de la primera de las herramientas desarrolladas para la transformación de PCN a smart contract y la elaboración de esta memoria.

5.3 Trabajos Futuros

En otro orden de ideas, y teniendo presente un enfoque de mejora continua, se han analizado a posteriori las herramientas presentadas en busca de puntos mejorables que pudieran verse reforzados mediante el planteamiento y desarrollo de funcionalidad extra. Tras una búsqueda de posibles mejoras adicionales, se proponen los siguientes **trabajos futuros**:

- Construcción de un **validador que compruebe si un smart contract puede convertirse en modelo PCN**. Actualmente, para convertir un smart contract en PCN hay que asegurar que la estructura de este cumpla el formato definido en el apartado 3.3. La construcción de un validador o un mecanismo que facilite la adopción de este formato concreto puede evitar problemas de uso y facilitar la utilización de la herramienta puente de smart contract a PCN.
- Ampliación de la herramienta puente de PCN a smart contract incluyendo un **asistente especializado en un tipo de modelo de negocio** (o varios). Por cada tipo de negocio a soportar existiría una batería de bloques de código Solidity prediseñados que implementen acciones comunes realizadas en las cadenas de procesos de esos modelos de negocio. Mediante una interfaz de usuario, se revisaría en cada paso del diagrama PCN si se quiere aplicar uno de estos bloques de código y de ser así, se aplicaría en la sección dedicada a escribir la lógica funcional de su función correspondiente del smart contract. Esto aumentaría casi al 100% el porcentaje de código generado en smart contracts resultantes

que pertenezcan a modelos de negocio soportados. Por ejemplo, podría incluirse el negocio de la restauración que incluya una batería de bloques que implementen lógica de entrega de menú, toma de pedido o encuesta de satisfacción, entre otros.

- Creación de un **diagramador online para modelar diagramas PCN**. Actualmente la forma de hacer diagramas PCN es mediante el plug-in dedicado a ello que incluye INNoVaServ, haciendo posible realizar esta tarea exclusivamente de manera local. Esto evitaría depender de una instalación local y se beneficiaría de las posibilidades de la nube permitiendo diseñar diagramas PCN y acceder a ellos en línea desde cualquier parte. El entorno adecuado para lograr esto podría ser Sirius Web.
- Ampliación de la herramienta puente de PCN a smart contract para permitir la **transformación de un diagrama PCN a otro tipo de smart contracts escritos en un lenguaje que no sea Solidity**. Los smart contracts generados con Solidity son utilizables en la red Ethereum y en otras redes compatibles con EVM, pero podría valorarse el uso de otros lenguajes como Rust o Haskell para permitir la compatibilidad con las redes blockchain Solana o Cardano.
- **Corrección del *layout* de diagramas PCN**. El plug-in de INNoVaServ incluye un diagramador que puede utilizarse para generar un diagrama PCN a partir de un modelo PCN. Actualmente, al generar dicho diagrama, la distribución de los elementos no sigue un formato de diseño adecuado y esto dificulta la interpretación del diagrama. Ajustar este aspecto del diagramador haría su experiencia de uso mucho más satisfactoria y útil.

Apéndice I - Tabla de Siglas

SIGLAS	DESCRIPCIÓN
BPMN	Business Process Modeling Notation
DSL	Domain Specific Language
EMF	Eclipse Modeling Framework
EVM	Ethereum Virtual Machine
GMF	Graphical Modeling Framework
M2M	Model To Model
M2T	Model To Text
MDE	Model Driven Engineering
PCN	Process Chain Network
SBP	Service Blueprint

Apéndice II - Manual de Usuario

En este apéndice, se presenta el manual de usuario con las instrucciones necesarias para hacer uso apropiado de las herramientas desarrolladas y presentadas en el Capítulo 3. Las herramientas se pueden obtener de su **repositorio** dedicado [40].

II.1 Creación de un proyecto Java en Eclipse

Un modelo PCN o SmaC requiere de la existencia de un proyecto creado y abierto previamente en el entorno Eclipse en el que ser ubicado. Si ya se dispone de un proyecto creado, puede omitir este apartado.

