Prototipado automático de sistemas de información transaccionales usando una especificación en lenguaje natural restringido

Jean Pierre Alfonso Hoyos

Universidad Nacional de Colombia
Facultad de Ingeniería
Departamento de Ingeniería de Sistemas e Industrial
Bogotá, Colombia
2017

Prototipado automático de sistemas de información transaccionales usando una especificación en lenguaje natural restringido

Jean Pierre Alfonso Hoyos

Tesis presentada como requisito parcial para optar al título de:
Magíster en Ingeniería de Sistemas y Computación

Director:
Felipe Restrepo Calle, Ph.D.

Línea de Investigación:
Ingeniería de software - Lenguajes de programación
Grupo de Investigación:
PLaS - Programming Languages and Systems

Universidad Nacional de Colombia
Facultad de Ingeniería
Departamento de Ingeniería de Sistemas e Industrial
Bogotá, Colombia
2017

A mis padres que con su dedicación diaria y apoyo incondicional hicieron esto posible.

Resumen

Las actividades del ciclo de vida del desarrollo de software (o SDLC por sus siglas en inglés) incluyen: análisis de requisitos, diseño de modelos, desarrollo, pruebas y mantenimiento. Las tareas tempranas de este ciclo (análisis de requisitos y diseño) tienen un amplio impacto en el éxito del proyecto y por esto es fundamental que se lleven a cabo de una forma correcta.

Estas actividades por supuesto están incluidas en el proceso de desarrollo de sistemas de información de procesamiento de transacciones. Estos sistemas de información son una de las maneras de generar valor desde la información producida en una organización. Además, dan pie para generar sistemas de información de mayor complejidad y también permiten mejorar los procesos de toma de decisiones en las organizaciones.

Sin embargo, los errores en las etapas tempranas del desarrollo de software son bastante comunes. Estos errores pueden llevar a dificultades a nivel de presupuesto y calendario en los proyectos de desarrollo de software, inclusive, a fracasos totales.

Por esta razón, en esta tesis se propone diseñar, desarrollar y evaluar una metodología para el prototipado automático de sistemas de información transaccionales desde una especificación en lenguaje natural restringido. Lo cual busca mejorar los procesos de análisis de requisitos y diseño de modelos puesto que permitiría validar rápidamente la funcionalidad del software, y así, facilitar la detección de errores y por ende su corrección temprana durante el desarrollo del proyecto.

Para esto, en esta tesis se propone una metodología de prototipado rápido basada en un lenguaje natural restringido. Para crear este lenguaje natural restringido se usan como insumo dos lenguajes de especificación populares: BPMN (Business Process Modeling Notation) y E-R (Entity - Relationship). Además, para la generación del prototipo funcional, se usarán técnicas de generación de código fuente guidas por la sintaxis de este lenguaje.

Como resultados de esta tesis, se llevaron a cabo dos implementaciones de la herramienta de generación de código fuente. Además, se presentan tres casos de estudio que permiten validar la aplicabilidad y efectividad de la metodología propuesta: “Question cycle”, “Email Voting” y “Odoo clone”.

Palabras clave: requisitos de software, BPMN, E-R, sistemas de información transaccionales, prototipado rápido, generación de código fuente, metodología de prototipado, lenguaje natural restringido, automatización de construcción de software.

Abstract

Software development life cycle (or SDLC) activities include: requirements analysis, models design, development, testing and maintenance. The early tasks of this cycle (requirements analysis and design) have a large impact on the success of the project and for this reason it is essential to perform them correctly.

These activities are of course included in the process of developing transaction processing information systems. These information systems are one of the ways to generate value from the information produced in an organization. They also provide the basis for generating more complex information systems and also improve the decision-making processes in organizations.

However, errors in the early stages of the software development process are quite common. These errors can lead to difficulties in the budget and schedule of software projects, or even, total failures.

For this reason, the objective of this thesis is to design, develop and evaluate a methodology for the automatic prototyping of transactional information systems from a restricted natural language specification. This is aimed at improving the processes of analysis of requirements and design of models since it would allow to quickly validate the software functionality, and thus, facilitate the detection of errors and reduce costs by correct them early during the development.

With that goal in mind, this thesis proposes a rapid prototyping methodology based on a restricted natural language. To create this restricted natural language, two popular specification languages are used as input resources: BPMN (Business Process Modeling Notation) and E-R (Entity - Relationship). In addition, source code generation techniques guided by the syntax of this language will be used for the generation of the functional prototype.

As results of this thesis, two implementations of the source code generation tool were developed. In addition, three case studies were performed to validate the applicability and effectivity of the proposed methodology: “Question cycle”, “Email Voting”, and “Odoo clone”.

Keywords: software requirements, BPMN, E-R, information systems, rapid prototyping, source code generation, prototyping methodology, restricted natural language, software construction automation

Contenido

Lista de Figuras

Lista de Tablas

1 Introducción

En esta tesis se presentará el sustento teórico, desarrollo y validación de una herramienta de software capaz de generar prototipos de sistemas de información transaccionales desde una especificación en lenguaje natural restringido. Para este fin, se realizará una revisión de la literatura relacionada con la problemática que concierne al desarrollo de esta tesis, se propondrá un método para diseñar un lenguaje restringido, se implementará una herramienta con el resultado de este método y se explorarán algunos casos de estudio para mostrar el alcance de la herramienta.

1.1. Motivación

Los sistemas de información transaccionales son "sistemas computarizados que hacen y almacenan las transacciones de rutina necesarias para conducir el negocio" de acuerdo a [Laudon and Laudon, 2012]. Estos sistemas son una de las primeras maneras de generar valor desde la información producida en una organización [Laudon and Laudon, 2012]. Estos sistemas se desarrollan normalmente utilizando alguna metodología de desarrollo. Independientemente de cualquier tipo de metodología que se use, esta encaja en el ciclo de vida del desarrollo software (SDLC por sus siglas en inglés).

Como se muestra en la figura **1-1**, durante el SDLC se incluyen las actividades de: análisis de requisitos, diseño, desarrollo, pruebas y mantenimiento. Durante la fase de análisis de requisitos se llevan a cabo actividades como la elicitación de requisitos, refinamiento, selección y control de requisitos con el fin de determinar las capacidades del producto desarrollar. Posteriormente se construyen modelos con el fin de comenzar la traducción desde los requisitos hasta llegar a un programa de computador [Davis, 1990]. Estos pasos (requisitos y diseño) tienen un alto impacto en el éxito del proyecto. Los errores cometidos en estas primeras etapas de desarrollo pueden tener impacto negativos en el presupuesto y la duración del proyecto [Walia and Carver, 2009].

Durante la fase de elicitación de requisitos se pueden presentar errores debido a la in-

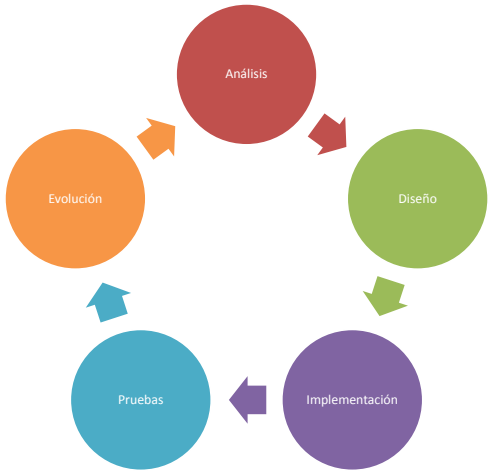


Figura 1-1: Ciclo de vida del software

completitud y la ambigüedad propias del lenguaje natural en el que estos requisitos se escriben [Walia and Carver, 2009]. La falta de conocimiento del ingeniero de requisitos sobre el dominio de conocimiento específico del negocio y la falta de comunicación con el cliente también pueden introducir errores en el producto de software final [Walia and Carver, 2009, Fairley, 2009]. Para superar estas dificultades, se han propuesto varias aproximaciones incluyendo diferentes formas de restringir la forma en que se escriben los requisitos de los sistemas [Bhatia et al., 2013], La automatización de la construcción de modelos a partir de textos sin restringir [Nishida et al., 1991, Ibrahim and Ahmad, 2010] y diversas metodologías de desarrollo [Fairley, 2009, Kroll and Kruchten, 2003].

La fase de diseño del SDLC puede ser vista como una traducción de los requisitos describiendo el software a modelos desarrollados usando algún lenguaje de especificación como UML (Unified Modeling Language), BPMN (Business Process Model Notation), o diagramas E-R (Entity-Relationship). Esta tarea se realiza de forma manual con la ayuda de alguna herramienta CASE (Computer Aided Software Engieneering) como Rational Rose[1] o Visual Paradigm[2]. Es por esto que esta fase requiere de personal con el conocimiento de las reglas del negocio propias del cliente y al mismo tiempo con conocimientos que le permitan desarrollar modelos de estas reglas.

Las metodologías de desarrollo de software han sido ampliamente usadas para controlar el proceso de desarrollo de software de una manera sistemática. Para proyectos grandes, metodologías como RUP (Rational Unified Process) son usadas para gestionar el SDLC en grandes grupos de trabajo y obtener software de alta calidad [Kroll and Kruchten, 2003].

[1] http://www-03.ibm.com/software/products/en/enterprise
[2] https://www.visual-paradigm.com/

Sin embargo, usando metodologías de desarrollo tradicional como RUP, el desarrollo es lento y requiere completar un ciclo de desarrollo completo para poder validar el software [Kroll and Kruchten, 2003, Fowler and Highsmith, 2001]. Por esto han aparecido metodologías ágiles de desarrollo, estas resultan mas apropiadas para equipos pequeños y software de pequeña-mediana escala, dejando de lado proyectos de gran escala y de misión crítica [Fowler and Highsmith, 2001, Githens, 2006].

El prototipado rápido de software es otra forma de afrontar algunos de los problemas anteriormente descritos, estas aproximaciones intentan hacer el SDLC más rápido, menos sensible a fallos humanos, a la ambigüedad del lenguaje natural y a la incompletitud de las especificaciones, por tanto el prototipado rápido de aplicaciones puede reducir tiempo y costos de desarrollo [Harris, 2012]. En muchos casos (como en esta tesis) estas especificaciones están escritas en lenguaje natural (restringido y sin restringir) del cual se extrae un código, un modelo o un programa ejecutable. Esto permite lograr una validación rápida del software propuesto y una retroalimentación más directa de los clientes.

1.2. Definición del problema

Las organizaciones que pretendan empezar a conseguir valor de la información que producen deben implementar sistemas de información, pero el costo de desarrollarlos puede ser proporcional a la complejidad de la organización. Las organizaciones de mediana envergadura carecen de sistemas que se ocupen de sus operaciones específicas y esto puede deberse a los altos costos y tiempos que conlleva su desarrollo.

En las metodologías de desarrollo de software que se usan para construir estos sistemas (RUP o SCRUM), hay una tarea o fase en la que el sistema deseado es descrito usando lenguaje natural. Esta tarea es seguida de la creación de un modelo y de su posterior codificación. Estos procesos son lentos y propensos a errores humanos [Leffingwell, 2011, Augustine and Martin, 2005]. Los errores cometidos en estos procesos de recolección de requisitos y especificación del sistema resultan críticos porque tienen alto impacto en el presupuesto y calendario del proyecto.

Para amortiguar los problemas causados por estos errores, una aproximación posible es realizar prototipos de manera veloz para validar directamente con el cliente sus necesidades o confirmar la factibilidad del proyecto. La principal razón para usar prototipado rápido es que estos son mucho más baratos de realizar que el producto final.

Parte de los métodos de prototipado rápido incluyen procesamiento de textos como fuente de la información de diseño. Estos métodos tienen ventajas a nivel de comunicación con los interesados del cliente y simplicidad en el manejo de estos. Los métodos identificados en este trabajo usan como insumo para su trabajo una especificación realizada en lenguaje natural con y sin restricciones.

Una primera aproximación usa técnicas de aprendizaje de máquina y procesamiento de lenguaje natural para "extraer" características de diseño de un texto de forma libre. Estos procesos usan la información de las categorías morfosintácticas de las palabras en la oración, información estructural de la oración, relaciones entre términos explicitas o implícitas, conocimiento predefinido y varias herramientas más para lograr la creación de un modelo.

Una segunda aproximación utiliza plantillas para escribir listas de requisitos de un sistema para posteriormente procesarlas usando técnicas de compiladores y patrones de especificación en un lenguaje objetivo. Estos recursos se emparentan para lograr generar un modelo que posteriormente podrá ser convertido automáticamente a código.

Finalmente, en la última aproximación se crea un lenguaje restringiendo partes de la gramática de un lenguaje natural. Este nuevo lenguaje puede procesarse usando técnicas de compiladores o usando Prolog. Cabe resaltar que estos lenguajes son un modelo por sí mismo y no requieren un paso intermedio para convertirse en código fuente, como si sucede en las dos aproximaciones anteriormente descritas.

La mayoría de los trabajos realizados en esta área no obtienen un ejecutable y los modelos generados están aún muy alejados de sistemas funcionales. Esto se debe a que los métodos utilizados en estos trabajos (técnicas de minería de datos, plantillas o lenguajes) no son aptos para la tarea de esta tesis, esto puede deberse a que tratan de ser de propósito general (y no de propósito específico como en esta tesis), o también, a que no se han perfeccionado lo suficiente las herramientas de procesamiento de lenguaje natural.

Esta tesis de maestría propone una metodología para el prototipado automático de sistemas de información transaccionales. La metodología se basa en una herramienta computacional que genera una aplicación web con las capacidades propias de estos sistemas. Este tipo de sistemas de información encaja en este trabajo debido a su naturaleza: manipula entidades para organizar la información, sin realizar estadísticas, ni operaciones complejas sobre los datos.

1.3. Objetivos

Objetivo general:

- Diseñar, desarrollar y evaluar una metodología para el prototipado automático de sistemas de información transaccionales desde una especificación en lenguaje natural restringido.

Objetivos específicos:

1. Revisar el estado del arte de los enfoques existentes para el desarrollo o modelado automático de software desde una especificación en lenguaje natural.
2. Proponer una metodología para generar automáticamente prototipos de sistemas de información transaccionales desde una especificación en lenguaje natural restringido.
3. Desarrollar una herramienta de software implementando la metodología propuesta.
4. Evaluar la solución propuesta.

1.4. Alcance

En resumen, lo que se pretende en esta tesis es construir una metodología que permita la generación automática de prototipos funcionales de sistemas de información. Usando como insumo un lenguaje natural restringido. La idea principal es incluir un único diseñador/desarrollador que conozca el nuevo lenguaje de especificación de requisitos (que se desarrollará en esta tesis) en una sesión con los clientes. De esta sesión, resultará una especificación escrita en este nuevo lenguaje, y a partir de esta, se generará automáticamente el código de una aplicación ejecutable como se muestra en la figura **1-2**.

1.5. Estructura de esta tesis

El resto de este documento está dividido de la siguiente manera: en el Capítulo 2 se encuentra el marco teórico y una revisión de los trabajos relacionados; en el Capítulo 3 se encuentra al descripción de la metodología que cumple el objetivo de la tesis; en el Capítulo 4 se describen las dos implementaciones que se realizaron de la herramienta computacional para soportar la metodología propuesta; posteriormente, en el Capítulo 5 se presentan tres

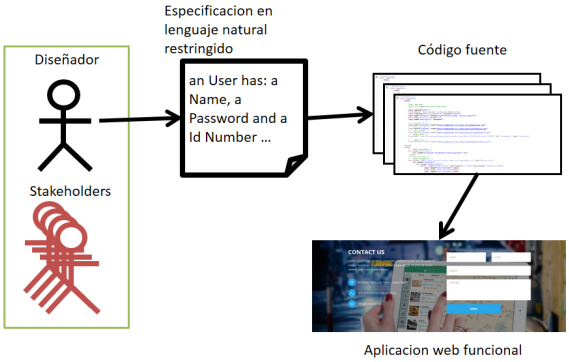


Figura 1-2: Flujo de trabajo propuesto para esta metodología

casos de estudio para validar la aplicabilidad y la efectividad de la metodología propuesta; luego, en el Capítulo 6 se discuten los resultados obtenidos; y por último, en el Capítulo 7 se presentan las conclusiones de esta tesis y el trabajo futuro.

2 Marco conceptual

En este capítulo se exponen los conceptos principales y la introducción a la teoría utilizada para el desarrollo de esta tesis. También se presentan los trabajos relacionados con esta propuesta con el fin de poder evaluar las contribuciones de la misma.

Teniendo en cuenta que el objetivo general de esta tesis es "diseñar, desarrollar y evaluar una metodología para el prototipado automático de sistemas de información transaccionales desde una especificación en lenguaje natural restringido", en la sección 2.1 se presenta una introducción a las tipologías de sistemas de información; en la sección 2.2 se explica el ciclo de vida del desarrollo de software (SDLC); en la sección 2.3 se definen y detallan los requisitos de software, ya que constituyen el punto de partida de la metodología propuesta en esta tesis; en la sección 2.4 se presentan algunos lenguajes de especificación representativos que se usan como insumo para el desarrollo del lenguaje de requisitos de software que se propone en esta tesis. Además, puesto que el procesamiento de algunos lenguajes de especificación se hace basándose técnicas tradicionales de compiladores, en la sección 2.5 se hace una introducción a los conceptos principales de estas técnicas. Por último, en la sección 2.6 se presentan los trabajos relacionados que se revisaron para el desarrollo de esta tesis, mostrando una clasificación realizada basándose en sus características principales.

2.1. Sistemas de información

Para Laudon [Laudon and Laudon, 2012] un sistema de información "es un conjunto de componentes interrelacionados que recolecta, procesa almacena y distribuye información para apoyar los procesos de toma de decisiones y de control en una organización. Los sistemas de información también pueden ayudar a los gerentes y trabajadores del conocimiento a analizar problemas, visualizar temas complejos y crear nuevos productos". Cabe resaltar que en esta definición no se abarcan únicamente los componentes computacionales.

Los sistemas de información permiten tres operaciones sobre los datos: la recolección, el procesamiento y la salida; la recolección es tomar los datos en bruto desde su origen, el procesamiento es convertir estos datos a un formato significativo y la salida es cuando la información sale del sistema para que el actor que toma decisiones proceda con su labor.

Los sistemas de información son actualmente una parte vital de las organizaciones, por ejemplo en la actualidad el negocio de los reportes crediticios o de los bancos serían inmanejables sin los sistemas apropiados.

En la actualidad los sistemas de información se apoyan en sistemas computacionales o tecnologías de la información las cuales incluyen: el software, la tecnología de almacenamiento, el hardware, las redes; también, algunos de estos sistemas, en ocasiones, son delegados a servicios de terceros tales como: Amazon web services[1] o Google cloud plataform[2].

Estos sistemas son valiosos en las organizaciones porque proveen mejoras que se reflejan en la productividad, en el aumento de ingresos, o tal vez, un posicionamiento estratégico superior de la empresa en el mercado, todo esto apoyado en que mejoran los procesos de toma de decisiones e innovación.

El paradigma actual de los sistemas de información está basado en los procesos de negocio, los cuales se refieren a "la forma en que se organiza, coordina, y enfoca el trabajo para producir un producto o servicio valioso" también pueden definirse como "el conjunto de actividades requeridas para crear un producto o servicio", esto ha dado pie a que aparezcan soluciones como Bizzagi[3] para computarizar estos sistemas.

[1] https://aws.amazon.com/
[2] https://cloud.google.com/
[3] www.bizagi.com

2.1.1. Tipología de los sistemas de información

Clasificación basada en el grupo objetivo dentro de la organización:

1. **Sistemas de procesamiento de transacciones (TPS)**: los TPS son los sistemas que llevan registro de las actividades y transacciones elementales de la organización (ventas, flujo de materiales, compras, etc...).