Para crear un proyecto en Eclipse, hay que hacer clic en el botón *File* de la barra de navegación de Eclipse para abrir un menú desplegable, después seleccionar la opción *New* para abrir un segundo menú desplegable y finalmente hacer clic sobre la opción *Java Project*, tal y como se muestra en la Figura 54.

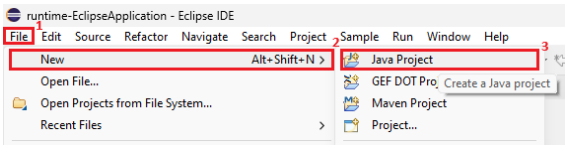


Figura 54. Iniciando la creación de un proyecto Java en Eclipse.

Se abrirá una nueva ventana en la que hay que poner el nombre deseado para el proyecto en el campo *Project name*, elegir un entorno de ejecución Java en la sección *JRE* y hacer clic en el botón *Finish* para terminar de crear el proyecto Java. Estos últimos pasos se ilustran en la Figura 55.

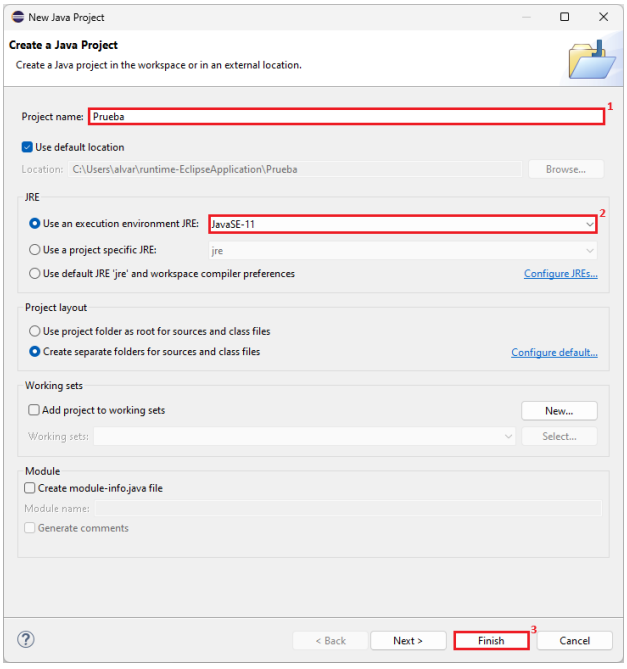


Figura 55. Creando un proyecto Java en Eclipse.

II.2 Creación de un modelo PCN

Una vez se dispone de un proyecto abierto en el entorno de trabajo, para crear un modelo PCN hay que hacer clic derecho sobre el proyecto en el que desea incluir el modelo PCN, seleccionar la opción *New* para abrir un segundo menú desplegable y hacer clic sobre el botón *Other*... como explica la Figura 56.

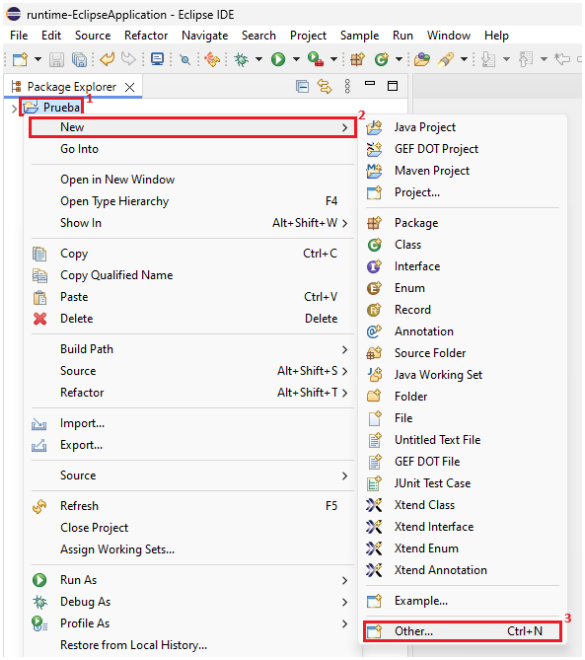


Figura 56. Iniciando la creación de un modelo.

Se abrirá otra ventana en donde basta con escribir "pcn" en el campo de texto para que se muestre en el panel la carpeta *Example EMF Model Creation Wizards* con la opción *Pcn Model*; se selecciona dicha opción y se hace clic en el botón *Next* >. A continuación se puede definir un nombre para el modelo PCN en el campo *File name* y después hay que volver a hacer clic en *Next* >. Por último, en el menú desplegable *Model Object*, hay que seleccionar la opción *PCN Diagram* y dar al botón *Finish* para terminar de crear el modelo. Estos pasos se ilustran en la Figura 57.

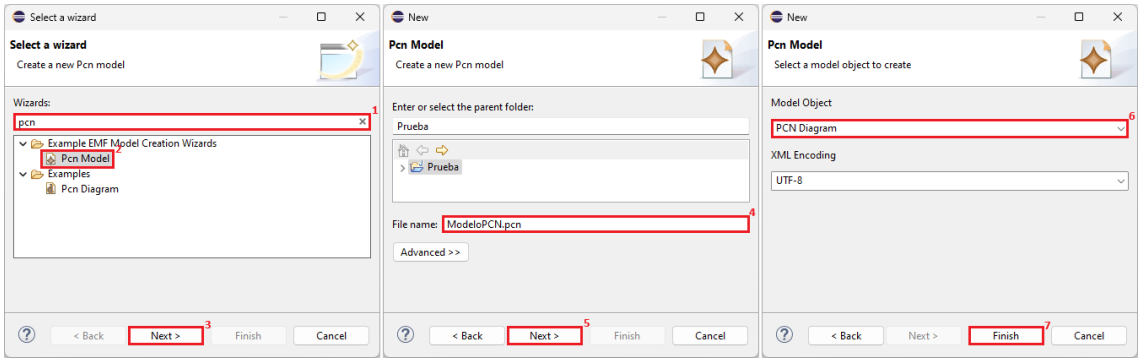


Figura 57. Creando un modelo PCN.

Tras haber creado el modelo PCN, este se abrirá en su vista de árbol. De manera opcional se puede crear un archivo de diagrama asociado al modelo que permite editar el modelo en un diagramador para hacer el proceso más gráfico y sencillo; para ello se puede hacer clic derecho sobre el modelo PCN en la pestaña *Package Explorer* o *Project Explorer* y dar al botón *Initialize pcn_diagram diagram file.* Cabe informar que un modelo PCN vacío es un modelo válido, ya que representa un diagrama PCN sin entidades.

⚠ A la hora de editar un modelo PCN, **se debe asegurar** que los campos necesarios de los distintos elementos que lo forman estén rellenos correctamente para evitar errores durante la generación de smart contracts. Ante cualquier duda, se puede hacer clic derecho en un espacio en blanco en la vista de árbol del modelo y seleccionar la opción *Validate* para validar el modelo y detectar posibles errores.

⚠ Debido a incompatibilidad, los modelos PCN que contengan elementos de tipo Reference **no se consideran válidos** para ser transformados en smart contract. Estos elementos se pueden sustituir por elementos de tipo Dependency.

Para más información, se recomienda consultar el manual de INNoVaServ [36].

II.3 Transformación de PCN a smart contract

Antes de poder transformar un modelo PCN en smart contract, es necesario disponer de un modelo PCN creado correctamente para esta finalidad. En el caso de no tener uno, habrá que crearlo desde cero (ver apartado II.2).

Para llevar a cabo la transformación, se deben seguir los pasos que se pueden ver en la Figura 58 haciendo clic derecho sobre el modelo PCN objetivo dentro de la pestaña *Package Explorer* y pulsando el botón *Generate Solidity code*.