2. **Sistemas de información gerencial (MIS)**: los MIS responden a los diagnósticos y reportes que se hacen sobre los datos transaccionales. Están enfocados a la gerencia media que necesita tomar decisiones administrativas. Esta información es útil para controlar la organización además de cuantificar y predecir el desempeño.

3. **Sistemas de soporte de decisiones (DSS)**: los DSS brindan apoyo en la toma de decisiones no rutinaria, se enfocan en problemas únicos y sin solución predefinida. Estos sistemas tratan de responder con análisis de efectos en cambios al interior de la organización. Muchas veces requieren de información externa a la organización para completar los análisis de sensibilidad.

4. **Sistemas de apoyo a ejecutivos (ESS)**: los ESS ayudan a la gerencia de nivel superior a tomar decisiones con implicaciones de alto impacto en la organización. Muestran gráficos resumen que le permiten a estas personas planificar el futuro de la organización.

Clasificación basada en la función: se clasifica basándose en el área funcional en el que opera dentro de la organización.

1. **Sistemas de planificación de recursos empresariales (ERP)**: son sistemas que integran los procesos de manufactura, producción, finanzas, contabilidad, ventas, marketing y recursos humanos en un solo repositorio central con el fin reutilizar información que se fragmentaba en muchos sistemas.

2. **Sistemas de administración de la cadena de suministro**: se usan para administrar las relaciones con los entes externos a la organización como los son proveedores, estos exponen datos de las organizaciones para facilitar y hacer más eficiente los procesos de inventarios y entregas.

3. **Sistemas de administración de relaciones con el cliente (CRM)**: los CRM asisten la administración de la relación con los clientes, con el fin de optimizar los procesos de venta y mercadeo.

4. **Sistemas de administración del conocimiento (KMS)**: los KMS permiten a las

organizaciones basadas en conocimiento administrar y aplicar mejor estos conocimientos para obtener valor a largo plazo.

Para esta tesis, los sistemas que se implementarán con ayuda de la metodología de prototipado que es objetivo de esta tesis, son los sistemas de procesamiento de transacciones. La principal razón que motivó esta elección se fundamenta en que este tipo de sistemas es la base para todas operaciones que se llevan a cabo en otros tipos de sistemas de información.

2.2. Ciclo de vida del desarrollo de software

El ciclo de vida del desarrollo de software, o SDLC por sus siglas en inglés, es una estructura de tareas que se usa para organizar el desarrollo de software. El SDLC se encuentra incluído en el estándar *Systems and software engineering - Software life cycle processes (ISO IEC 12207:2008)* [IEEE, 2008]. Este estándar establece un marco de trabajo común que contiene procesos, actividades y tareas para ser aplicados durante cualquier proceso de adquisición de software o cualquier proceso de su ciclo de vida (desarrollo, operación, mantenimiento o desecho) [IEEE, 2008]. Este además de los procesos propios del software también incluye procesos de contexto en la organización en la que se pretende implementar el software. El mapa de procesos de este estándar se muestra en la figura **2-1**.

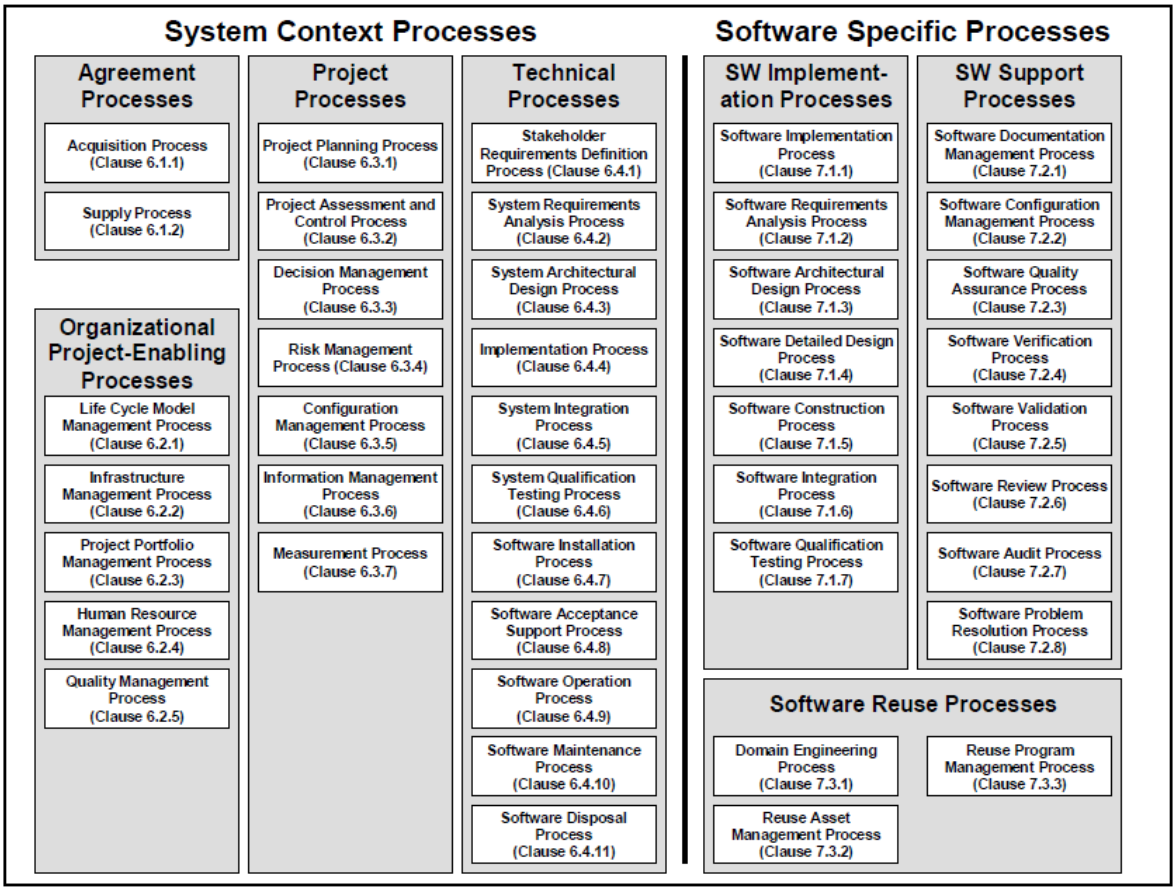


Figura 2-1: Procesos ISO IEC 12207:2008 p. 14

La parte de los procesos en la figura **2-1** que representa mejor el modelo en la figura **1-1** es la parte de procesos específicos del software. Esto suponiendo que se hará el desarrollo como parte del proceso de implementación de software (*sw implementation process*). El diagrama

del SDLC mostrado en la figura **1-1** es la maqueta general para todos los procesos que contienen desarrollo de software sin tomar en cuenta procesos previos de mercadeo ni de estrategia.

El SDLC puede constar de un número diferente de fases dependiendo de la interpretación que se le de y el alcance del proyecto. Puede incluir únicamente las 5 fases mostradas en la figura **1-1** hasta todos los procesos mostrados en la figura **2-1**. Por esta razón hay varias interpretaciones de este ciclo que incluyen otras fases tales como: puesta en marcha, calificaciones de rendimiento y manejo de riesgo [Turpin, 1996, Dagnino, 2002, Kaur and Sengupta, 2010, Mittal and Chopra, 2011, Sivaji et al., 2013]. Entre las actividades más comunes encontradas en los modelos de SDLC están:

- Las actividades de análisis de requisitos incluyen la recolección de los requisitos de las personas involucradas o interesadas *(stakeholders)*, su análisis, clasificación y viabilidad técnica. Esta es tradicionalmente la fase donde más interacción se requiere con el cliente.

- Las actividades de diseño incluyen el desarrollo de modelos del sistema desde diferentes perspectivas y con diferentes niveles de profundidad. Estos modelos tradicionalmente se hacen usando lenguajes de especificación basados en modelos como los incluidos en UML (Unified Modeling Language) (secuencias, estados, clases, casos de uso, casos de prueba, despliegue, interacción, etc ...) o como BPMN (Bussiness Process Model Notation), E-R (Entity-Relation). Lenguajes de especificación formal como Z, VDM++ o B son usados en proyectos de misión critica. En esta fase se decide la arquitectura del software y del sistema en general.

- La fase de implementación es en la que se reúnen las salidas de las fases anteriores para escribir el código fuente de la aplicación deseada. Es la fase donde el producto realmente es construido.

- La fase de pruebas (o pruebas y evaluación) es en la que el software se somete a la calificación funcional y no funcional de manera interna y externa a la organización que desarrolló el software. Puede incluir pruebas automáticas o contratar un servicio de pruebas/certificaciones externo.

- La fase de evaluación (evolución, ejecución, mantenimiento o puesta en marcha) es la fase en la que el software es usado por el cliente final y los nuevos requisitos son obtenidos para mejorar la calidad y alcance del software.

A continuación se mostrará cómo estas fases están presentes en metodologías de desarrollo

populares como RUP y SCRUM. También se hará una introducción al prototipado rápido de sistemas.

2.2.1. RUP y el SDLC

El proceso racional unificado o RUP (por sus siglas en inglés de *Rational Unified Process*) es una metodología de análisis, diseño, implementación y documentación de software [Kroll and Kruchten, 2003]. Se apoya fuertemente en la notación UML y en la documentación para producir una serie de artefactos asociados a las actividades del SDLC, y al igual que éste, es un proceso iterativo. RUP divide el SDLC en 4 fases: Iniciación, Elaboración, Construcción y Transición. Estas fases son muy similares a las que recomiendan en PMBOK [Kroll and Kruchten, 2003, Project Management Institute, 2013].

Durante estas 4 fases RUP ejecuta unos flujos del trabajo con intensidad variable. Estos flujos de trabajo son en parte los procesos de [IEEE, 2008] por supuesto conteniendo las fases del SDLC. Las fases y procesos de RUP se muestran en la figura **2-2**.

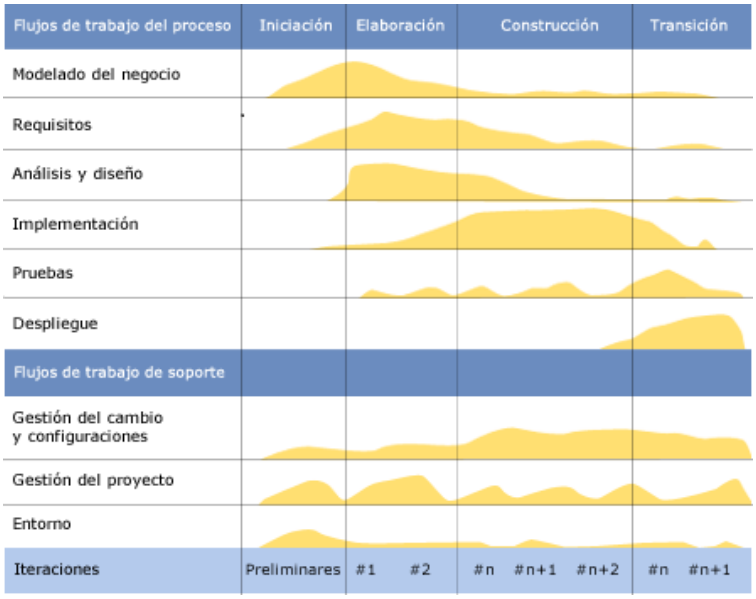


Figura 2-2: Fases y flujos de RUP

2.2.2. SCRUM y el SDLC

SCRUM es un marco de trabajo para el desarrollo de software iterativo e incremental basado en el trabajo [Takeuchi and Nonaka, 1986]. SCRUM define una manera no secuencial

de desarrollar un producto de software. Este marco de trabajo asume que los requisitos del sistema son volátiles, es decir que pueden cambiar en cualquier momento.

SCRUM divide el trabajo en *sprints* que son esfuerzos acotados en el tiempo (generalmente dos semanas). Estos esfuerzos contienen planificación, desarrollo, pruebas y retrospectiva. Un diagrama de proceso del *sprint* de SCRUM se muestra en la figura **2-3**[4].

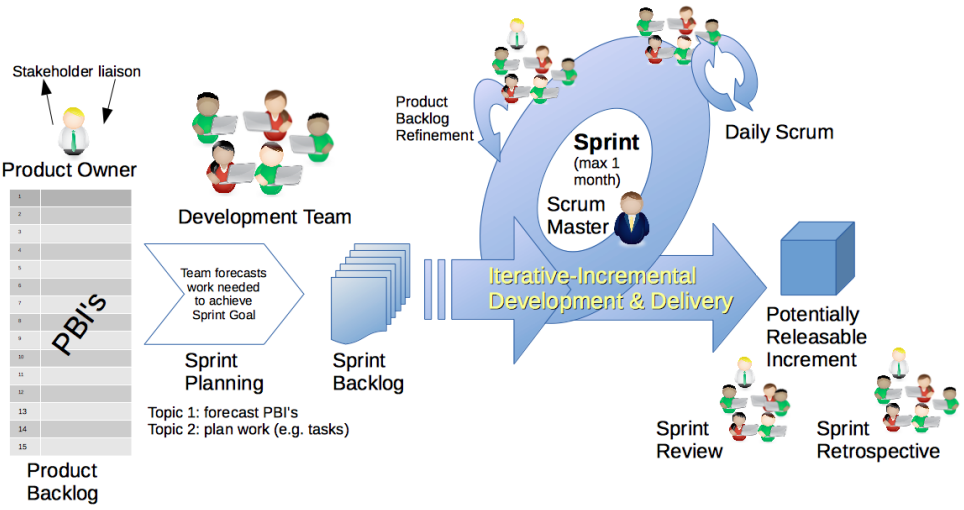


Figura 2-3: Sprint de SCRUM

Aunque en SCRUM es más difícil ver las fases del SDLC, podemos ver que durante el sprint un ciclo del SDLC es realizado: las historias de usuario corresponden con los requisitos [Leffingwell, 2011], el diseño e implementación se hacen durante el sprint. Las pruebas y el lanzamiento también se realizan durante este periodo de tiempo [Augustine and Martin, 2005].

En este marco de trabajo se realizan todas las fases del SDLC de manera continua y sobrepuesta, no se separan en el tiempo como en el caso de RUP o de la metodología de cascada [Mitchell and Seaman, 2009]. También se confía en el equipo y sus capacidades de auto organización. No se mantienen ni crean muchos documentos para soportar el proceso. Se estimula el uso de repositorios de código y procesamientos automáticos (a nivel de pruebas). Se intenta mantener la transparencia del proceso de desarrollo y ser abiertos con los problemas que se encuentren [Fowler and Highsmith, 2001].

[4]Tomado de http://en.wikipedia.org/wiki/Scrum_(software_development)

2.2.3. Prototipado rápido de sistemas

El prototipado rápido es un tipo de desarrollo en el que se impulsa el desarrollo de prototipos cuanto antes en el SDLC para permitir realimentación y análisis rápidamente [Kordon and Luqi, 2002]. Este tipo de desarrollo es necesario dado que la complejidad creciente de los sistemas inducen errores constantemente. Algunos de estos errores pueden ser:

- En las fases tempranas del desarrollo de software, los requisitos no quedan claros por una de las partes y esto induce errores (como veremos en la sección 2.3.2).
- En las fases posteriores pueden presentarse errores de integración.
- El mantenimiento puede producir errores fácilmente, puesto que puede incluir un cambio de equipo de desarrollo o evidenciar deuda técnica no pagada.

La principal razón para usar el prototipado rápido es que este es económicamente eficiente. Esto permite descartar rápidamente la funcionalidad que no es requerida y detectar los fallos en los sistemas de una manera oportuna.

El prototipo representa un subconjunto de características escogidas para ser implementadas y validadas. Estos prototipos son útiles a la hora de validar la factibilidad técnica del sistema propuesto y ayuda a encontrar posibles dificultades futuras. El prototipado rápido busca generar estos prototipos de manera veloz.

Este prototipo puede contener sólo un subconjunto de las características propuestas y puede requerir características completamente diferentes (lenguajes, capacidades de hardware) a los que necesitaría el producto final. Los prototipos pueden ser clasificados en dos categorías:

1. **La aproximación desechable:** el prototipo esta hecho con el fin de ser desechado.
2. **La aproximación de evolución:** cada prototipo desarrollado contribuye al producto final, hasta que el último prototipo es el producto final.

Para este último se requieren herramientas de software especializadas, como por ejemplo Bizzagi[5], que usa una interfaz gráfica y algunos lenguajes de especificación con el fin de crear un sistema de información; openXava[6], que crea una aplicación web partiendo de una definición de clases de java; o justInMind[7], que crea prototipos interactivos para aplicaciones móviles y páginas web usando interfaces de tipo desplazar y soltar.

[5] https://www.bizagi.com/
[6] http://openxava.org
[7] https://www.justinmind.com/

2.3. Requisitos de software

La recolección y comprensión de los requisitos de un sistema de información es la parte inicial de los procesos desarrollo de software. Estos requisitos toman diferentes formas y nombres dependiendo de la metodología que se pretenda implementar durante el desarrollo del producto. Para la metodología que se propondrá más adelante en esta tesis esto no es diferente.

Los requisitos, la mayor parte del tiempo, están redactados en lenguaje natural y se pueden ver como el primer modelo del sistema a construir. El uso de técnicas para validar y construir esta lista de requisitos también se ha investigado y se presenta más adelante.

2.3.1. Definición

Un requisito [ISO/IEC and IEEE, 2010] (3.2506) según la IEEE es:

1. Una condición o capacidad necesitada por un usuario para solucionar un problema o alcanzar un objetivo.
2. Una condición o capacidad que debe ser cumplida o poseída por un sistema, componente de sistema, producto o servicio para satisfacer un acuerdo, estándar, especificación, u otros documentos formalmente impuestos.
3. Una representación documentada de una condición o capacidad como en los numerales 1 y 2.
4. Una condición o capacidad que debe ser cumplida o poseída por un sistema, producto, servicio, o componente para satisfacer un contrato, estándar, especificación, u otros documentos formalmente impuestos. Los requisitos incluyen las necesidades, deseos y expectativas cuantificadas y documentadas de los patrocinadores, clientes y otros stakeholders.

Estos requisitos por lo general toman forma usando una representación en lenguaje natural. La figura **2-4** corresponde a una representación de un requisito de una metodología de desarrollo como RUP.

Por otra parte, el requisito mostrado en la figura **2-5** corresponde a una historia de usuario, redactado como si el usuario mismo lo hiciera y con partes bien definidas como: el rol del usuario, una meta y una razón.

La base de datos deberá soportar la generación y control de objetos de configuración; estos son elementos que son agrupamientos de otros objetos en la base de datos. las facilidades del control de configuración deberán permitir el acceso a los objetos en un grupo de versiones con el uso de un nombre incompleto.

Figura 2-4: Ejemplo de requisito en RUP

yo como usuario administrador, puedo especificar archivos o carpetas para respaldar basándome en el tamaño de archivo, fecha de creación y fecha de modificación.

Figura 2-5: Ejemplo de historia de usuario

Adicionalmente los requisitos deben poseer un conjunto de características para considerarse buenos [Berenbach et al., 2009]:

1. **Posible**: que su implementación sea posible en la plataforma planeada.
2. **Válido**: que deba implementarse en la plataforma, que no sea opcional.
3. **No ambiguo**: que tenga una única interpretación.
4. **Verificable**: que el producto final pueda ser probado para verificar que cumple el requisito.
5. **Modificable**: que al ser cambiado no modifique de manera amplia los requisitos que coexisten con él en la especificación.
6. **Consistente**: que no contradiga o entre en conflicto con otro requisito.
7. **Completo**: que contenga la información suficiente para lograr construir el sistema.
8. **Trazable**: que sea posible seguir el requisito durante el ciclo de vida de éste y el de la aplicación.

Este conjunto de características hacen menos riesgosos los procesos posteriores y por esto es muy importante una buena calidad en los requisitos [Davis, 1990]. Como se puede ver en la sección 2.3.2, los errores en los requisitos tienen consecuencias importantes y comprometen el éxito del proyecto. Adicionalmente, estrategias como las presentadas en la sección 2.3.3, son propuestas para mitigar estos efectos.

2.3.2. Errores en los requisitos

En el artículo [Walia and Carver, 2009] podemos encontrar varias cosas interesantes en cuanto se refiere a errores en los requisitos. En este se presenta una taxonomía que clasifica los errores en tres posibles categorías:

- Errores de gente: son errores que se originan de fallos en las personas individuales involucradas en el proyecto
  - Comunicación
  - Participación
  - Conocimiento del dominio
  - Aplicación específica del conocimiento
  - Ejecución del proceso
  - Otros de cognición
- Errores de procesos: son causados por errores en la forma de alcanzar los objetivos y se concentran principalmente en los procesos inadecuados en la ingeniería de requisitos.
  - Método inadecuado de alcanzar objetivos
  - Gerencia
  - Elicitación
  - Análisis
  - Trazabilidad
- Errores de documentación: son errores en la organización y la especificación sin tomar en cuenta si los desarrolladores los entendían
  - Organización
  - No usar un estándar
  - Especificación

Como podemos ver gran parte de los errores detectados en este estudio son causados por fallas en las personas y sus relaciones. También se incluyen errores en la forma en que los requisitos en si mismos son escritos y analizados por los desarrolladores.

Más tipos de errores son mostrados en este estudio, como por ejemplo, errores a nivel de cultura y políticas, funcionalidad oculta y fallos por presión de los stakeholoders que llevan a dejar funcionalidades por fuera del producto final.

En este estudio se muestran también los errores que pueden ocurrir en otras fases de la fase de elicitación y que también pueden ocurrir en esta:

- Información perdida: falta de entendimiento del dominio de conocimiento.
- Resbalones en el diseño del sistema: malentendidos en el estado actual y meta del sistema resultando en una serie de acciones mal encaminadas.
- Programas de sistema: de tecnología, organización, históricos y de otras causas.
- Desglose cognitivo: mucha atención o muy poca llevando a fallos en la acción.

Por otra parte, en [Ferrari et al., 2016] se analizan las posibles ambigüedades que se pueden producir en las entrevistas con los stakeholders. Presenta las siguientes cuatro categorías de ambigüedades:

Falta de claridad: son situaciones en las que el analista de requisitos no puede dar ninguna interpretación o significado aceptable a la unidad de información expresada.

Entendimiento múltiple: son situaciones en las que el analista de requisitos puede darle múltiples interpretaciones aceptables a la expresión que el cliente, una correcta y varias incorrectas.

Desambigüacion incorrecta: son situaciones en las que el analista de requisitos asigna una sola interpretación a la expresión del cliente, pero el significado de esta es diferente a la que el cliente pretendía.

Desambigüacion correcta (ambigüedad inconsciente): es un fenómeno teórico en el que se encuentran situaciones en las que el analista puede asignar más de una interpretación al fragmento de discurso del cliente, pero la única aceptable para el analista es la que empata con el significado del discurso del cliente.

En [Ferrari et al., 2016] también se hace un recorrido por los diferentes tipos de ambigüedades que se encuentran en documentos de requisitos únicamente:

- Léxicas: términos con más de un significado
- Sintácticas: más de un árbol de sintaxis por frase
- Semánticas: la frase tiene más de un significado en su contexto

- Pragmáticas: el significado de una frase depende del contexto en el que se coloque
- Inducidas por vaguedad: ningún análisis puede asignar un significado preciso a los términos.
- Inducidas por generalidad: la frase puede ser especificada mas precisamente.

Por último, en [Kaur and Sengupta, 2011] se muestran una serie de estadísticas que muestran el impacto de los fallos en la etapa de requisitos en el éxito de los proyectos de desarrollo de software, por ejemplo:

- TCS (Tata Consultancy Services) 2007: Reportó que el 67 % de organizaciones experimentaron extensiones en el calendario del proyecto, 49 % hicieron extensiones en el presupuesto y 41 % fallaron en entregar el valor agregado propuesto inicialmente.
- Avanade Research Report (2007): 66 % fallan por la especificación del sistema, 51 % fallaron por no entender los requisitos, 49 % por la tecnología seleccionada.
- ESSU (European Service Strategy Unit) Research Report 2007: 57 % experimentaron retrasos, 33 % sufrieron sobrecostos, 30 % fueron cancelados.

Estas estadísticas muestran lo complicado de llevar un proyecto de software en el tiempo estimado, sin sobre-costos y con la funcionalidad deseada.

2.3.3. Estrategias para mitigar errores

Varias estrategias se han generado para mitigar los errores en los requisitos como por ejemplo:

- Utilizar software [Li et al., 2005a].
- Utilizar plantillas para los requisitos [Fatwanto, 2012b].
- Utilizar lenguajes restringidos [Fockel and Holtman, 2015, Aiello et al., 2014].
- Utilizar traducciones a lenguajes formales [Yan et al., 2015].
- Utilizar procesamiento de lenguaje natural para mejorar la calidad de los requisitos [Arora et al., 2015, Dzung and Ohnishi, 2013].
- Utilizar metodologías [Soares and Moura, 2015].

Según [Ferrari et al., 2016] las aproximaciones que usan algún tipo de procesamiento de

lenguaje natural se pueden dividir en dos clases:

Lenguaje natural restringido y lenguajes (semi-)formales: estos lenguajes tiene la ventaja de no ser ambiguos pero debe dedicárseles tiempo para ser aprendidos.

Lenguaje natural sin restricciones: cada una de estas aproximaciones pueden lidiar con un solo tipo de problemas (por ejemplo determinar la validez del uso de la palabra "todos" o "solo" en un requisito). También pueden usar procesamiento de lenguaje natural para buscar patrones comúnmente erróneos o evidencia estadística de que puede haber alguna inconsistencia.

2.4. Lenguajes de Especificación

Según el IEEE en su estandar [ISO/IEC and IEEE, 2010] un lenguaje de especificación (3.2827) es: "un lenguaje, generalmente una combinación de un lenguaje natural y uno formal, usado para expresar requisitos, diseño, comportamiento u otras características del componente de un sistema". Además un lenguaje de especificación de requisitos (3.2519) es: "un lenguaje de especificación con constructos especiales y a veces protocolos de verificación usados para desarrollar, analizar y documentar requisitos de software o hardware".

A continuación se muestran algunos de los lenguajes de especificación que se usan en este trabajo o que de alguna manera han inspirado el desarrollo de esta tesis.

2.4.1. Z

Z es un lenguaje de especificación formal. Las especificaciones escritas en Z modelan un sistema: coloca nombres a los componentes y expresa las restricciones entre estos componentes. El significado de una especificación en Z es definido como el conjunto de interpretaciones (valores para los componentes nombrados) que son consistentes con las restricciones [ISO and IEC, 2002].

Z usa notación matemática, por esto se dice que las especificaciones en este lenguaje son formales: el significado es capturado las notaciones matemáticas usadas, independientemente de los nombres usados. Esta base formal permite razonamiento matemático, y en consecuencia pruebas de que las propiedades deseadas son consecuencias de la especificación. La robustez de las reglas de inferencia usadas en este razonamiento debe ser probada relativa a la semántica de Z [ISO and IEC, 2002].

Algunas características particulares de Z son:

- Su kit extensible para notaciones matemáticas.
- Su notación para especificar estructuras en los sistemas y para estructurar la especificación en si misma.
- Su sistema de tipos definible, que permite algunos chequeos automáticos de buena construcción de la especificación.

Z está construido para ser usado en sistemas de misión crítica, algunos ejemplos de sistemas en los que Z se ha implementado son:

- Sistemas de seguridad crítica: señalamiento de vías de trenes, equipos médicos, y sistemas de energía nuclear.
- Sistemas de seguridad, como sistemas de procesamiento de transacciones y de comunicaciones.
- Sistemas generales como lenguajes de programación y procesadores de punto flotante.

Un ejemplo clásico de una especificación hecha en Z se muestra en la figura **2-6**.

$$[NAME, DATE]$$

BirthdayBook

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$
$$known = dom\ birthday$$

Figura 2-6: Ejemplo de especificación hecha en lenguaje Z

2.4.2. Gellish

Gellish es un lenguaje de especificación formal diseñado por Andries Van Renssen en su tesis de doctorado [Renssen, 2005]. Este lenguaje trata de ser independiente del lenguaje natural. Cada una de las variantes de Gellish (Inglés, Danés) es un subconjunto de algún lenguaje natural. Al mismo tiempo cada cosa, expresión o concepto es representado por un número (UIDs de Gellish), esto permite la traducción entre variantes de Gellish [Renssen, 2005].

Este lenguaje funciona haciendo declaraciones formadas principalmente por una tripla del tipo < Término, Relación, Término > en la cual cada termino denota una entidad u objeto y la relación es la que existe entre esos dos términos. Elementos adicionales son agregados para generar la Tabla Gellish completa, estos elementos son los UID de cada término y de cada relación, también son las unidades de medida que pueden aparecer [Renssen, 2005].

Gellish también puede ser usado para expresar varios tipos de hechos como:

- Conocimiento, resultando en modelos de conocimiento.
- Especificaciones estándar.
- Productos individuales.

En la tabla **2-1** se puede ver como se representa una entidad carro, con algunas definiciones de su color.

Id	**Objeto de mano izquierda**	**Id de la relación**	**Nombre de la relación**	**Objeto de la mano derecha**
1	Carro	564564	puede tener como aspecto un/a	color
2	rojo	533434	es un/a	color
1	Carro rojo	3545464	es	rojo
1	Carro rojo	878787	es un	carro

Tabla 2-1: Ejemplo de tabla Gellish

Las entidades y relaciones de Gellish se mapean a un número usando un diccionario de entidades y relaciones. Esto para permitir la traducción automática entre los diferentes lenguajes controlados de Gellish.

2.4.3. UML (Unified Modeling Language)

El objetivo de UMl es proveer a los arquitectos de sistemas, ingenieros de software y desarrolladores de software con herramientas para el análisis, diseño e implementación de sistemas basados en software, negocios o procesos similares [OMG, 2010].

La sintaxis abstracta de UML es especificada usando un modelo de UML llamado el *metamodelo* UML. Este metamodelo usa constructos de un subgrupo de UML. Esto es que UML está definido en términos de si mismo [OMG, 2010].

Un modelo es siempre un modelo de *algo*, la cosa siendo modelada puede ser considerada generalmente como un sistema en algún dominio del discurso. El modelo entonces hace algunas afirmaciones de interés sobre el sistema, abstrayendo de todos los posibles detalles del sistema, desde un punto de vista y para cierto propósito. Para un sistema existente, el modelo puede representar un análisis de las propiedades y comportamientos del sistema. Para un sistema en planificación, puede representar una especificación de cómo el sistema debe estar construido y como debe representarse [OMG, 2010].

UML separa sus posibles modelos en dos áreas semánticas: estructurales y de comportamiento. Una jerarquía de los posibles modelos de UML es mostrado en la figura **2-7**.

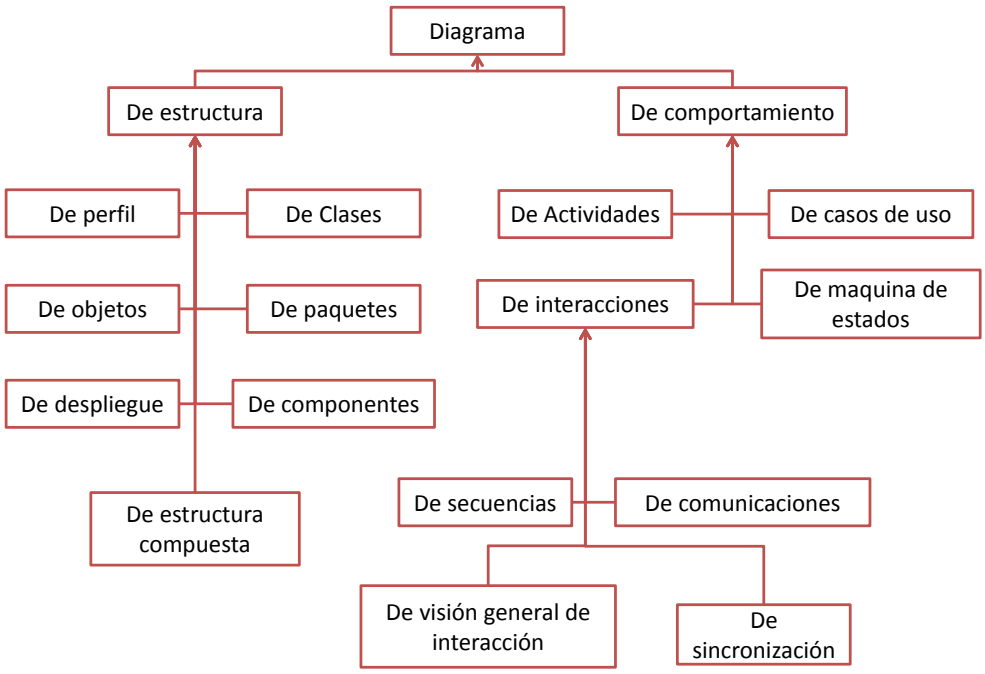


Figura 2-7: Árbol de jerarquías de diagramas UML

Diagrama de clases

Definidos en el capítulo 9 de [OMG, 2010] como "Clasificaciones", son diagramas que muestran relaciones entre conceptos o clases que agrupan instancias (o valores) de determinado tipo en un sistema. Cada clase se representa como un cuadro nombrado, una serie de atributos de determinado tipo, una serie de métodos con determinados parámetros y determinados tipos de retorno y con una serie de enlaces a otras clases que determinan las relaciones que existen en el sistema entre estas dos entidades. Un ejemplo de este tipo de diagramas se muestra en la figura **2-8**[8].

En el ejemplo de la figura **2-8** se pueden observar 11 clases entre ellas: *Customer, Administrator, User, Department y Category*. Cada una con una lista de parámetros que abstraen este tipo de entidades en el dominio de aplicación y una serie de métodos que indican las acciones que realiza cada instancia del tipo en el sistema. En este ejemplo existe una relación de generalización/especificación entre las clases User y Customer, la cual significa que *Customer* es *User*. También hay una relación de agregación entre *Customer* y *Order* la cual significa que Un *Customer* tiene *Orders*. Los números a los extremos de cada enlace significan

[8]Tomado de: http://people.cs.ksu.edu/ reshma/798_ClassDiagram.htm

la restricción numérica en la relación. Por ejemplo, cada *Order* debe tener una *ShippingInfo*.

Estos diagramas son útiles a la hora de modelar: las entidades de dominio que conformaran el modelo de datos de un sistema de información, como en el ejemplo **2-8**; las clases que conformarán el código fuente de una aplicación orientada a objetos; las entidades que se relacionan en un negocio; UML en si mismo está definido en términos de estos diagramas [OMG, 2010].

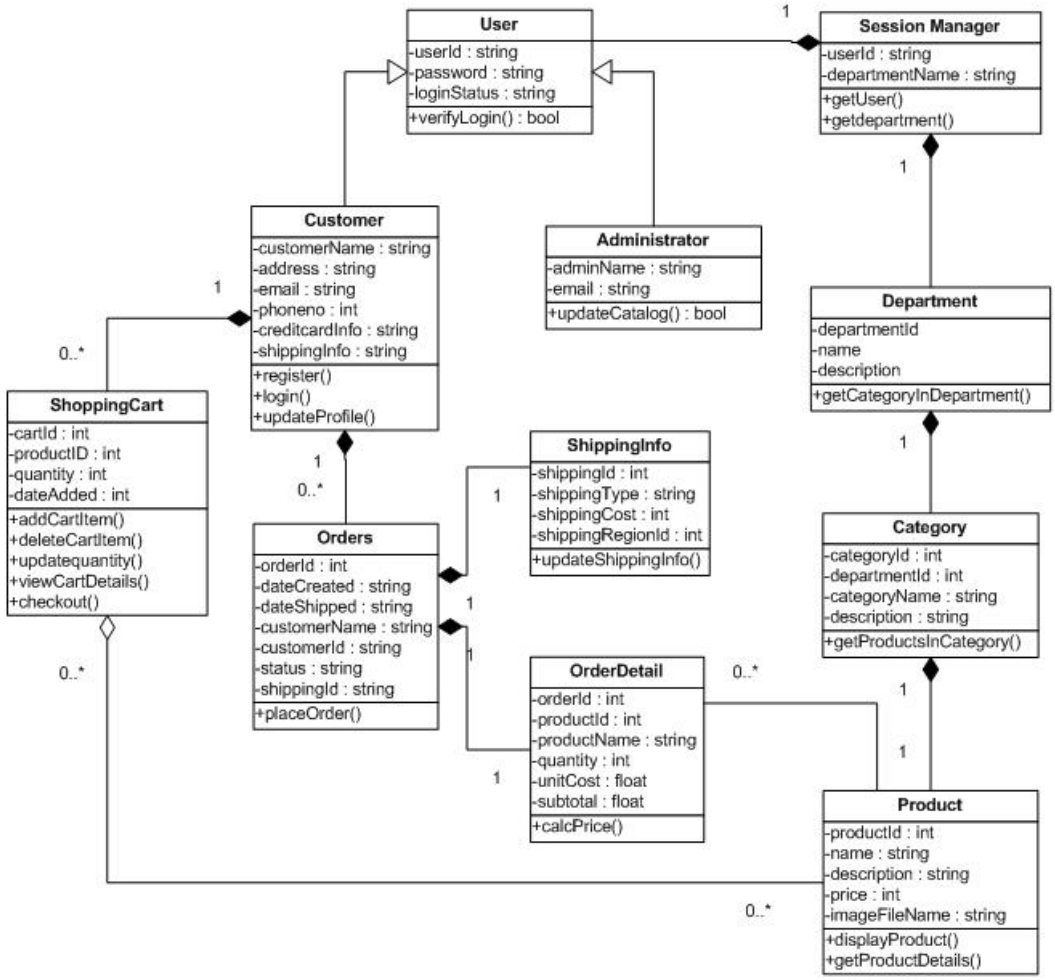


Figura 2-8: Ejemplo de diagrama de clases de UML

Diagrama de estados

Definidos en el capítulo 14 de [OMG, 2010] como un conjunto de conceptos que pueden ser usados para describir comportamientos discretos impulsados por eventos usando un formalismo de máquinas de estado finito. Estas máquinas son similares a los autómatas de estado finito que se pueden encontrar en el análisis léxico [Aho et al., 2007].

Pueden ser usados principalmente para dos fines:

- Expresar comportamientos de partes del sistema (por ejemplo de una clase), estas máquinas son llamadas *de comportamiento*.
- Expresar las secuencias de interacciones válidas (protocolos) para partes del sistema, estas máquinas son llamadas *de protocolo*.

Estos diagramas cuentan con una serie de elementos: un estado inicial; un estado final; una serie de estados intermedios; una serie de transiciones disparadas por cierto evento en el sistema y una serie de acciones a ejecutar durante la ejecución de un estado. Estos diagramas también pueden contener elementos como regiones y sub-máquinas. Un ejemplo de este tipo de diagramas se encuentra en la figura **2-9**.