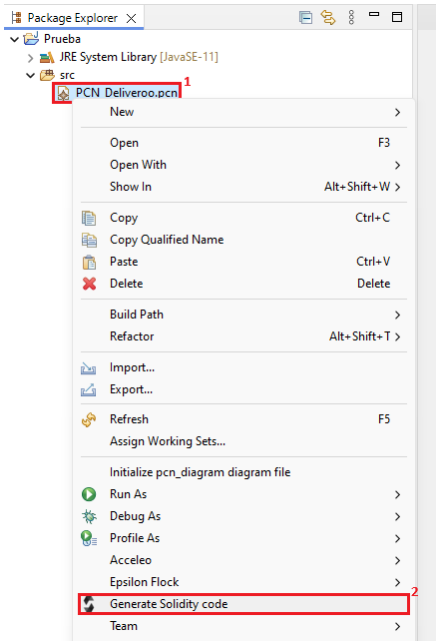


Figura 58. Iniciando la transformación de PCN a smart contract.

Tras esto, dará comienzo el proceso de generación del smart contract a partir del modelo PCN. Durante este proceso pueden aparecer distintas ventanas de interfaz en función de los elementos que se encuentren en el modelo PCN de origen. Las ventanas que aparezcan lo harán de manera secuencial y han de completarse; en caso de cerrar una ventana de interfaz, se utilizarán valores por defecto. Cada interfaz dispone de un botón de ayuda para ayudar a cumplimentar cada dato requerido.

Una vez finalizado el proceso cuando no aparezca ninguna interfaz en pantalla, se habrá creado una nueva carpeta llamada "sce-gen" dentro del proyecto. Como se observa en la Figura 59, en esta carpeta se encontrará el smart contract generado a partir del modelo PCN inicial en forma de modelo SmaC.

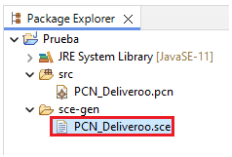


Figura 59. Fichero generado con la transformación de PCN a smart contract.

II.4 Creación de un smart contract (modelo SmaC)

Tras abrir un proyecto en el entorno de trabajo, para crear un modelo SmaC, hay que hacer clic derecho sobre el proyecto en el que se quiera incluir, seleccionar la opción *New* para desplegar un nuevo menú y luego haz clic en *Other...*, según se observa en la Figura 56.

Esto abrirá una nueva ventana en dónde se puede escribir "smac" en el campo de texto para mostrar en el panel la carpeta *Example EMF Model Creation Wizard* con la opción *SmaC Model*. Hay que seleccionar esta opción y hacer clic en el botón *Next >*. Después, se puede asignar un nombre al modelo SmaC en el campo *File name* y hay que volver a clicar en *Next >*. Para finalizar, se selecciona *File* en el menú desplegable *Model Object* y después se hace clic en *Finish*. Los pasos se muestran en la Figura 60.

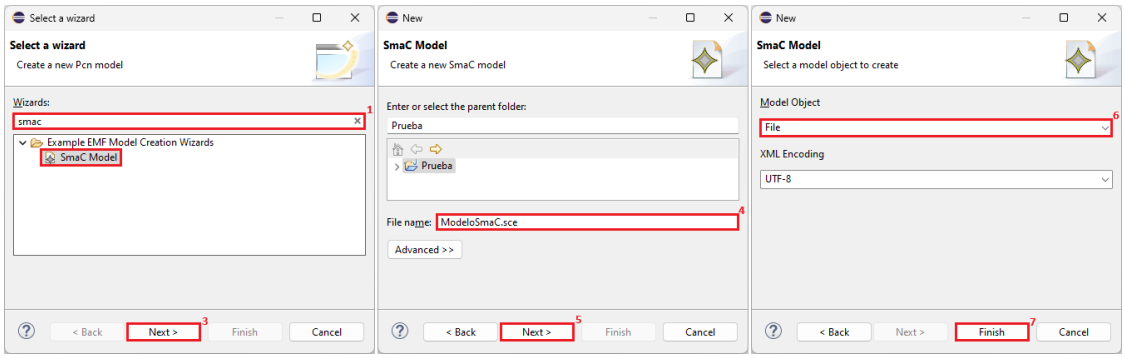


Figura 60. Creando un modelo SmaC.