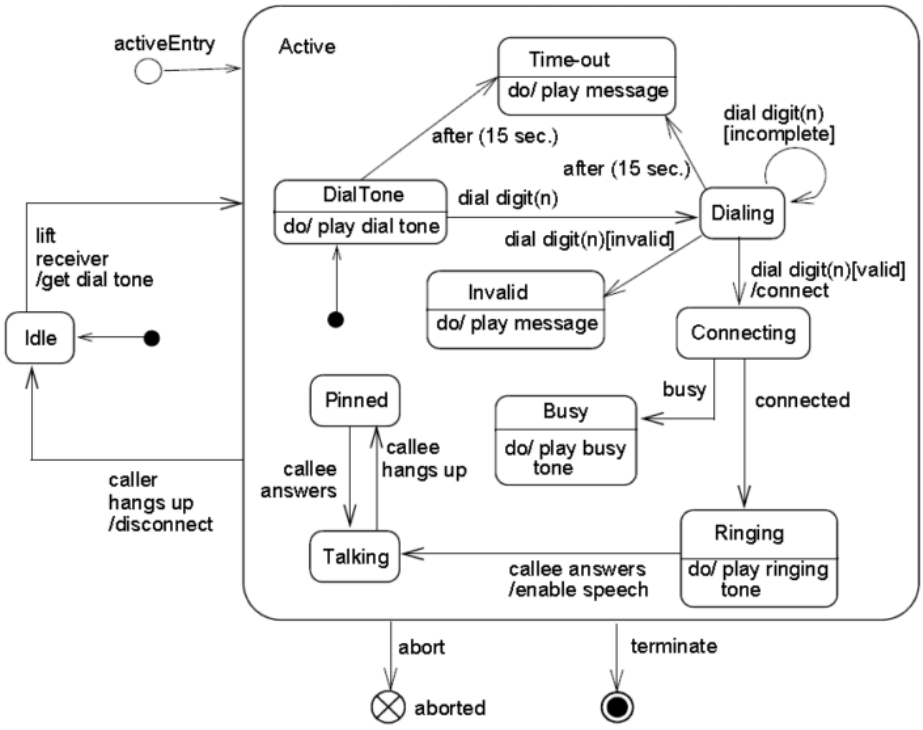


Figura 2-9: Ejemplo de diagrama de estados de UML representando un teléfono

En este ejemplo podemos ver elementos como los estados que son los rectángulos redondeados y las transiciones que son las flechas que hay entre cada estado. También podemos ver el concepto de sub-máquina que es el estado "Active" definido a su interior como una maquina de estados.

Diagrama de secuencia

Definidos en el capítulo 17.8 de [OMG, 2010] como un diagrama de interacción que describe una secuencia de mensajes que son intercambiados por una serie de *lifetimes*.

Los diagramas de secuencia cuentan con una serie de elementos para su construcción, los más comunes incluyen: lifetimes (*Computer, Server*), frames y messages (CheckEmail, SendUnSent, NewEmail, response, deleteOldEMail). La figura **2-10** presenta un ejemplo de un diagrama de secuencia.

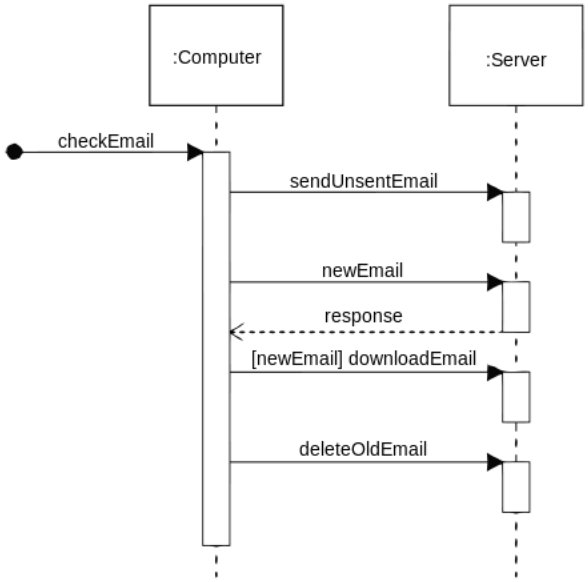


Figura 2-10: Ejemplo de diagrama de secuencia UML

2.4.4. BPMN (Bussines Process Model and Notation)

Según [Object Management Group (OMG), 2011], el objetivo primordial de BPMN (Bussiness Process Model and Notation) es proveer una notación que sea clara para todos los usuarios del negocio, desde los analistas de negocio que crean los bosquejos iniciales del

proceso, a los desarrolladores técnicos responsables por implementar la tecnología que desarrollará esos procesos, y finalmente, a la gente de negocios que manejará y revisará estos procesos. Entonces BPMN crea un puente estandarizado para la brecha entre el diseño de los proceso de negocio y su implementación.

Otra meta de BPMN es asegurar que los lenguajes XML diseñados para la ejecución de procesos de negocio como WSBPEL (Web Services Business Process Execution Language) puedan ser visualizados con una notación orientada al negocio.

Este lenguaje de especificación es usado para comunicar a una amplia variedad de información a una amplia variedad de audiencias. BPMN está diseñado para cubrir varios tipos de modelado y permite la creación de procesos de negocio "end-to-end".

Este lenguaje de especificación cuenta con 3 tipos de diagramas: de colaboración, de procesos, y de coreografía. Se profundizará únicamente en la semántica de los diagramas de proceso. Las categorías de elementos de BPMN son 5 con sus respectivas subdivisiones:

- Objetos de flujo
  - Eventos
  - Actividades
  - Compuertas
- Datos
  - Objetos de datos
  - Entradas de datos
  - Salidas de datos
  - Almacenamiento de datos
- Objetos de conexión
  - Flujos de secuencia
  - Flujos de mensaje
  - Asociaciones
  - Asociaciones de datos
- *Swimlanes*

- *Pools*
- *Lanes*

- Artefactos
  - Grupos
  - Anotaciones de texto

En la tabla **2-2** se hará una breve descripción de algunos de estos elementos con el fin de mostrar la notación gráfica que se usa para este lenguaje. Estos elementos se usan en los diagramas de proceso principalmente.

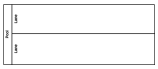
Elemento	Descripción	Diagrama
Evento	Los eventos "suceden" durante la ejecución de un proceso. Estos eventos afectan en flujo del modelo y usualmente tienen una causa o "trigger" o un impacto o "result". Existen 3 tipos dependiendo de como afectan el flujo: de comienzo, de finalización, e intermedios.	
Actividad	Es un tipo genérico de trabajo que una organización realiza.Una actividad puede ser atómica o no atómica. Los tipos de tareas que se encuentran en un diagrama de procesos son: sub-procesos y tareas.	
Compuertas	Las compuertas son usadas para controlar la divergencia y convergencia de la secuencia de actividades. Estas pueden ser: "branching", "forking", "merging" y "joining".	
Flujos de secuencia	Se usa para mostrar el orden de las actividades en el proceso.	
Flujos de mensaje	Se usa para mostrar un flujo de mensajes entre dos participantes (Pools) que están listos para enviarlos y recibirlos.	
Pools	Es una representación gráfica de un participante, cumple la función de partir gráficamente una serie de tareas. Puede tener detalles del proceso o simplemente dejarse en blanco	
Lanes	Es una partición de un pool en medio de un proceso. Los lanes se usan para categorizar actividades, por ejemplo, en el rol en el que se ejecutan al interior de una organización.	En la fila anterior

Tabla 2-2: Elementos principales de BPMN

Además, cada uno de los elementos anteriormente descritos tiene una serie de representaciones diferentes para denotar: en el caso de los eventos, los posibles tipos (en la tabla **2-3**) de eventos y sus diferentes "triggers" (en la tabla **2-4**); en el caso de las compuertas los diferentes tipos de compuertas (en la tabla **2-5**).

Evento	Descripción
	Señala el punto de partida del proceso
	Señala cualquier evento que ocurra entre el inicial y el final del proceso
	Señala el punto de finalización del proceso

Tabla 2-3: Tipos de eventos en BPMN

Elemento	Descripción	Elemento	Descripción
	llega un mensaje		Sucede alguna hora, minuto o fecha
	Alguien de mayor rango se involucra en el proceso		Se cumple alguna regla o norma de negocio
	Un sub-proceso es parte de un proceso mayor		Un error es capturado
	Una transacción ha sido cancelada		Las operaciones fallan parcialmente
	Una señal es capturada		Varios tipos diferentes de eventos
	Varios tipos de eventos, todos deben haber sucedido para continuar		Termina inmediatamente el proceso y todas las instancias asociadas

Tabla 2-4: Tipos de símbolos en los eventos de BPMN

2.4.5. E-R (Entity-Relationship)

Los modelos E-R (Entidad-Relación) describen una serie de tipos inter-relacionados en un dominio de conocimiento específico. En ingeniería de software estos modelos son creados para representar cosas que se necesitan para ejecutar los procesos de negocio [Chen, 1976]. En consecuencia este modelo representa un modelo de datos abstracto. Estos modelos pueden ser automáticamente implementados en una base de datos relacional.

A diferencia de los otros lenguajes de especificación señalados anteriormente, los modelos E-R son suficientes para lograr implementar la base de datos de manera completa. Un ejemplo

Elemento	Descripción	Elemento	Descripción
	Evalúa el estado del proceso y basado en esta evaluación divide el flujo en flujos mutuamente exclusivos		igual que la exclusiva pero basando en los eventos ocurridos durante la ejecución del proceso
	parte el flujo en varios flujos sin importar el estado del proceso		divide el flujo en uno o varios flujos dependiendo de cierta condición
	Por cada aparición de evento inicia una nueva instancia del proceso		una condición especial o varias puertas anidadas
	similar a la compuerta paralela, pero con los procesos son dependientes de los eventos		

Tabla 2-5: Tipos de compuertas en BPMN

de un modelo entidad relación es mostrado en la figura **2-11**[9]. En este ejemplo se puede observar las entidades que son las cajas, los atributos que son el contenido de esta caja. y las relaciones que son las líneas que unen estas cajas. Estos modelos pueden ser obtenidos automáticamente de una definición de clases hecha en una diagrama de clases de UML. Se usa generalmente un sistema ORM (Object-Relational mapping) para esta tarea.

2.4.6. WebML (ahora IFML)

WebML o *Web Modeling Language* es un lenguaje de especificación diseñado para soportar aplicaciones web complejas que hacen uso intensivo de datos [Ceri et al., 2000]. Posteriormente fue extendido para soportar mayor cantidad de interfaces web dando como resultado el lenguaje IFML y fue adoptado como un estándar [Object Management Group, 2015].

WebML es un lenguaje de especificación gráfico que se ocupa de expresar las características básicas de alto nivel de un sitio web sin especificar las características de bajo nivel. Estas características permiten desarrollar una herramienta CASE que soporta estas construcciones. Adicionalmente posee una sintaxis en XML que puede ser usada para producir software de

[9]Tomado de: http://laurel.datsi.fi.upm.es/ ssoo/DAW/Trabajos/2008-2009/024/

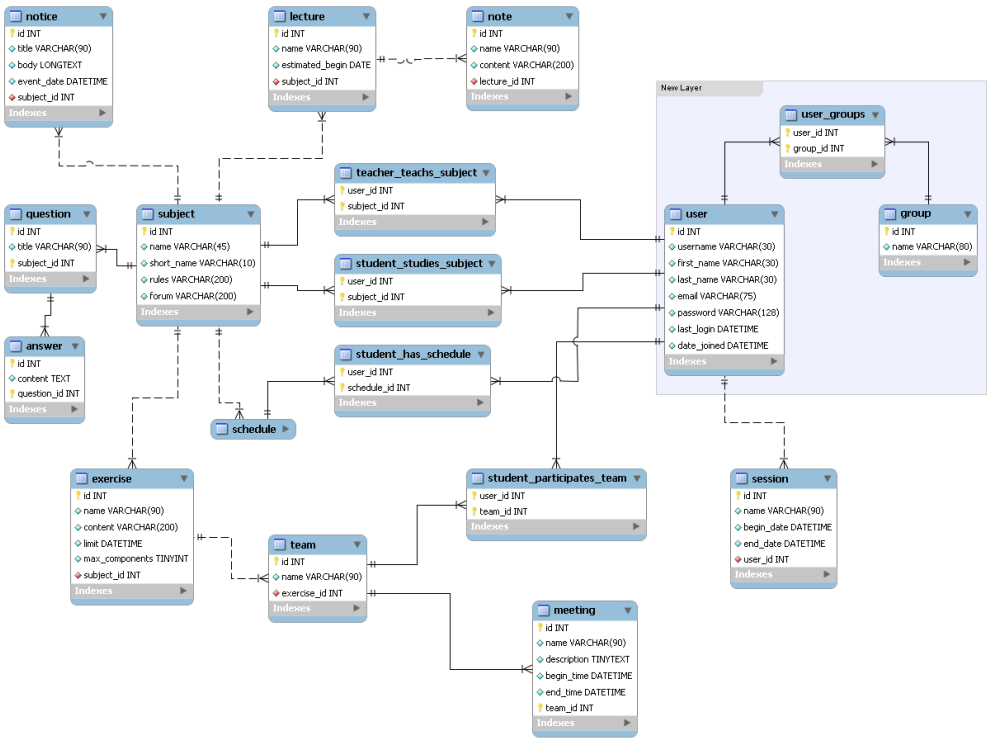


Figura 2-11: Ejemplo de diagrama E-R

manera automática.

Una especificación en WebML consiste en cuatro perspectivas diferentes:

1. Modelo Estructural: modelo de datos del sitio web.
2. Modelo de Hipertexto: describe los hipertextos que se mostraran en el sitio.
3. Modelo de Presentación: representa el *layout* y propiedades gráficas del sitio.
4. Modelo de Personalización: para separar los contenidos por usuarios o grupos de usuarios. Define quien tiene acceso a que contenidos.

Estas cuatro perspectivas permiten definir un sitio web. Además, estas representaciones gráficas pueden ser de utilidad para comunicarse con las demás personas involucradas en el proyecto (por ejemplo con los diseñadores gráficos).

2.5. Compiladores

El procesamiento de las especificaciones que se hará en esta tesis estará basado en técnicas de compiladores que extraerán la información de diseño presente en las especificaciones escritas en lenguaje natural restringido. Posteriormente se usará esta información para generar código fuente. Por esto, en esta sección se hace una breve introducción a los conceptos principales y las fases del proceso de compilación.

Un compilador es un programa de computador que toma como entrada un texto escrito en un lenguaje x y retorna un texto escrito en un lenguaje y [Aho et al., 2007]. Este proceso generalmente cuenta con las fases que se muestran en la figura **2-12**.

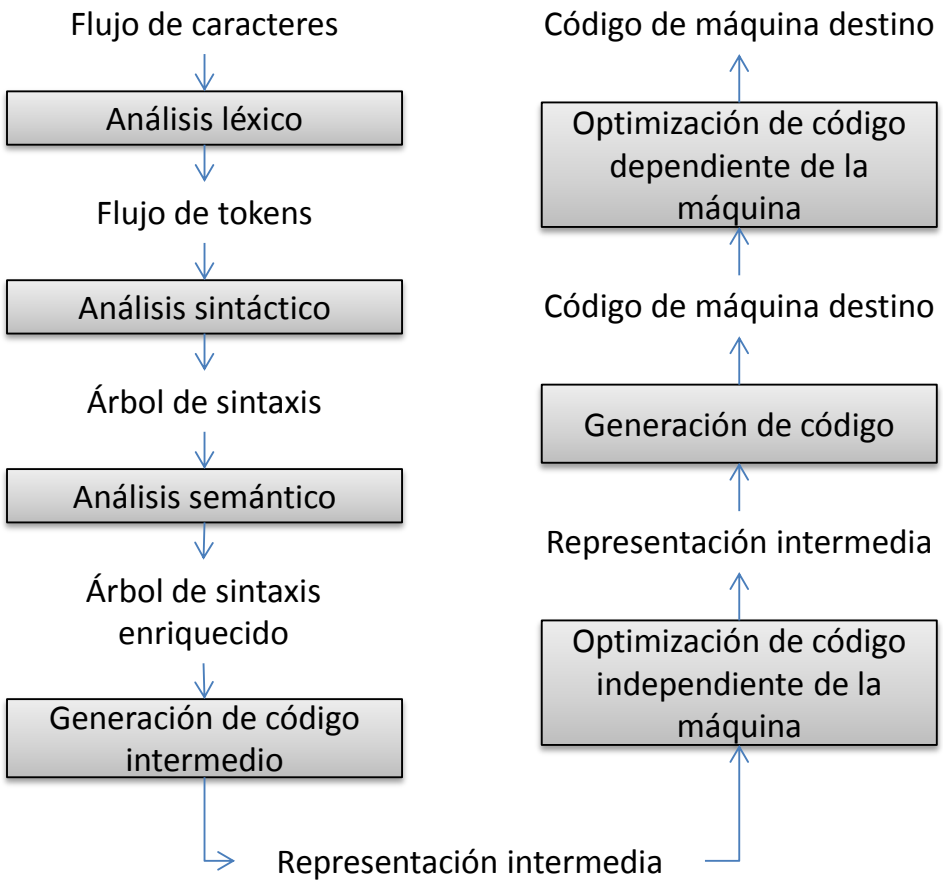


Figura 2-12: Fases de un compilador

2.5.1. Análisis léxico

Como explica [Aho et al., 2007] el punto de partida para este proceso es el análisis léxico el cual generalmente se realiza con ayuda de autómatas determinísticos. Para cada lexema este análisis produce duplas de la forma:

$$< nombre - token, valor - atributo >$$

El nombre del token denota un tipo abstracto (variable, entero, cadena de caracteres, etc.) y el segundo valor representa el valor concreto de ese tipo abstracto (*nombre_de_variable*, *15978*, *"cadena de prueba"*). Un ejemplo de este procesamiento se puede ver en la figura **2-13**.

$$minutos = horas * 60$$
$$\Downarrow$$
$$[(id, minutos), (igual, =), (id, horas), (por, *), (entero, 60)]$$

Figura 2-13: Ejemplo de análisis léxico

Generalmente esta etapa se hace en conjunción con las dos siguientes (análisis sintáctico y semántico). Para realizar esta tarea es necesaria una definición de patrones que identifiquen cada posible secuencia de caracteres en la entrada como un token. Un ejemplo de patrones comunes es mostrado en la tabla **2-6**.

Token	Descripción informal	ejemplos
if	caracteres i, f	if
else	caracteres e, l, s, e	else
id	letra seguida de números o letras	pi, score,i,d2
número	cualquier constante numérica	3 , 3.15, -16.8

Tabla 2-6: Ejemplo de patrón de tokens

Para implementar estos patrones en el analizador sintáctico es necesario implementar un autómata finito que logre tomar el carácter en la posición actual y determine los cambios de estados pertinentes para determinar el principio y fin del lexema. Esta implementación se puede realizar con una estructura *switch-case* en un ciclo hasta el final del texto de entrada.

Esta tarea de construir un analizador léxico se puede también realizar usando un generador de analizadores léxicos, el cual recibe como entrada una serie de expresiones regulares

(REGEX) para determinar el token al cual pertenecen los lexemas. Un ejemplo de herramienta que generan estos analizadores es FLEX[10]. La tabla **2-6** se muestra usando expresiones regulares en la tabla **2-7**.

Token	Expresión regular	ejemplos
if	[iI][fF]	if
else	[eE][lL][sS][eE]	else
id	[a-zA-Z][a-zA-Z0-9]*	pi, score,i,d2
número	(-)?[0-9]+('.'[0-9]+)?	3 , 3.15, -16.8

Tabla 2-7: Ejemplo de patrón de tokens usando REGEX

2.5.2. Análisis sintáctico

Esta fase también conocida como *"parsing"* usa como entrada la salida de la fase de análisis léxico y produce una representación intermedia en forma de árbol que representa la forma gramatical de la secuencia de tokens de entrada. Un ejemplo del árbol de la figura **2-13** puede verse en la figura **2-14**. En este árbol cabe resaltar que el nodo que este mas arriba es el que se resuelve primero.

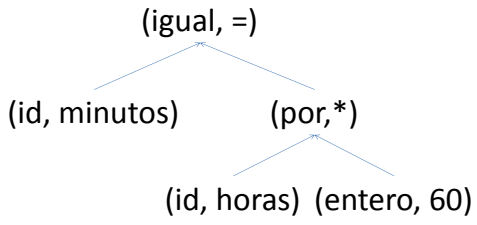


Figura 2-14: Ejemplo de análisis sintáctico

Por diseño cada programa tiene una serie precisa de reglas que describen la sintaxis del programa escrito en este lenguaje. Esta sintaxis se pueden representar en notación BNF (Backus - Naur Form). Un ejemplo de una gramática BNF es presentado en la figura **2-15**.

Las gramáticas que se pueden usar eficientemente para la tarea de compilación son gramáticas libres de contexto. Estas gramáticas tienen 4 elementos: los terminales que son las unidades de las que se construye el programa (elementos léxicos); no-terminales que son variables

[10]https://github.com/westes/flex

$$E \rightarrow E + T|T$$
$$T \rightarrow T * F|F$$
$$F \rightarrow (E)|id$$

Figura 2-15: Ejemplo de gramática BNF

sintácticas que denotan conjuntos ordenados de tokens u otros no-terminales; un no-Terminal distinguido como el punto de partida del procesamiento; y reglas de producción que determinan la manera en que terminales y no terminales pueden ser combinados para formar cadenas.

Estas gramáticas son usadas para definir la estructura de los lenguajes. Por ejemplo el no terminal (cada letra mayúscula que representa una regla de producción) F de la gramática de la figura **2-15** es capaz de aceptar las cadenas "(minutos)" y "(minutos+segundos)" con paréntesis. Usando estas reglas de producción se construye un árbol de sintaxis.

Estos analizadores sintácticos se pueden generar de manera automática únicamente con la definición de esta gramática. Un ejemplo de herramientas que generan estos analizadores son Xtext[11], ANTLR4[12] y BISON[13].

2.5.3. Análisis semántico

Esta fase revisa el árbol de sintaxis para ver si es semánticamente válido: que las variables estén definidas antes de usarse, que los tipos de dato encajen, y demás definiciones propias de cada lenguaje. Adicionalmente modifica el árbol de sintaxis incluyendo información útil para los procesos posteriores. Un ejemplo del resultado de este análisis usando como entrada el ejemplo de la figura **2-14** es mostrado en la figura **2-16**.

2.5.4. Generación de código intermedio

Esta puede ser una de las representaciones intermedias del programa final o el programa final en si mismo. El objetivo es generar un archivo a partir del árbol de sintaxis obtenido del análisis semántico. El código resultante puede ser directamente ensamblador o pude ser un código intermedio como el código de tres direcciones que se muestra en la figura **2-17**.

[11] eclipse.org/Xtext/
[12] www.antlr.org/
[13] https://www.gnu.org/software/bison/

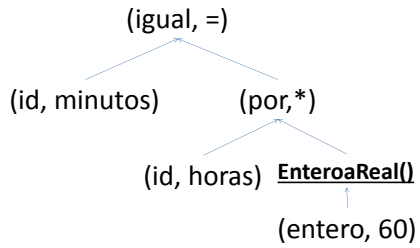


Figura 2-16: Ejemplo de análisis semántico

t1 = EnteroaReal(60)
t2 = t1*horas
minutos = t2

Figura 2-17: Ejemplo de traducción dirigida por la sintaxis

Este código se puede generar usando un esquema de traducción dirigido por la sintaxis. Este tipo de traducciones usan atributos añadidos a los símbolos de la gramática que representan construcciones. Por ejemplo se muestra una regla que traduce de notación prefija a postfija en la figura **2-18**.

Regla de producción: $E \rightarrow E_1 + T$
Regla semántica: $E.code = E_1.code||T.code||'+'$

Figura 2-18: Ejemplo de análisis semántico

Haciendo uso intensivo de esta fórmula es posible generar todo tipo de códigos intermedios y finales. Este tipo de traducciones tiene sus respectivas restricciones y puede requerir más de un recorrido por el árbol de sintaxis. También es posible usar un esquema similar para generar intérpretes.

2.6. Trabajos relacionados

Los trabajos seminales para generar modelos, código fuente o programas ejecutables a partir de un conjunto de especificaciones escritas en lenguaje natural (restringido o sin restringir) fueron propuestos por [Abbott, 1983, Saeki et al., 1989]. Estos trabajos proponían procesos manuales donde se realizaba una clasificación de las palabras y las frases en la especificación basada en las partes del discurso (verbos, sustantivos, adjetivos, etc.), o también de acuerdo al tipo de frase (frase verbal, frase sustantiva, etc.); posteriormente, a partir de estas clasificaciones se realizaba una traducción a modelos formales.

En particular, [Abbott, 1983] proponía partir desde una descripción informal del sistema objetivo escrita en lenguaje natural y se conseguía generar manualmente un programa ejecutable. Esta aproximación se basa en un conjunto de reglas que determinaban las relaciones entre estructuras lingüísticas, tipos de datos, objetos y operadores. Esta información, combinada con módulos previamente escritos y el conocimiento del desarrollador, podía finalmente producir un programa ejecutable escrito en lenguaje ADA.

Posteriormente, [Saeki et al., 1989] propusieron otro proceso manual que involucraba la construcción de tablas clasificando palabras de una especificación informal: tabla de verbos, tabla de acciones y tabla de relación con la acción. Para obtener finalmente un conjunto de diagramas que formalizaban la especificación.

Más tarde estas ideas seminales propuestas por [Abbott, 1983, Saeki et al., 1989] fueron automatizadas por otros autores. Esta automatización ha sido realizada de varias maneras: sin intervención humana posterior a la especificación [Ibrahim and Ahmad, 2010] o con ayuda del usuario para validar ciertos aspectos [Patel, 2014]; usando conocimiento del dominio en forma de ontologías [Geetha and Anandha Mala, 2014] o sin este tipo de conocimiento previo [Krishnan and Samuel, 2010]; produciendo representaciones intermedias y convirtiendo a varios lenguajes de especificación objetivo [Ben Abdessalem et al., 2015].

Por otra parte, otros autores han intentado restringir la manera en que se escriben los requisitos. De esta manera, la información de diseño (la información necesaria para generar un modelo en algún lenguaje de especificación) puede ser fácilmente extraída desde los requisitos. Algunos ejemplos representativos de este tipo de enfoque incluyen los trabajos de: [Nishida et al., 1991, Smith et al., 2003, Konrad and Cheng, 2005, Ilić, 2007, Zeaaraoui et al., 2013, Dahhane et al., 2015].

En el resto de esta sección se presentan los diferentes enfoques utilizados para obtener un modelo o un programa ejecutable a partir de una especificación escrita en alguna forma de

lenguaje natural. Las herramientas y técnicas utilizadas varían en los diferentes enfoques y principalmente combinan aprendizaje de máquina, compiladores, procesamiento de lenguaje natural y reglas de construcción para requisitos. La figura **2-19** ilustra el mapa conceptual que resume los procesos involucrados en cada uno de los enfoques relacionados con este trabajo y la forma en la que están inter-conectados entre sí.

La clasificación de los diferentes enfoques se divide en 3 aproximaciones principales que están basadas en las restricciones que tiene el texto de entrada (especificaciones), así: lenguaje de especificación basado en lenguaje natural, requisitos escritos utilizando plantillas, y finalmente, descripciones en lenguaje natural sin restricción alguna. No obstante, es importante aclarar que algunos trabajos combinan varios de estos enfoques. Por ejemplo, en [Ambriola and Gervasi, 1997] se usan plantillas para procesar texto sin restringir; en [Krishnan and Samuel, 2010] se usan gramáticas sin ambigüedades y compiladores para procesar texto sin restringir; en [Desai et al., 2015] se propone un *framework* para construir programas en lenguajes de dominio específico usando aprendizaje de máquina para determinar un conjunto de programas posibles que permita al usuario seleccionar el que más se acomoda a sus necesidades; y en [Raghothaman et al., 2016] se propone la sugerencia automática de código fuente basándose en consultas hechas en lenguaje natural.

2.6.1. Lenguajes de especificación basados en lenguaje natural

Tratar de coordinar todas las opiniones y deseos de los *stakeholders* del proyecto usando un lenguaje de especificación formal como Z, B o VDM++, puede ser una tarea difícil de alcanzar debido al nivel de formalismo requerido para hacer y entender especificaciones usando estos lenguajes [Mossakowski et al., 2004, ISO and IEC, 2002, Sannella and Wirsing, 1999]. En cambio, para facilitar la construcción de especificaciones, se propone desarrollar otro lenguaje de especificación formal, cuyo nivel de abstracción esté en medio entre el lenguaje natural y un lenguaje de programación. De esta forma, se busca que tanto los participantes técnicos como los no-técnicos puedan entender claramente las especificaciones.

Por lo tanto, estos lenguajes de especificación basados en lenguaje natural deben ser fáciles de entender para una persona sin un profundo conocimiento de las matemáticas y lo suficientemente formal para ser traducido a un lenguaje de programación. En [Bryant and Lee, 2002] se propuso un lenguaje de especificación formal llamado TLG usando una gramática de dos niveles. Posteriormente, los mismos autores desarrollaron una solución que genera código fuente a partir de textos sin restringir usando este lenguaje de especificación como representación intermedia [Lee and Bryant, 2004]. Un ejemplo de una breve especificación escrita en

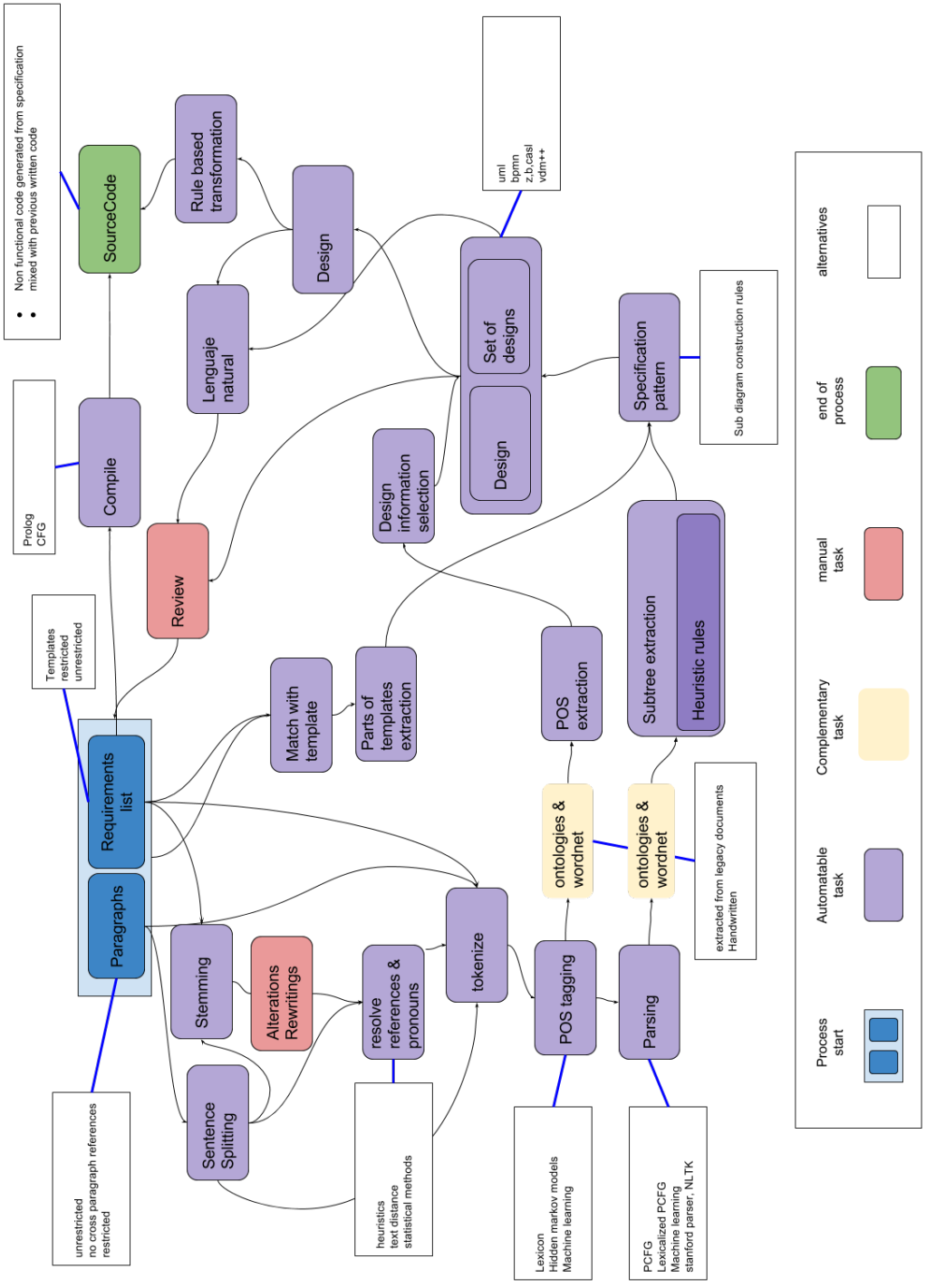


Figura 2-19: Mapa conceptual de trabajos relacionados

lenguaje TLG se puede ver en la figura **2-20**.

People :: { first name String middle initial Character last name String }*.
Symbol Table :: { id Identifier type Type value Integer}+.

Figura 2-20: Ejemplo TLG tomado de [Bryant and Lee, 2002]

Otra propuesta representativa consistía en desarrollar una versión restringida del lenguaje inglés como un lenguaje de especificación [Schwitter, 1996]. Esto significaba restringir la gramática del idioma inglés a unas reglas gramaticales específicas y crear unas reglas de interpretación para eliminar las ambigüedades en los textos. En la figura **2-21** se puede ver un ejemplo de una especificación escrita en este lenguaje llamado *Attempto*. Con estas características se da a un compilador tradicional (o uno implementado en Prolog) la habilidad de extraer un único árbol de sintaxis para luego traducir y/o ejecutar la especificación.

There is a customer.
Every customer has a card.
There are at least 3 apples.

Figura 2-21: Ejemplo de Attempto tomado de [Schwitter, 1996]

Por otra parte, el trabajo denominado *"Semantic of Business Vocabulary and Business Rules"* (SBVR) [OMG, 2008] también puede estar considerado dentro de esta categoría. Se trata de un lenguaje de especificación llamado SBVR, el cual es procesable mediante técnicas de compilación, usa un lenguaje en el dominio del negocio, y a su vez, se aleja de tecnicismos. Un ejemplo de una especificación realizada en SBVR puede ser vista en la figura **2-22**[14].

rental has driver
it is obligatory that rental has driver
it is obligatory that each rental has at least one driver
it is obligatory that each rental has no more than 4 drivers

Figura 2-22: Ejemplo de SBVR

[14]http://www.kdmanalytics.com/sbvr/sbvr_intro_1.html

Estas aproximaciones tienen como desventaja principal de que al diseñador le corresponde aprender un lenguaje casi artificial. En contraste, su principal ventaja consiste en que el procesamiento del texto es más sencillo y toda la información contenida en el texto puede ser usada en el proceso sin dificultades ni ambigüedades [Aho et al., 2007, Schwitter, 1996, Bryant and Lee, 2002].

Trabajos clasificados en este enfoque

Estos trabajos no dependen del uso de ontologías puesto que todos los elementos se definen en términos de la especificación misma y con sus tipos predefinidos. De esta forma, no necesitan de conocimiento previamente definido para realizar la nueva especificación. Además, no es necesario realizar ningún pre-procesamiento a estas especificaciones ya que su forma favorece el procesamiento automático. Para estos trabajos es necesario contar con un código previamente escrito para poder ejecutar la especificación al igual que como, por ejemplo, los *"header (.h)"* de lenguaje C o las librerías de Python. Los trabajos revisados que encajan en esta categoría se muestran en la tabla **2-8**, donde se presentan sus características principales.

Referencia	Ayuda humana	Tipo de procesado	Origen de información	Usa ontologías	Código predefinido	Modelo intermedio	Modelo objetivo	Pre-proceso
[Bryant, 2002]	No	Compilador	AST + reglas	No	Si	C	Ejecutable	No
[Schwitter, 1996]	No	Compilador Prolog	AST	No	Si	-	Ejecutable	No
[OMG, 2008]	No	Compilador	AST+ reglas	No	Si	-	-	No

Tabla 2-8: Lenguajes de especificación basados en lenguaje natural

2.6.2. Requisitos utilizando plantillas

Otro tipo de aproximación para resolver ambigüedades en textos escritos en lenguaje natural y generar modelos automáticamente a partir de éstos, consiste en restringir las posibles entradas a un conjunto de plantillas. Estas plantillas representan el modelo final en una forma más amigable para el usuario no técnico, y al mismo tiempo, contienen la información de diseño en posiciones fijas en el texto, lo cual simplifica su análisis. Algunos trabajos representativos de este enfoque son: [Nishida et al., 1991, Smith et al., 2003, Konrad and Cheng, 2005, Ilić, 2007, Zeaaraoui et al., 2013, Dahhane et al., 2015].

Este tipo de soluciones funcionan con listas de requisitos escritas usando plantillas predefinidas. A continuación se presentan algunos ejemplos:

- [Nishida et al., 1991]

```
<procedure sentence>::=
<procedure predicate verb><term>
[(to|from|with)<term>]* [in<format>]
[by<procedure name>]
<procedure sentence>, and store
the result in <term>
```

- [Zapata, 2006]

```
A <ESTA CONFORMADO POR> B
```

- [Zeaaraoui et al., 2013]

```
As a <role>,
I want to <action-1> and <action- 2>...
<action-n> <object>,
so that <business value>
```

- [Videira et al., 2006]

```
<Entity Inheritance Definition> :
<Entity> is a <Entity>
```

- [Konrad and Cheng, 2005]

```
\it is always the case that "
(durationCategory | periodicCategory
| re- altimeOrderCategory)
```

Estos patrones tienen que ser escritos manualmente y extender el conjunto de requisitos procesables requiere escribir manualmente las nuevas reglas. También, es importante aclarar que, en la mayoría de los casos, cuando los requisitos no encajan en ninguna plantilla son descartados, y por lo tanto, su información no se ve reflejada en el modelo final [Granacki and Parker, 1987].

Después de la extracción de la información de diseño, cada requisito (como el que se muestra en la figura **2-23**) es relacionado a un subdiagrama (como el que se muestra en la figura **2-24**), y cuando todos los subdiagramas están completos se unen en uno solo.

"Whenever Class1 is in state Wait, it will enter the state Process within 5 time units."

Figura 2-23: Requisito de ejemplo tomado de [Konrad and Cheng, 2005]

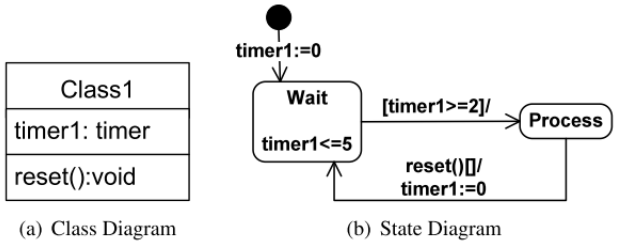


Figura 2-24: Ejemplo de subdiagramas tomado de [Konrad and Cheng, 2005]

Trabajos clasificados en este enfoque

En estos trabajos, el conjunto de expresiones realizables puede requerir predefinir conceptos, es decir, de algún tipo de ontología. La mayoría de los trabajos encontrados se encargan de obtener un modelo, o un conjunto de estos, pero no un programa ejecutable. Estos trabajos hacen uso de código previamente escrito si el resultado final del procesamiento es código fuente. También, es importante considerar que hay que ser especialmente cuidadoso en el texto a escribir puesto que el procesamiento es preciso, como en un lenguaje de programación. Por esta razón, algunas de las aproximaciones usan software especializado para escribir los requisitos en las plantillas correspondientes. El procesamiento de estos requisitos ser puede realizar fácilmente mediante técnicas de compiladores. Los trabajos que se clasifican dentro de esta categoría se muestran en la tabla **2-9**.