Una vez creado el modelo SmaC, se abrirá el fichero del modelo textual en blanco listo para escribir en él un smart contract usando el lenguaje Solidity, por lo que es recomendable consultar la documentación oficial sobre el lenguaje [15].

⚠ Cuando se edite un modelo SmaC con el fin de utilizarlo para transformarlo en un modelo PCN, es **imprescindible** que el smart contract escrito se ajuste estrictamente a un formato muy específico definido para ese fin (ver apartado 3.3).

Para más información, es recomendable consultar el manual de SmaC [37].

II.5 Transformación de smart contract a PCN

Para llevar a cabo la transformación de un modelo SmaC en un modelo PCN, se requiere contar con un modelo SmaC creado que cumpla un formato específico. En caso negativo, será necesario crear uno (ver apartado II.4).

El primer paso es, en la pestaña *Package Explorer*, hacer clic derecho sobre el modelo PCN del que se desea generar un smart contract y en el menú desplegado clicar sobre el botón *Generate PCN Model*, tal y como se indica en la Figura 61.

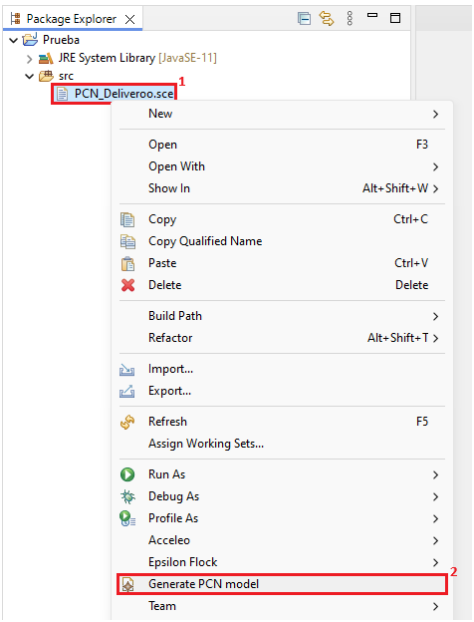


Figura 61. Iniciando la transformación de smart contract a PCN.

Esta acción finalizará con la creación inmediata de una nueva carpeta llamada "pcn-gen" dentro del proyecto que contiene el modelo SmaC de origen. En el interior de esta carpeta se encontrará el modelo PCN generado a partir del smart contract (ver Figura 62).

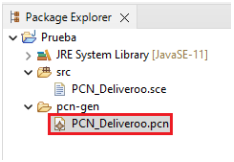


Figura 62. Fichero generado con la transformación de smart contract a PCN.

Bibliografía

[1] Banco Santander S.A. (11 de mayo, 2023). ¿Qué es "blockchain"?. *Banco Santander*. https://www.santander.com/es/stories/blockchain-que-es

[2] IBM (s.f.). ¿Qué son los contratos inteligentes en blockchain?. *IBM*. https://www.ibm.com/es-es/topics/smart-contracts

[3] Kazemzadeh, Y., Milton, S. K., & Johnson, L. W. (2015). *Process chain Network (PCN) and business process modeling notation (BPMN): a comparison of concepts*. Journal of Management and Strategy, 6(1), 88-99. https://doi.org/10.5430/jms.v6n1p88

[4] García Díaz, V., Núñez Valdez, E. R., Pascual Espada, J., Pelayo García Bustelo, B. C., Cueva Lovelle, J. M., & Montenegro Marín, C. E. (2014). A brief introduction to model-driven engineering. Tecnura, 18(40), 127-142. http://www.scielo.org.co/pdf/tecn/v18n40/v18n40a11.pdf

[5] Eclipse Foundation AISBL. (s.f.). Eclipse Modeling Framework. *Eclipse*. https://eclipse.dev/modeling/emf/

[6] Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. *Bitcoin.org*. https://bitcoin.org/bitcoin.pdf

[7] Ethereum. (7 de marzo, 2024). Introducción a los contratos inteligentes. *Ethereum*. https://ethereum.org/es/smart-contracts/