Referencia	Ayuda humana	Tipo de procesado	Origen de información	Usa ontologías	Código predefinido	Modelo intermedio	Modelo objetivo	Pre-proceso
[Nishida et al., 1991]	Ayuda con la resolución de módulos	Encajar en plantilla	Parte de plantilla	No	Si	Expresiones Case-Like	C - Lisp	No

[Smith et al., 2003]	Escoger elementos	Encajar en plantilla	Parte de plantilla	No	No	-	Diagrama de estados y árbol de decisiones	No
[Liu et al., 2004]	Completar información faltante	Encajar en plantilla	Parte de plantilla	Si	No	Tabla de casos de uso	UML (Caso de uso y clase)	Reescribir
[Konrad and Cheng, 2005]	Escoger elementos	Encajar en plantilla	Parte de plantilla	No	No	-	UML (clases y secuencias)	No
[Videira et al., 2006]	No	Encajar en plantilla + Fuzzy matching	Parte de plantilla	No	No	-	UML (varios)	Alteración por reglas
[Zapata, 2006]	No	Encajar en plantilla	Parte de plantilla	No	No	Esquema pre-conceptual	UML(3 diagramas)	No
[Ilić, 2007]	método manual	Encajar en plantilla	Parte de plantilla	No	No	Tabla de requisitos	B	Reescribir requisitos
[Popescu et al., 2008]	Calificar el modelo final	Encajar en plantilla	Parte de plantilla	Yes	No	Textual OAAM	UML (clases)	No
[Gordon and Harel, 2009]	Resolver ambigüedades	Encajar en plantilla + POS tagging	Parte de plantilla + POS	No	Si	LSC	LSC o UML(secuencias)	No
[Fatwanto, 2012a] [Fatwanto, 2012c]	Escribir en notación tabular y encerrar palabras	Encajar en plantilla	Parte de plantilla	No	No	-	UML (estados, clases)	Representación tabular
[Zeaaraoui et al., 2013] [Dahhane et al., 2015]	No	Encajar en plantilla	Parte de plantilla	No	No	Historia de usuario	UML (clases)	No

Tabla 2-9: Trabajos que hacen uso de plantillas para requisitos

2.6.3. Especificaciones en texto en lenguaje natural sin restringir

¿Qué tal si el diseñador quiere extraer una especificación desde un texto escrito en formato libre?, ¿o de un texto largo escrito por el cliente en sus propias palabras?, ¿o de una una lista de requisitos escritos sin restricción alguna? Esta aproximación parcialmente soluciona este problema implementando y extendiendo las ideas originales de [Abbott, 1983, Saeki et al., 1989] usando múltiples técnicas y herramientas como lexicones, wordnet, ontologías, aprendizaje de maquina, procesamiento de lenguaje natural y reglas heurísticas.

Cada uno de los trabajos implementa diferentes subconjuntos de los procesos descritos en esta sección [Li et al., 2005b], o pasan por varias representaciones intermedias para obtener el modelo final del sistema deseado [Bajwa and Choudhary, 2012]. A continuación se describen las tareas más comunes para llevar a cabo este tipo de procesamiento de lenguaje natural sin restringir. En la mayoría de los casos se asume la correctitud gramatical de la especificación.

Sentence splitting

En la mayoría de los trabajos en los que se trata con texto escrito en párrafos sin restringir, el primer procesamiento realizado consiste en partir este párrafo en varias frases (muchas veces buscando estructuras del tipo *"sujeto+objeto+verbo"*, dependiendo del lenguaje a procesar). Esto es necesario para reducir la capacidad computacional necesaria para realizar el procesamiento del texto en lenguaje natural [Chioaşcă, 2012]

Esta tarea no es tan sencilla de automatizar como podría parecer. La primera aproximación que uno puede pensar fácilmente es dividir la cadena de caracteres usando como delimitadores las comas y puntos. Sin embargo, esto deja abierto un problema cuando las construcciones gramaticales utilizan anáforas, pronombres y puntos de abreviaciones [Denis and Baldridge, 2007, Tetreault, 2001, Hobbs, 1977]. A menudo se usan enfoques más complejos para lidiar con este tipo de problemas. Algunos de estos se basan en heurísticas o en aprendizaje de máquina [Bellegarda and Monz, 2015]. La figura **2-25** ilustra un ejemplo de *"sentence splitting"*.

Stemming

Para llevar a cabo los procesos subsecuentes mas rápidamente, la raíz de cada palabra es extraída (como podemos ver en la figura **2-26**). Este proceso es conocido como *stemming* y puede ser realizado de varias maneras: *rapid stemming algorithm* [More, 2012], *Porter*

*"**The CEO of the company** checks the performance metrics of the sales program**, then he** sends reviews to the sales program people"*

⇓

*"**The CEO of the company** checks the performance metrics of the sales program"*

*"**The CEO of the company** sends reviews to the sales program people"*

Figura 2-25: Ejemplo de *sentence splitting*

algorithm [Willett, 2006], o con procesos basados en aprendizaje de máquina [Smirnov, 2008].

argue, argued, argues, arguing, argus

⇓

argu (que no es una palabra)

Figura 2-26: Ejemplo de *stemming*

Etiquetado POS (Part-of-speech tagging)

Este proceso era el primero en las aproximaciones de [Abbott, 1983, Saeki et al., 1989]: clasificación de las palabras en: verbos, adjetivos, sustantivos, determinantes, adverbios, pronombres, preposiciones, conjunciones o intersecciones. El numero de partes del discurso varía entre 8 [Bongers, 1946], 17 [Nikora, 2005], y 48 [Harmain and Gaizauskas, 2000]. Estas son partes del discurso (POS, por sus siglas en inglés), y constituyen la primera fuente de información para el diseño. Un ejemplo se puede ver en la figura **2-27**.

Para realizar automáticamente esta tarea, varias técnicas han sido desarrolladas con el tiempo [Sampson, 2003]. Por ejemplo, usando un lexicon en el que cada palabra es colocada en una lista con su respectiva parte del discurso [Bryant and Lee, 2002]; usar modelos ocultos de Markov donde cada palabra es etiquetada con respecto a las etiquetas y características de las palabras previas [Yuan, 2010]; técnicas de aprendizaje de máquina que usan información (características o *features*) de palabras cerca a la que se quiere etiquetar [Ma, 2002, Bellegarda and Monz, 2015], este proceso puede alcanzar precisiones del 94 % [Yuan, 2010].

And/CC now/RB for/IN something/NN completely/RB different/JJ

Figura 2-27: Ejemplo de etiquetado POS

Parsing

Únicamente con la información de las partes del discurso no es suficiente para encontrar relaciones precisas entre términos. Por ejemplo en la frase "el cliente es una persona", sabiendo que "cliente" y "persona" son sustantivos, y que "es" es un verbo, no es suficiente para determinar que hay una relación entre esos dos sustantivos.

Para obtener esas relaciones, la aproximación consiste en *parsear* la frase, es decir, analizarla sintácticamente para generar un árbol sintáctico o *parse tree* (como se muestra en la figura **2-28**). En este árbol se buscan sub-árboles que dependen de lo que se busque en cada trabajo. Por ejemplo, se pueden buscar las siguientes construcciones: frases sustantivas como "the blue system" o frases verbales como "each branch is included in exactly one local area" que son una fuente importante de información de diseño [Selway et al., 2013].

Esta tarea es realizada automáticamente mediante gramáticas probabilísticas libres de contexto [Johnson, 1998]. Las gramáticas probabilísticas son gramáticas libres de contexto tradicionales con un componente estadístico adicional que permite usar información extra acerca de cómo los humanos entendemos los textos (basándose en grandes bancos de información de árboles sintácticos). Este proceso puede alcanzar precisiones del 92 % [Collins, 2013].

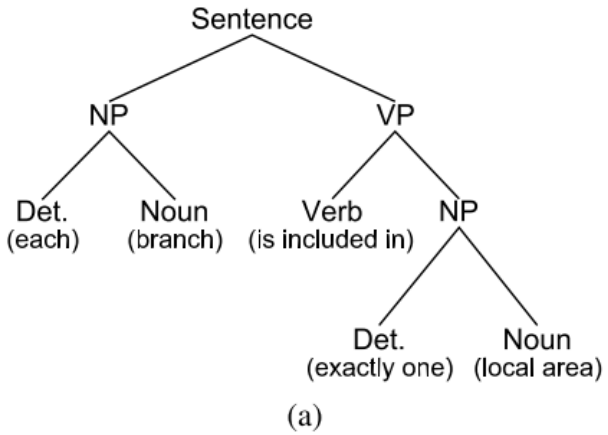


Figura 2-28: Ejemplo de árbol sintáctico tomado de [Selway et al., 2013]

Reglas y heurísticas para seleccionar información de diseño

¿Qué hacer con la información del etiquetado POS y el árbol de sintaxis?, estas tareas se llevan a cabo para obtener información del texto, ahora en este proceso se soluciona el problema de "¿qué información es útil para solucionar este problema?". Estas propuestas suelen apoyarse en el uso de ontologías y herramientas como Wordnet [Patel, 2014, Zhou, X; Zhou, 2008, Deeptimahanti and Sanyal, 2011].

Las ontologías son definiciones de tipos, entidades, relaciones en el contexto de un dominio de conocimiento [Bouquet et al., 2004]. En el contexto de esta tesis, las ontologías se usan para encontrar información y relaciones que podrían no estar de forma explícita en el texto de entrada [Zhou, X; Zhou, 2008, Herchi and Abdessalem, 2012, Granacki and Parker, 1987, Harmain and Gaizauskas, 2000].

Por otra parte, Wordnet es *"una base léxica electrónica, que es considerada como el recurso disponible más importante para investigadores en lingüística computacional, análisis de textos, y varias áreas relacionadas"* [Fellbaum, 1998]. En el contexto de este trabajo es usado para encontrar sinónimos entre palabras y relaciones entre términos que no están explicitas en un texto libre [Gordon and Harel, 2009, Ibrahim and Ahmad, 2010, Harmain and Gaizauskas, 2000, Overmyer et al., 2001, Popescu et al., 2008].

Usando estos recursos en conjunto con algunas reglas heurísticas es posible extraer información de diseño como entidades y relaciones o sub-diagramas. Estas reglas toman la información y la convierten en información útil para el diseño. Por ejemplo:

- Todos los sustantivos serán tomados como clases [Omar and Razik, 2008].
- Cuando dos sustantivos aparecen seguidos en el texto, entonces el primer sustantivo será traducido en una clase [Omar and Razik, 2008].
- El número cardinal que exista antes de un alfabético o sustantivo será tomado como variable [Omar and Razik, 2008].
- Los adjetivos serán usados para determinar una función [Omar and Razik, 2008].
- Si hay un sustantivo precedido de un verbo como leer, lee, imprimir, imprime y muestra; o de un sustantivo como entrada, entradas, entonces éste será tomado como variable [Omar and Razik, 2008].
- Si un concepto ocurre sólo una vez en el documento y su frecuencia es menos del 2 %, entonces ignorarla como clase [Ibrahim and Ahmad, 2010].

- Si un concepto es una frase sustantiva (sustantivo+sustantivo), y si el segundo sustantivo es un atributo, entonces el primero es una clase y el segundo es un atributo de esa clase. Ejemplos: “Customer Name” o “Book ISBN” [Ibrahim and Ahmad, 2010].
- Si un concepto es una frase sustantiva (sustantivo+sustantivo) incluyendo el guión bajo “_” en medio de los dos sustantivos, entonces el primer sustantivo es una clase y el segundo es un atributo de esa clase. Ejemplos: “customer_name”, “departure_date” [Ibrahim and Ahmad, 2010].
- Si el concepto es verbo (VB) y satisface la regla R-Rule2, y el concepto es igual a uno de los siguientes: {“consiste de”, “contiene”, “mantiene”, “incluye”, “dividido en”, “tiene parte”, “consiste de”, “carga”, “involucra”, “implica”, “adopta”}, entonces la relación descubierta es la de composición o agregación. Ejemplo: “Library Contains Books” entonces la relación entre “Library” y “Book” es composición [Ibrahim and Ahmad, 2010].

Esta información de diseño puede ser guardada para cada frase o para toda la especificación de requisitos dependiendo del trabajo.

Generación de sub-modelos

Si la especificación inicial fue dividida en varias frases, para cada frase con información de diseño se crea un sub-diagrama como el que se muestra en la figura **2-24**, o la información de diseño identificada se fusiona en una única representación. Cuando todos los sub-diagramas están listos se unen obteniendo un solo modelo (o un conjunto de ellos) [Smith et al., 2003, Konrad and Cheng, 2005, Song et al., 2004].

Modelo Final

El modelo final puede ser revisado de forma manual [Li et al., 2005b, Patel, 2014] o traducido a código fuente en un lenguaje especifico usando otro tipo de herramientas complementarias externas [Beum-Seuk Lee, 2001, Liu et al., 2004].

Lenguajes de especificación objetivo

Como salida de este proceso se obtiene una especificación sin ambigüedades (con un único significado) que puede ser implementada. Esta especificación generalmente no esta muy alejada de lo que modelaría un humano [Ben Abdessalem et al., 2015]. El lenguaje objetivo o

la representación de salida puede variar ampliamente. A continuación se presentan los más utilizados:

- Programa ejecutable [Bryant, 2002, Schwitter, 1996, Njonko and El Abed, 2012].
- UML (Unified Modeling Language) [Bryant and Lee, 2002, Bryant, 2000, Song et al., 2004, Ilieva and Ormandjieva, 2005, Harmain and Gaizauskas, 2000, Overmyer et al., 2001].
- Lenguaje Z [Shukur et al., 2002].
- E-R [Meziane and Vadera, 2004, Geetha and Mala, 2013, Geetha and Anandha Mala, 2014].
- Verilog [Harris, 2012].
- ADA [Abbott, 1983].
- BPMN [Friedrich et al., 2011].
- SBVR [Selway et al., 2015].
- C - Lisp [Nishida et al., 1991].
- B [Ilić, 2007].

La mayoría de los trabajos revisados generan un diagrama de clases de UML o varios de los diagramas de UML (secuencia, casos de uso, estados).

Trabajos clasificados en este enfoque

Estos trabajos constituyen el mayor esfuerzo que se ha realizado en esta área. Muchas veces es necesaria la intervención posterior de una persona especializada para lograr un buen resultado. El uso extensivo de herramientas de procesamiento de lenguaje natural (NLP) puede inducir errores extra. Por otra parte, el uso de ontologías es mucho más generalizado que en los dos enfoques explicados anteriormente dada la libertad que permite el lenguaje natural. Rara vez los trabajos clasificados en este enfoque obtienen un programa ejecutable y se limitan a llegar a algún tipo de modelo, siendo los diagramas de UML los más populares. En los casos que estos trabajos obtienen un programa ejecutable, no se trata de procesos completamente automáticos.

Una clasificación extra es hecha para estos trabajos: primero, los que se muestra en la tabla **2-10**, que usan una lista de frases sin restringir para generar los modelos; segundo los que usan una lista de párrafos completos como entrada para el proceso, estos se muestran en la

tabla **2-11**.

Referencia	Ayuda humana	Tipo de procesado	Origen de información	Usa ontologías	Código predefinido	Modelo intermedio	Modelo objetivo	Pre-proceso
[Granacki and Parker, 1987]	Revisar interpretación	Encajar en plantilla	Parte de plantilla	Si	No	-	Diagramas de estado	No
[Ambriola and Gervasi, 1997]	Escribir reglas y términos que buscar	Encajar en plantilla + Fuzzy Matching	Parte de plantilla	Si	No	-	Diagrama de flujo de datos, de entidad y otros	reemplazar palabras y stemming
[Bryant, 2000]	Resolver referencias	Parsing	POS y subárbol	No	No	TLG	UML(clases)	No
[Harmain and Gaizauskas, 2000]	No	Bottom-up chart parser	Frases	Si	No	-	UML(clases)	Stemming
[Shukur et al., 2002]	No	Grammar parser	POS + Frases	No	No	Modelo conceptual	Z	No
[Meziane and Vadera, 2004]	No	Grammar parser	POS + Frases	No	No	LFL	E-R	No
[Song et al., 2004]	Proceso manual	Manual	POS + frases + reglas	No	No	-	UML(clases)	No
[Ilieva and Ormandjieva, 2005]	No	POS tagging + partir	Partes de frases	Yes	No	red semántica	UML (clases)	No
[Li et al., 2005b]	Varias confirmaciones	POS tagging	POS	Si	No	Red semántica	UML (varios)	No
[Chioaşcă, 2012]	No	shallow parser	POS + rules + Match con modelo(ML)	No	No	-	"Model"	No
[Landhäußer et al., 2012]	Revisar y corregir	Parsing	POS + rules	No	No	Nodos textuales	UML(clases)	No
[Harris, 2012]	No	recursive descent syntactic parser	POS + frases	No	Yes	-	Verilog	No

[Rui, 2013]	Proceso manual	Manual	Representación tabular	No	No	Representación tabular	Diagrama de estados	No

Tabla 2-10: Trabajos que usan frases de forma libre

Referencia	**Ayuda humana**	**Tipo de procesado**	**Origen de información**	**Usa ontologías**	**Código predefinido**	**Modelo intermedio**	**Modelo objetivo**	**Pre-proceso**
[Abbott, 1983]	Proceso manual	Manual	POS + phrases	No	Yes	-	Ada	No
[Saeki et al., 1989]	Proceso manual	Manual	POS	No	No	Tablas de Acción, verbo y sustantivo,	Diagrama de Clases	No
[Overmyer et al., 2001]	Escoger clases y atributos	POS tagging	POS	No	No	-	UML (clases)	No
[Bryant and Lee, 2002] [Lee and Bryant, 2004]	corregir interpretación	Parsing	POS + parse tree	Si	No	TLG - VDM++	UML (clases) , código Java	No
[Ishihara et al., 1993]	Proceso manual	Manual	Árbol de dependencia	No	No	"estructura C"	Especificación algebraica	No
[Anandha and Uma, 2006]	No	Encajar en plantilla	Parte de plantilla	No	No	texto normalizado	UML (clases)	Normalizar
[Omar and Razik, 2008]	No	Memory based shallow parser	POS + Frases+ heuristicas	No	No	-	UML (clases)	No
[Zhou, X; Zhou, 2008]	Escribir ontologias	Modelo de araña	ontologia, grafo de relaciones	Si	No	Modelo de araña	UML (clases)	No
[Ibrahim and Ahmad, 2010]	No	OpenNLP	POS + Frases + heuristicas	Si	No	-	UML (clases)	Stemming

[Krishnan and Samuel, 2010]	No	Grammar Parsing	frases + árbol de dependencias	No	No	-	UML (clases)	No
[Deeptimahanti and Sanyal, 2011]	Reescribir & Revisar	Stanford Parser + Match con regla	frases + árbol de dependencia	Si	No	-	UML (varios), Java	reescribir basándose en reglas
[Friedrich et al., 2011]	No	Stanford Parser (factored model)	frases + heuristicas	Si	No	-	BPMN	No
[Njonko and El Abed, 2012]	No	MSLA, parsing	Frases + Etiquetas de rol	Si	Si	SBVR	Ejecutable	No
[More, 2012]	corregir sentencias	OpenNLP + Encajar en plantilla	partes de frases + heuristicas	Si	No	-	UML (clases)	No
[Bajwa and Choudhary, 2012]	No	Parsing	partes de SBVR	No	No	SBVR	UML (clases)	Stemming
[Geetha and Mala, 2013] [Geetha and Anandha Mala, 2014]	No	Grammar parsing	POS + frases	Si	No	XML	E-R	Sentence splitting
[Patel, 2014]	Revisar	Stanford Parser(PCFG)	POS + frases + reglas	Si	No	-	UML (clases, casos de uso)	Sentence splitting
[Vidya Sagar and Abirami, 2014]	No	Stanford Parser(Deep parse)	árbol dependencias + reglas	Si	No	-	UML (clases)	Sentence splitting
[Selway et al., 2015]	No	GATE	parsing + pos	Si	No	Multiples SBVR	SBVR	No
[Ben Abdessalem et al., 2015]	No	Encajar en patrón	parsing + heuristicas	No	No	XML	UML (clases)	Sentence splitting
[Oliveira et al., 2006]	Escribir cosas diferentes dependiendo del rol	Encajar en patrón + selección basadas en historial	parsing + heuristicas	Yes	No	-	UML (clases)	Sentence splitting

Tabla 2-11: Trabajos que usan párrafos de forma libre

3 Prototipado automático de sistemas de información transaccionales

En este capítulo de describe el enfoque propuesto para cumplir el objetivo de esta tesis: diseñar, desarrollar y evaluar una metodología de prototipado rápido de sistemas de información transaccionales. Como punto de partida para del diseño de esta metodología se usa el ciclo de vida del desarrollo de software (SDLC) presentado en la sección 2.2. La metodología que se propone en esta tesis tiene una serie de fases que se basan en las fases del SDLC. Al igual que en el SDLC, se pretende que en la metodología propuesta el prototipado se haga de manera cíclica. Se busca además que esta metodología sea aplicable de manera rápida y dinámica para obtener retroalimentación de los *stakeholders* rápidamente y conseguir corregir errores fácilmente en etapas tempranas del desarrollo de software.