[8] Szabo, Nick. (1994). Smart Contracts. *Phonetic Sciences*. https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html

[9] UAH. (3 de junio, 2019). Historia de los Smart Conctracts. *UAH*. https://masterethereum.com/historia-smart-contracts/

[10] Fedorova, E. P., & Skobleva, E. I. (2020). *Application of blockchain technology in higher education*. European Journal of Contemporary Education, 9(3), 552-571. https://eric.ed.gov/?id=EJ1272331

[11] Cachin, C. (Julio, 2016). *Architecture of the hyperledger blockchain fabric*. In Workshop on distributed cryptocurrencies and consensus ledgers (Vol. 310, No. 4, pp. 1-4). https://cognizium.io/uploads/resources/IBM%20Research%20-%20Architecture%20of%20the%20Hyperledger%20Blockchain%20Fabric%20-%202016%20-%20July.pdf

[12] Howell, James. (1 de marzo, 2024). Top Examples Of Smart Contracts. *101 Blockchains*. https://101blockchains.com/top-smart-contracts-examples/

[13] Saez Gonzalez, Juan Antonio. (5 de marzo, 2024). Comparative de lenguajes de programación para Contratos Inteligentes (Smart Contracts). *Codemotion*. https://www.codemotion.com/magazine/es/backend-es/blockchain-es/comparativa-de-lenguajes-de-programacion-para-contratos-inteligentes-smart-contracts/

[14] Pascual, Juan Luis. (26 de mayo, 2022). Top 5 Lenguajes de Programación de Smart Contracts. *Bit2Me*. https://academy.bit2me.com/top-5-de-lenguajes-de-programacion-de-smart-contracts/

[15] The Solidity Authors. (s.f.). Documentation. *Solidity*. https://docs.soliditylang.org/en/v0.8.26/#

[16] SYDLE. (19 de septiembre, 2023). Procesos de negocio: ¿Qué son y cómo modelarlos? Ejemplos. *SYDLE*. https://www.sydle.com/es/blog/que-son-procesos-de-negocio-610afc74504afa7e3653c2c3

[17] E. Sampson, Scott, & M. Passey, James. (mayo, 2011). *Introduction to PCN Analysis*. Brigham Young University, Provo, Utah. https://studylib.net/doc/18502809/introduction-to-pcn-analysis---sampson-on-services

[18] E. Sampson, Scott, M. Passey, James, & Management Marriott School of Management. (5 de enero, 2015). *Essentials of Service Design and Innovation*. Brigham Young University, Provo, Utah. https://services.byu.edu/pcn/ESDI-4E_Chapters_1-6.pdf

[19] Maglio, P. P., Kieliszewski, C. A., Spohrer, J. C., Lyons, K., Patrício, L., & Sawatani, Y. (2019). *Handbook of Service Science, Volume II*. Cham, Switzerland: Springer International Publishing. https://link.springer.com/book/10.1007/978-3-319-98512-1

[20] Pearce, S. (2020). *Digital and Analogue interactions: Process Chain Networks for the Design of Service Processes*. School of Management University of Bristol, Bristol, UK. https://research-information.bris.ac.uk/ws/portalfiles/portal/244712747/MECO2020_Pearce_Final_1.0_Accept.pdf

[21] Gordijn, J., & Akkermans, H. (agosto, 2001). *Designing and Evaluating E-Business Models*. IEEE Intelligent Systems. http://dx.doi.org/10.1109/5254.941353

[22] Gordijn, J., & Akkermans, H. (abril, 2001). *e3-value : Design and Evaluation of e-Business Models*. IEEE Intelligent Systems. https://www.semanticscholar.org/paper/e-3-value-%3A-Design-and-Evaluation-of-e-Business-Gordijn-Akkermans/9e1ecab922e1e14d087dace850e13961c1c0820b

[23] Pombinho, J., Aveiro, D., & Tribolet, J. (Septiembre, 2014). *A Matching Ontology for e3Value and DEMO*. 2014 IEEE 16th Conference on Business Informatics. http://dx.doi.org/10.1109/CBI.2014.48