El tipo de prototipos que se pretende desarrollar en esta propuesta se encuentra categorizado en el grupo que pretende ser parte del producto final como se muestra en la sección 2.2.3. Estos prototipos deben cumplir con la característica de ser legibles por desarrolladores humanos, quienes se encargarán de refinar los aspectos que queden por fuera del alcance de los prototipos. En otras palabras, el prototipo debe ser código fuente que se pueda ejecutar, y no un ejecutable ya compilado.

Para generar un prototipo que haga parte del sistema final, es necesario construir una herramienta de prototipado especial que permita la generación de estos sistemas a partir de una especificación dada. En esta tesis, esta herramienta se basa en el procesamiento de una especificación en lenguaje natural restringido para generar los prototipos automáticamente.

La elección del lenguaje natural restringido se hace tomando en cuenta la revisión del estado del arte mostrada en la sección 2.6. El procesamiento de este tipo de lenguajes restringidos se realiza usando técnicas tradicionales de compilación, las cuales permiten realizar esta generación de código de una manera rápida y precisa lo cual es indispensable para el dinamismo y velocidad buscados para la metodología propuesta. No se seleccionó un enfoque basado en el procesamiento de textos sin restringir porque, como se anotó en la sección 2.6,

este tipo de procesamiento podría ser impreciso y lento debido al uso de técnicas de procesamiento de lenguaje natural. Además, estas aproximaciones generalmente buscan generar modelos que pueden ser usados para generar código esquemático pero no funcional. En esta tesis se busca llegar a la generación de código fuente que sea funcional.

Las características de velocidad y dinamismo en el proceso de prototipado son necesarias para mitigar los errores asociados a la fase de requisitos de software, presentados en la sección 2.3.2. La aproximación para aminorar estos problemas consiste en lidiar rápidamente con los errores a un costo reducido durante una etapa temprana en el desarrollo. Un error en un prototipo podría ser fácilmente corregido durante una reunión donde participen clientes, diseñadores de software, en general, *stakeholders* técnicos y no-técnicos. De esta forma, se puede continuar con la especificación del resto del sistema de una manera rápida. Al mismo tiempo, se garantiza que los requisitos poseen las características mostradas en la sección 2.3.

Para estos fines, se seleccionaron dos lenguajes de especificación a partir de la lista que se presentó en el marco teórico de esta tesis: BPMN y E-R. Estos dos lenguajes de especificación se usaron como un recurso para construir el lenguaje natural restringido propuesto en esta tesis. Este lenguaje será utilizado para escribir una especificación de software, la cual será el insumo para generar el código fuente que corresponde a esta nueva especificación.

Los lenguajes BPMN y E-R fueron seleccionados con el fin de poder expresar una serie de actividades (BPMN) que operen sobre un conjunto de clases de dominio (E-R). Se usa en paralelo a esta idea, el patrón de diseño conocido como modelo-vista-controlador para generar los diferentes archivos que son necesarios para la ejecución del sistema resultante.

En la figura **3-1** se muestra un esquema general BPMN del flujo de trabajo propuesto para el prototipado automático de sistemas de información transaccionales. Este flujo de trabajo será desarrollado en una reunión o serie de estas dependiendo del tamaño del sistema a prototipar.

En primera instancia, todo el proceso se realiza en el contexto de una reunión con los *stakeholders* del proyecto u organización. Es posible que previo a la reunión, por petición de los *stakeholders* o acuerdo mutuo, se realice un primer prototipo con el fin de desarrollar una discusión en torno a este prototipo y cómo mejorarlo.

Como primer paso de la metodología se comunican al diseñador los requisitos que el sistema debe satisfacer. Para esto es importante que se tenga claridad sobre los procesos que se llevan a cabo en la organización y sobre los roles que responsables cada tarea al interior de la organización. También podría ser útil que se lleven documentos y formularios (físicos o digitales) que se usen a diario en la organización, esto con el fin de extraer requisitos de estos.

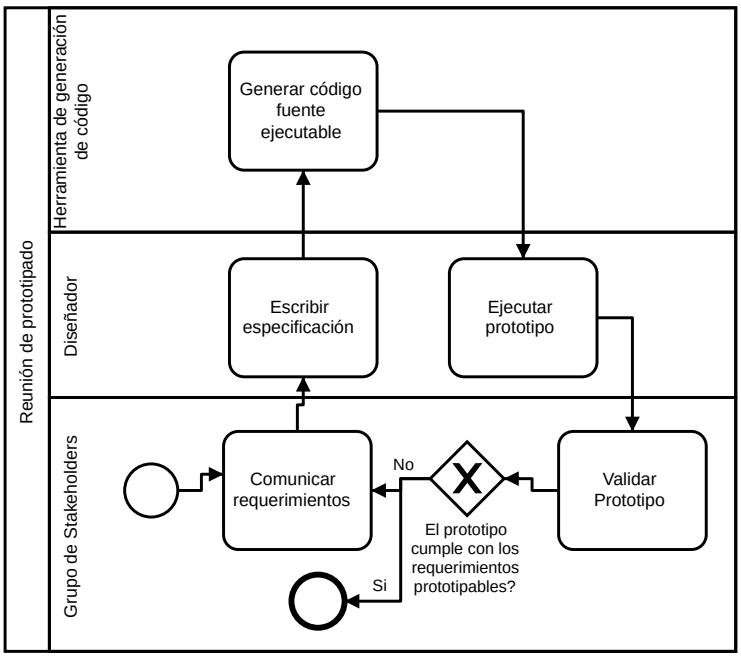


Figura 3-1: Flujo de trabajo propuesto para el SDLC propuesto: prototipado automático de sistemas de información transaccionales

Lo que se pretende entonces es reemplazar la mayor cantidad posible de tareas rutinarias y/o manuales por tareas en el sistema de información a prototipar.

A continuación un diseñador procederá a escribir la especificación en lenguaje natural restringido que se presenta en la sección 3.1. Aunque el lenguaje natural restringido podría ser entendido por un *stakeholder* sin conocimientos técnicos, el usuario objetivo de esta metodología es un diseñador con conocimientos técnicos. Esto es necesario debido a que las relaciones y conceptos expresados con el lenguaje restringido no son fáciles de entender para alguien que no tenga conocimientos en esta área. Este proceso podría ser asistido por herramientas de auto-completado dependiendo de la implementación que se realice de la herramienta propuesta en esta tesis.

Como paso siguiente, la herramienta de generación de código, usando el esquema mostrado en la sección 3.2, generará el prototipo del sistema de información. Después, el diseñador se encargará de colocar en funcionamiento el prototipo y ubicar la parte de ese en la que se estén realizando los ajustes del ciclo actual. Cabe anotar que el prototipo, idealmente, seguirá ejecutándose y detectará los cambios realizados al código fuente. Es decir los cambios en la especificación realizada se actualizarán automáticamente en el prototipo y se podrán visualizar en segundos en el navegador.

Después, los *stakeholders* revisan y validan el prototipo del sistema de información. Si el sistema aún no cumple con los requisitos que pueden ser obtenidos con el sistema de prototipado, la especificación es alterada y el código es generado nuevamente a partir de la nueva especificación. Así, es posible afinar el prototipo de forma cíclica. De esta manera, es posible obtener una retroalimentación directa de los *stakeholders* y conseguir mitigar los posibles errores del SDLC y los requisitos, obteniendo la validación inmediata por parte de todos los participantes en el proyecto.

Por último, se pretende que se tome una decisión con ayuda del diseñador para determinar cuando todos los requisitos prototipables están completos y se deberá completar el resto de requisitos usando el resultado de la metodología como insumo para continuar con el desarrollo de otra manera.

3.1. Lenguaje natural restringido

Este lenguaje restringido pretende definir los procesos y clases de dominio presentes en el sistema de información. Para esto se realiza una transformación desde dos lenguajes de especificación populares como lo son BPMN y E-R. Estas transformaciones tendrán como resultado una serie de reglas gramaticales que se deberán seguir a la hora de escribir estos requisitos para poder realizar el procesamiento automático de estos. En la sección 3.1.1 se describe el razonamiento para llegar a estas transformaciones y las reglas gramaticales que se deberán emplear durante esta fase

3.1.1. Supuestos

Dados dos modelos construidos en diferentes lenguajes de especificación: e construido usando el lenguaje E-R y b usando BPMN, existen dos funciones f y g tal que, esas funciones generan código fuente (S) para el mismo lenguaje objetivo: S_e y S_b (ecuaciones 3-1 y 3-2).

$$f(e) = S_e \tag{3-1}$$

$$g(b) = S_b \tag{3-2}$$

La unión de S_e y S_b puede verse como el código fuente total generado desde los dos modelos originales (ecuación 3-3).

$$f(e) + g(b) = S_b + S_b \tag{3-3}$$

Considere también dos funciones h y j que toman estos modelos y lo convierten en una especificación en lenguaje natural restringido (T), y en sus respectivas funciones inversas h^{-1} y j^{-1} que convierten el lenguaje natural en modelos, así:

$$h(e) = T_e \tag{3-4}$$

$$j(b) = T_b \tag{3-5}$$

$$h^{-1}(T_e) = e \tag{3-6}$$

Como ejemplo de posibles implementaciones de esta función (ecuación 3-6) podemos encontrar diferentes trabajos propuestos por [Geetha and Mala, 2013, Meziane and Vadera, 2004, Geetha and Anandha Mala, 2014].

$$j^{-1}(T_b) = b \tag{3-7}$$

Al igual que en la ecuación 3-6, también podemos encontrar ejemplos en la literatura de posibles implementaciones de la función de la ecuación 3-7 como [Friedrich et al., 2011, Steen et al., 2010].

El esquema general de transformación de texto en lenguaje natural a código fuente resulta de reemplazar las ecuaciones 3-6 y 3-7 en la ecuación 3-8:

$$f(h^{-1}(T_e)) + g(j^{-1}(T_b)) = S_b + S_b \tag{3-8}$$

También es importante considerar que ambos modelos después de la transformación se encuentran en el mismo espacio: lenguaje natural restringido. Entonces la generación de código puede hacerse directamente desde la nueva representación en lenguaje natural restringido. Y más importante aún, se puede extender esta nueva representación (T_n) para incluir elementos que no están presentes en ninguna de las dos representaciones iniciales.

Por lo tanto, la hipótesis de esta tesis es que el código fuente generado desde esta nueva representación conjunta puede alcanzar un código más cercano al producto final que el código fuente generado desde los dos modelos originales. Esto se resume en las ecuaciones 3-9 y 3-10.

$$T = T_e + T_b + T_n \tag{3-9}$$

$$S_b + S_b < G(T) \tag{3-10}$$

Donde G es la función que genera código fuente a partir de esta representación en lenguaje natural restringido y T es la conjunción de los tres lenguajes restringidos.

Cabe anotar varios puntos respecto al modelo anterior:

- Los lenguajes T_e y T_b pueden ser lenguajes restringidos o sin restringir.
- Si T_e y T_b son lenguajes restringidos que puedan ser procesados automática y precisamente, los modelos intermedios e y b no son necesarios para generar código.
- Si e y b no son necesarios, las funciones 3-6 y 3-7 no son necesarias tampoco.
- El código fuente puede ser generado automáticamente desde una representación que puedan ser procesada de forma automática y precisa.

Finalmente, en esta tesis se implementará únicamente la función G puesto que las demás son innecesarias para el objetivo propuesto. A continuación se realiza una descripción detallada de los tres sub-lenguajes (T_e, T_b y T_n) que hacen parte del lenguaje natural restringido propuesto para escribir la especificación en el sistema de prototipado automático.

3.1.2. Lenguaje natural restringido para modelos E-R (T_e)

Para esta representación textual restringida de un modelo E-R se propone una gramática independiente de contexto, de acuerdo al marco conceptual presentado en la sección 2.5. En

Cualquier serie de palabras que comiencen en mayúscula:
fragmento_id : [A-Z][a-zA-Z0-9]+
Identifier: fragmento_id+

Figura 3-2: Token común para toda la especificación

primer lugar, se define un token, como se muestra en la figura **3-2**, para cumplir la función de identificador para cualquiera de los elementos involucrados en un diagrama E-R (y también para modelos BPMN). Algunos ejemplos de este token se muestran en la figura **3-3**.

User, Customer, Sales Direction, Sal4l5l6l Direction2, Deployed Salesman

Figura 3-3: Ejemplos de token "Identifier"

También se define un elemento de la definición, como puede ser un campo de una tabla en el diagrama E-R. Es decir, se le asigna un nombre al campo y se le coloca un tipo de dato asociado, la gramática para esta producción se muestra en la figura **3-4**:

Type, Name: Identifier
DefinitionItem: ('a' | 'an' | 'many') Type 'named' Name

Figura 3-4: Símbolo No-terminal para campo del modelo E-R

Cabe resaltar que con el identificador *Tipo* se hace referencia a alguno de los tipos básicos soportados por los diagramas E-R o a alguna otra tabla en la especificación. En este último caso, este campo representa una relación. La aridad de las relaciones se determina con el texto que comienza el no terminal "ItemDeDefinicion", así: si es 'a' o 'an', esta relación es de 'uno a uno'; si es 'many', la relación es de 'muchos a uno' o de 'muchos a muchos', dependiendo de si la tabla a la que se referencia contiene una referencia hacia la tabla inicial o no.

Para la definición de la tabla se emplean los no-terminales que se muestran en la figura **3-5**. Esta definición agrupa los campos bajo el nombre de una tabla. Además, en la figura **3-6** se presenta un ejemplo de la especificación de una tabla en lenguaje natural restringido.

Typen, Table: Identifier
Definition: Table ‘is a’ Type ‘and needs:’ DefinitionItem (‘,’ DefinitionItem)* ‘.’

Figura 3-5: No-terminal para definir tablas

User is a Domain Class and needs: a String named Username, a String named Password.

Figura 3-6: Ejemplo de especificación de tabla

3.1.3. Lenguaje natural restringido para modelos BPMN (T_b)

La siguiente parte de la gramática se propone para que el lenguaje natural restringido tenga la capacidad de ejecutar flujos de procesos como los que se describen con los diagramas de proceso de los modelos BPMN. Cabe anotar que se pretende incluir la gran mayoría de los elementos de BPMN, pero algunos elementos quedan por fuera del alcance de esta tesis, como por ejemplo, las coreografías. Se está limitando BPMN a lo que puede ocurrir en una única organización. A continuación se presentan los elementos presentados en la sección 2.4, acompañados de su respectivo fragmento de la gramática propuesta para el lenguaje natural restringido.

Para representar los eventos de comienzo y finalización de un proceso se usa el fragmento de gramática mostrado en la figura **3-7**. Estos eventos enmarcan la definición de los procesos, por esto se usan como delimitadores de la especificación. El no terminal “Tasklist” se usa

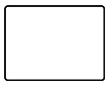
Evento de Inicio | Evento de final

ProcessName: Identifier
Tasklist: Asignedtask ((‘,’ Asignedtask)* ‘and’ Asignedtask)? ‘.’
Process : ‘the’ ProcessName ‘process starts, then’ ‘:’ Tasklist
Task: ’the process ends’

Figura 3-7: Fragmento de gramática para definir eventos de inicio y final de procesos

para mantener una lista ordenada de todos los elementos entre el principio y la finalización de la especificación del proceso.

Las tareas son la parte de BPMN en la que se hace algún trabajo. Estas tareas representan (idealmente) unidades de trabajo atómicas que se realizan por un individuo en la organización. La figura **3-8** muestra una tarea de BPMN y el fragmento de gramática asociado para describir una tarea básica.



Taskname: Identifier
Task: Taskname
Task: 'go back to' Taskname

Figura 3-8: Fragmento de gramática para definir tareas

Cada tarea tiene un nombre que, gramaticalmente corresponde al no-terminal **VP** del inglés. Este no terminal es la parte que describe la acción a realizar en una **Verb Phrase** [Chinosi and Trombetta, 2012]. Nótese que el flujo de salida de la tarea también puede llevar a una tarea anteriormente definida. Para considerar este caso, el no-terminal "Tarea" tiene una alterativa en que permite esto con el token "go back to".

Para representar las compuertas exclusivas basadas en datos de BPMN, es necesario hacer dos salvedades: en primer lugar, los flujos de salida de la compuerta se representan en un párrafo aparte; y además, la parte que se realiza en el cuerpo de la "Lista de tareas" del flujo principal es la que redirige a esos párrafos. Si hay una tarea después de esta pregunta, se asume que habrá convergencia. La gramática que representa esta compuerta se muestra en la figura **3-9**. La regla de "Tarea" que devuelve a una tarea previa también aplica en este caso.

Otro tipo de compuerta que es muy utilizada en BPMN es la compuerta paralela. Ésta ejecuta todos los flujos de salida y luego espera que todos concluyan para poder continuar. Al igual que en las compuertas exclusivas la definición de los flujos de salida se realiza en un párrafo aparte. Si hay una tarea después de la redirección en una lista de tareas, el sistema debe esperar que las actividades de todos los flujos de salida terminen para continuar con esta tarea. La gramática para representar estas compuertas se encuentra en la figura **3-10**.

GatewayName,Answer: Identifier
Task: ‘ask’ ‘if’ GatewayName
ExclusiveGatewayControl: ‘if the answer to’ GatewayName ‘is’ Answer ‘then:’ Tasklist

Figura 3-9: Fragmento de gramática para definir compuertas exclusivas

GatewayName: Identifier
Task: ‘do at the same time the’ GatewayName ‘tasks’
ParallelGatewayControl: ‘for’ GatewayName ‘tasks’ ‘also’ ‘do:’ Tasklist

Figura 3-10: Fragmento de gramática para definir compuertas paralelas

La última compuerta implementada es la compuerta basada en eventos (figura **3-11**). Esta compuerta espera que ocurra un evento específico para activar la ejecución del flujo posterior. Al igual que las otras compuertas que se implementarán en este trabajo, ésta se encuentra

GatewayName,NombreEvento: Identifier
Task: ‘check’ ‘the’ GatewayName ‘event’
EventGatewayControl: ‘if’ ‘the’ ‘event’ GatewayName ‘is’ NombreEvento (‘signal’|‘message’|‘timer’) ‘then’ ‘:’ Tasklist

Figura 3-11: Fragmento de gramática para definir compuertas basadas en eventos

dividida en dos partes: una parte donde se llama esta compuerta y otra en donde se determina qué evento esperar y cuáles son sus flujos posteriores. La gramática usada para representar las compuertas basadas en eventos se encuentra en la figura **3-11**.