[24] Petit, M., Gordijn, J., & Wieringa, R. (septiembre, 2006). *Understanding business strategies of networked value constellations using goaland value Modeling*. 14th IEEE International Requirements Engineering Conference. https://ris.utwente.nl/ws/portalfiles/portal/5337561/RE06.pdf

[25] Da Silva, A. R. (2015). *Model-driven engineering: A survey supported by the unified conceptual model*. Computer Languages, Systems & Structures, 43, 139-155. https://doi.org/10.1016/j.cl.2015.06.001

[26] Lucio, L., Zhang, Q., Nguyen, P. H., Amrani, M., Klein, J., Vangheluwe, H., & Le Traon, Y. (2014). *Advances in model-driven security*. Advances in Computers (Vol. 93, pp. 103-152). Elsevier. https://doi.org/10.1016/B978-0-12-800162-2.00003-8

[27] Zschaler, S. (s.f.). Specific MDE technologies. *MDENet*. https://community.mde-network.org/posts/overview-of-learning-resources-specific-mde-technologies

[28] Banco Santander S.A. (11 de mayo, 2023). ¿Qué es "blockchain"?. *Banco Santander*. https://www.santander.com/es/stories/blockchain-que-es

[29] Eclipse Foundation AISBL. (s.f.). Eclipse Modeling Framework (EMF). *Eclipse*. https://eclipse.dev/modeling/emf/

[30] Solus S.A. (2007). Guía de Usuario de Enterprise Architect 7.0. *Sparx Systems*. http://www.sparxsystems.com.ar/download/ayuda/index.html?importexport.htm

[31] Sirius. (s.f.). What is Sirius?. *Eclipse*. https://eclipse.dev/sirius/overview.html

[32] Acceleo. (s.f.). What is Acceleo?. *Eclipse*. https://eclipse.dev/acceleo/overview.html

[33] Itemis. (s.f.). Xtext. Itemis. https://www.itemis.com/en/it-services/methods-and-tools/xtext

[34] Ruiz Rube, I. (2013). *Desarrollo de editores textuales con Xtext* [Presentación de PowerPoint]. OCW Universidad de Cádiz. https://ocw.uca.es/pluginfile.php/2499/mod_resource/content/0/P6%20-%20Desarrollo%20de%20editores%20textuales%20con%20Xtext.pdf

[35] Efftinge, S., & Spoenemann, M. (s.f.). Xtend – Documentation. *Eclipse*. https://eclipse.dev/Xtext/xtend/documentation/

[36] Pérez Blanco, F. J. (2022). *INNoVaServ: un entorno de modelado para el diseño de servicios* [Tesis doctoral, Universidad Rey Juan Carlos]. BURJC-Digital. https://hdl.handle.net/10115/20701

[37] Gómez Macías, C. (2023). *SmaC: a model-based framework for the Development of smart contracts* [Tesis doctoral, Universidad Rey Juan Carlos]. BURJC-Digital. https://hdl.handle.net/10115/32765

[38] Gómez Macías, C., Pérez Blanco, F. J., Vara, J. M., De Castro, V., & Marcos, E. (2021). *Design and development of Smart Contracts for E-government through Value and Business Process Modeling*. Proceedings of the 54th Hawaii International Conference on System Sciences, 2021. https://aisel.aisnet.org/cgi/viewcontent.cgi?article=1527&context=hicss-54

[39] Pérez-Blanco, F. J., Vara, J. M., Gómez, C., De Castro, V., Granada, D., & Marcos, E. (2019). *Acercando modelos de negocio y de proceso para el diseño de servicios*. Actas las 24th Jornadas Ing. del Softw. y Bases Datos, JISBD 2019. https://biblioteca.sistedes.es/dspace.server/api/core/bitstreams/598179d8-2def-4b1a-9503-bc981029ce1e/content

[40] Barrio Luquero, Á.. (noviembre, 2024). *PCN-SmartContract-Bridges* [Repositorio en GitHub]. GitHub. https://github.com/alv4rob/PCN-SmartContract-Bridges