Los subprocesos son un tipo de actividad que se representa como una actividad atómica y también como un proceso completo, el cual describe la actividad con más detalle. Existe una restricción importante en los subprocesos que consiste en que los flujos no pueden salir de las fronteras del subproceso para comunicarse con tareas del proceso que los contiene. Al igual que las compuertas se hace el llamado al subproceso dentro de la lista de tareas y se describe el subproceso en un párrafo aparte. El fragmento de gramática encargado de estos subprocesos se encuentra en la figura **3-12**.

SubprocesName: Identifier
Task: ‘the’ SubprocesName ‘is made’
Subprocess: ‘the’ SubprocesName ‘subprocess’ ‘starts,’ ‘then:’ Tasklist

Figura 3-12: Fragmento de gramática para definir subprocesos

Es de resaltar que esta gramática es muy similar a la de procesos, puesto que un subproceso es exactamente eso, un proceso. Las características y funcionalidades del proceso persisten en los subprocesos.

Los eventos intermedios de BPMN son representados como se muestra en la figura **3-13**. Estos se refieren únicamente a los eventos intermedios, ya que los eventos de frontera son implementados de manera diferente. Estos eventos cumplen la función de pausar el proceso en un punto hasta que el evento esperado ocurra. Se colocan como parte de las tareas puesto que están en el flujo del proceso directamente. Expandir la lista de eventos soportados es posible colocando otra posibilidad al no-terminal “Tarea”.

Los eventos de frontera son eventos BPMN que pueden ocurrir durante la ejecución de una tarea. Estos pueden ser de dos tipos: de interrupción (que detienen la tarea y siguen con un flujo definido) y de no interrupción (que esperan que termine la tarea y luego ejecutan el flujo definido). La gramática que representa estos eventos se encuentra en la figura **3-14**. Al igual que en el caso anterior, los tipos de eventos son de mensaje, señal y tiempo. Además,

Evento mensaje | Evento de señal | Evento de tiempo

```
SingalName,MessageName: Identifier
Task: 'wait for the' SingalName 'signal'
Task: 'wait for the' MessageName 'message'
Number: Integer
Unity: ('minutes'|'hours'|'days')
Task: 'wait' Number Unity
```

Figura 3-13: Fragmento de gramática para definir eventos intermedios (mensaje, señal, tiempo)

es importante aclarar que estos eventos se agregan después de definir la tarea, es decir, que se colocan como un párrafo independiente en el lenguaje natural restringido.

```
NombreEvento, TaskName: Identifier
EventoFrontera: if 'the' NombreEvento ('signal'|'message'|'timer') 'arrives' 'while' 'doing'
TaskName ('stop' 'it'|'wait' 'to' 'complete') 'and' 'then' ':' Tasklist;
```

Figura 3-14: Fragmento de gramática para definir eventos de frontera

Nótese que la parte del no-terminal EventoFrontera que determina si interrumpe o no la tarea es "('stop' 'it'|'wait' 'to' 'complete')". A diferencia de las compuertas exclusivas, estos elementos no tienen una representación añadida a la lista de tareas.

Los 'Lanes' de BPMN representan un grupo de usuarios o un rol en una organización. La representación de estos se encuentra en la figura **3-15**. De no especificarse un 'Lane', la tarea se deja como pública para que cualquier usuario esté en capacidad de realizarla. Si se define un rol específico para una tarea, todas las tareas posteriores serán restringidas a este rol.

Rol: Identifier
Asignedtask: Tarea (‘(’ ‘by’ Rol)’)?

Figura 3-15: Fragmento de gramática para definir la asignación de tareas

3.1.4. Lenguaje natural restringido para cada tarea (T_n)

La última parte del lenguaje natural restringido propuesto consiste en la definición de las acciones a realizar en las tareas de cada proceso. Esto es necesario para generar el código del prototipo deseado objetivo de esta tesis. Esta definición debe involucrar las operaciones CRUD (Create, Read, Update o Delete), y debe involucrar también, una o más de las clases de dominio definidas en T_e. Además, dos operaciones extra son definidas para poder operar sobre más de una instancia de las clases de dominio al mismo tiempo. También se incluyen operaciones de selección para lograr comportamientos atados entre dos tareas separadas. La gramática que soluciona estos requisitos se presenta a continuación.

Primero, se define la gramática para seleccionar qué tarea es la que se va definir, como se muestra en la figura **3-16**. Este fragmento selecciona la tarea con este nombre y añade una lista de actividades a ésta.

TaskName: Identifier
RedefinicionDeTarea: TaskName ‘is a task where :’ Activities;

Figura 3-16: Fragmento de gramática para especificar la definición de tareas

Cada una de las actividades puede ser de 4 tipos. El primero de estos es la creación, el cual permite crear instancias de clases de dominio, también permite restringir los campos que se pueden llenar y permite la creación de una sola o de múltiples instancias. El segundo de estos tipos es la visualización múltiple, que permite mostrar una lista de las instancias

de las clases de dominio, también es posible mostrar únicamente las instancias de clases que se relacionan con otra. El tercer tipo permite ver una única instancia. Por último, el cuarto tipo permite ver una lista restringida de campos de una sola instancia. La gramática para estos 4 tipos se presenta en la figura **3-17**.

```
Activities: AbstactActivity+
AbstactActivity: AbstactActivity
AbstactActivity: MultipleView
AbstactActivity: SingleView
AbstactActivity: FieldView

DomainClass, ChildClass, ParentClass, Field : Identifier
Creation: '-' ('multiple'—'a') DomainClass ('are'—'is') 'created' FieldRestrictor? '.';

MultipleView: '- all the' ChildClass ('in' ParentClass)? 'are' 'shown' FieldRestrictor?
OperationAdition? '.'

SingleView : '- a' DomainClass 'is shown' OperationAdition? '.';
FieldView : '-' 'the' Field ((','—'and') Field)* 'in' DomainClass 'are' 'shown' Operatio-
nAdition? '.';
```

Figura 3-17: Fragmento de gramática para definición de actividades

Por último, se añaden restricciones de los campos que podrán ser visualizados en cada tipo de vista y se añaden capacidades a cada vista, por ejemplo, pudiendo seleccionar muchas instancias de una lista, editar las instancias, borrarlas o seleccionar únicamente una de ellas. La gramática de estas restricciones se puede ver en la figura **3-18**.

```
FieldRestrictor : ',' 'only' Field (',' Field)* 'are' 'shown';
Operations :'edition'—'deletion'—'multiple' 'selection'— 'single' 'selection'
OperationAdition: ',' 'with' Operations ((','—'and') Operations)* 'capabilities';
```

Figura 3-18: Fragmento de gramática para definir operaciones y restricciones

Con este último sub-lenguaje es posible prototipar los contenidos de las tareas, y de esta forma, conseguir que el prototipo tenga una funcionalidad básica. Cabe resaltar que las acti-

vidades de estas tareas no tienen la misma connotación semántica que las tareas de BPMN, sino que deben entenderse como las operaciones que deben llevarse a cabo al interior de una tarea específica. Por ejemplo: revisar el trabajo de otro rol en la organización, actualizar información de los registros, o crear nueva información de las nuevas entradas de datos de la organización. Por último, es importante notar que más de una actividad es permitida por tarea, logrando así una funcionalidad mayor, como por ejemplo: mostrar únicamente los datos de otra clase de dominio mientras se editan los de otra, o lograr editar los datos de una persona y todos los vehículos que tiene asociados.

3.2. Generación de código fuente

Como se muestra en la figura **3-1**, después de que el diseñador en conjunto con los *stakeholders* escribe la especificación usando el lenguaje natural restringido, ésta se usa para generar el código fuente del prototipo de software. Para llevar a cabo la generación de código se sigue el esquema que se presenta a continuación en la figura **3-19**.

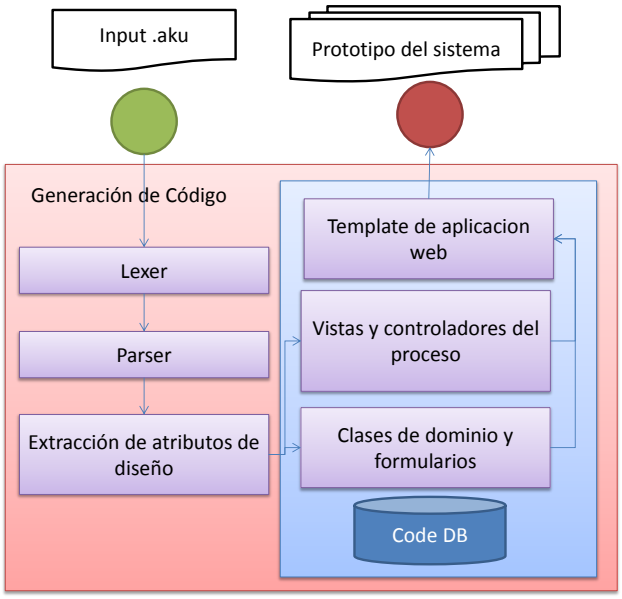


Figura 3-19: Proceso de generación de código

Tomando como entrada la especificación escrita en lenguaje natural restringido, e.g., Input.aku, se da inicio al proceso de generación de código fuente. Los primeros pasos corresponden al análisis léxico (lexer) y sintáctico (parser), al igual que en un compilador tradicional

como se muestran en la sección 2.5. Estas dos primeras etapas entregan como resultado el árbol sintáctico de la especificación. A continuación se extrae la información de diseño de este árbol, es decir, los nombres de las clases, los nombres de las tareas y las acciones a ser realizadas por tarea, entre otros.

La generación de código fuente se realiza con la ayuda de una serie de plantillas organizadas en una base de datos. El diagrama entidad-relación de esta base de datos se ilustra en la figura **3-20**. Cada especificación se asocia con una serie de códigos fuente de controladores y de vistas. También, se encuentra el código fuente que se usa de manera genérica para insertar el código resultante de las otras generaciones. Por ejemplo, el código asociado a un campo de una tabla del modelo E-R, se encontraría asociado con una instancia de la clase "Spec" cuyo nombre represente el tipo del campo. Al igual sucede con los tipos de tareas (los subprocesos, compuertas y eventos) ya que éstos deben poseer una especificación en esta base de datos y sus respectivos códigos fuente.

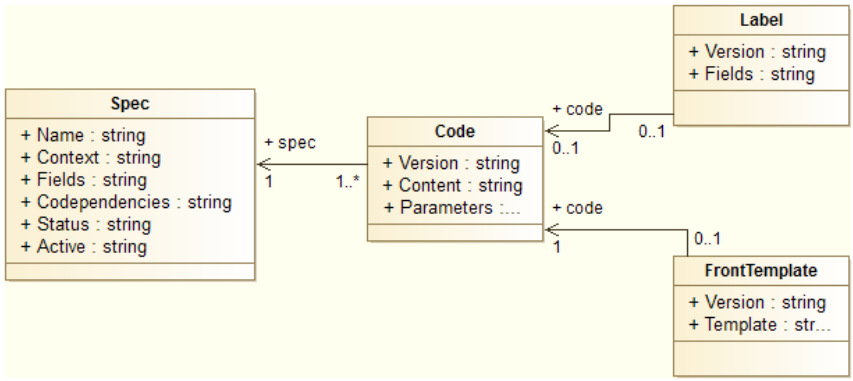


Figura 3-20: Diagrama de clases de plantillas

Es importante anotar que gracias a que este lenguaje es restringido, posee una única interpretación y el procesamiento del texto puede realizarse con cualquier herramienta que genere este tipo de procesadores de lenguajes de programación.

Después de estos procesos, el código para las clases de dominio y formularios es creado desde T_e. Para establecer en el código fuente las relaciones esperadas se clasifican las posibles relaciones en 4 casos:

1. Un atributo tiene un tipo que es un tipo básico de E-R.
2. Un atributo referencia otra clase en la especificación.
3. Un atributo tiene aridad múltiple (*'many'*) pero la tabla destino no tiene referencia a

la tabla inicial.

4. Un atributo tiene aridad múltiple y la tabla destino tiene una referencia a la tabla inicial.

En el primer caso, se usa la base de datos con el código fuente de este campo para generar el código deseado (puede ser un atributo de una clase que asocia un sistema ORM, una sentencia en SQL o lo que se determine en la implementación). En el segundo caso, se asigna el tipo del campo como una llave foránea que apunta a la llave primaria de la clase a la que referencia el atributo. En el tercer caso, se entiende como una relación de uno a muchos, entonces, se agrega un campo a la tabla destino que referencia a la tabla origen (esto se puede realizar con ayuda de un ORM o con una sentencia ALTER de SQL). Finalmente para el cuarto caso, se inserta una tabla adicional que referencia a las llaves primarias de las dos tablas relacionadas (una relación de muchos a muchos).

Posteriormente, se colocan los códigos fuente de cada campo en un código para la tabla. Este código puede tener una llave primaria por defecto dependiendo de la implementación. Por último, se colocan los códigos de todas las tablas resueltas en uno que sea capaz de crearlas durante el lanzamiento del prototipo.

A continuación, el código fuente de los procesos es generado. Esta tarea se puede realizar de varias maneras dependiendo de la implementación. Las dos implementaciones que se realizaron en este trabajo pueden describirse así:

1. Generar el código fuente para cada tarea, para que cuando cada tarea esté completa, ésta redirija a la siguiente.

2. Generar un método que controle estas transiciones entre tareas y llamarlo al final de cada ejecución exitosa de una tarea.

Por cada una de las tareas, se obtiene una plantilla de la base de datos y se resuelve. Si el controlador generado necesita una vista, se genera también el archivo .html de la vista. Si el tipo de la tarea es “TareaSimple”, se lleva a cabo el siguiente proceso para determinar qué debe generar:

1. Revisar si la tarea se define en otro párrafo de la especificación

2. Si la definición existe, entonces se resuelve el código de las actividades y se coloca en las plantillas de controlador y vista.

3. Si no existe, se usa un controlador y vista por defecto.

Si el no-terminal *DefinicionDeTarea* existe para la *TareaSimple*, entonces se usan fragmentos de código fuente predefinidos para colocar las actividades definidas en el controlador. Estos fragmentos se usan para buscar, mostrar, actualizar y crear las instancias de las tablas del modelo E-R.

Para generar el método de control del proceso, es decir, el método que toma la tarea actual y su respuesta para buscar la siguiente tarea, se usa la información contenida en el no-terminal "Tasklist". Cada vez que este no-terminal aparece en la definición de un proceso se toma la lista de tareas asociada y se emparenta cada tarea con su siguiente. Si hay un llamado a una compuerta o un subproceso, se realizan llamadas recursivas a las demás listas hasta completar el método que retorne para cada tarea del proceso una redirección a la siguiente tarea. Este método también ayuda a resolver las rutas de salida de las compuertas basándose en las acciones del usuario. Por último, también resuelve los eventos de frontera que estén asociados a cada tarea.

Para el caso de las compuertas, se genera en el prototipo, una vista con la pregunta y sus respectivas opciones en la especificación. Después de una solicitud POST del navegador se determina, basado en la respuesta del usuario, cuál es la siguiente tarea.

Finalmente, estas piezas de código fuente generadas se colocan en una platilla pre-definida de servicio web, que ejecuta los procesos del prototipo de software especificado en lenguaje natural restringido.

4 Implementación

Para mostrar la aplicabilidad de la propuesta y su funcionalidad se realizó el desarrollo de una herramienta de software para la generación automática de prototipos presentada en el Capítulo 3.

Para desarrollar esta herramienta se utilizó una metodología de desarrollo de software iterativa e incremental basada en el *framework* de SCRUM. Las características definidas para este desarrollo en el Capítulo 3 se usaron como historias de usuario o *features* en el *backlog* del producto, y se implementaron en *sprints* de dos semanas cada uno.

Primero es necesario hacer una diferenciación entre el proceso de generar el prototipo y el prototipo en sí mismo. Cada una de estas partes puede usar diferentes lenguajes y *frameworks* para ejecutar las tareas requeridas. De esta forma, se realizaron dos implementaciones (o *releases*) diferentes a lo largo de esta tesis:

1. Una que realiza el procesamiento del lenguaje natural restringido propuesto usando Python (ANTLR) y genera código en Python.
2. Una que realiza el procesamiento del lenguaje natural restringido propuesto en Java (Xtext) y genera código en Python.

Una vez terminado el desarrollo de la primera implementación, se propusieron varias mejoras que fueron tenidas en cuenta en la segunda iteración del desarrollo. La segunda versión, por su parte, reúne todas las funcionalidades de la primera versión y las mejoras propuestas, y además, permite la integración con el entorno integrado de desarrollo (IDE) Eclipse[1].

4.1. Implementación en ANTLR + Python

El esquema general de generación de código fuente para esta implementación se muestra en la figura **4-1**. En este esquema se muestran los mismos pasos que se encuentran en el esque-

[1] https://eclipse.org/

ma general presentado anteriormente en la figura **3-19**. Además, se incluye una fase extra ("Alteración de atributos de diseño") para facilitar la generación de código posteriormente.

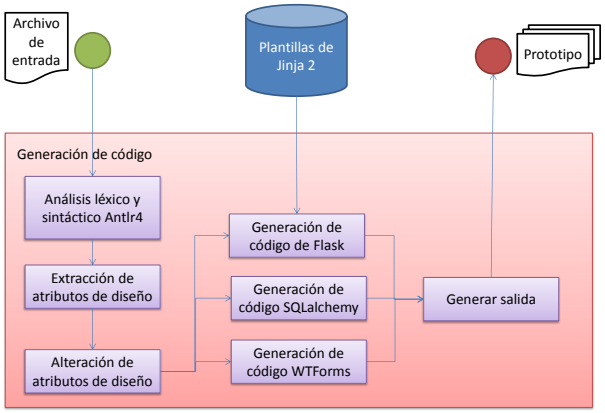


Figura 4-1: Flujo de trabajo para la generación de código

Para esta primera implementación se usó el lenguaje de programación Python y el generador de analizadores sintácticos ANTLR. Este analizador provee un mecanismo para implementar aplicaciones de procesadores de lenguajes de programación basado en el patrón de diseño conocido como *listeners*. Esto facilita recorrer el árbol sintáctico y ejecutar las acciones semánticas correspondientes para cada uno de los símbolos no terminales de la gramática. El objeto encargado de recorrer el árbol sintáctico desencadena eventos a medida que se va realizando el análisis sintáctico durante el análisis del texto de entrada. Estos métodos son similares a los que se podrían encontrar en las aplicaciones orientadas a las interfaces gráficas, por ejemplo, al dar *click* en un botón. Se implementó un método *listener* para cada una de las reglas de la gramática del lenguaje natural restringido descrito en la sección 3.1. Un ejemplo de estos métodos se presenta en la figura **4-2**.

```
def exitEntity(self, ctx):
    ctx.payload ={'type':'entity','name':ctx.children[1].payload,'fields':ctx.children[4].payload}
        registered_clases.append(ctx.children[1].payload["name"])
```

Figura 4-2: Ejemplo de *listener* en Python

El árbol de sintaxis resultado de la ejecución del análisis sintáctico se usa junto con estos *listeners* para extraer la información de diseño. Esta información es entonces almacenada en un diccionario para su posterior uso.

En la fase de alteración de atributos de diseño se realizan las siguientes modificaciones al diccionario anteriormente obtenido. Las referencias entre las tablas del modelo E-R son

resueltas como se describió anteriormente en esta propuesta. Es de notar que es necesario transformar entre representaciones singulares y plurales del mismo sustantivo. Estas transformaciones pueden ser realizadas usando una aproximación basada en reglas y diccionarios para los sustantivos irregulares. También se agregan a los campos de las tablas las llaves foráneas correspondientes a las relaciones detectadas en la definición de la entrada.

Para las compuertas y subprocesos que son referenciados en alguno de los párrafos de la especificación, se busca la definición de cada uno de estos en el documento y se coloca en el diccionario justo donde estas compuertas son llamadas desde una lista de tareas.

Cuando la información de diseño está lista para ser transformada en código fuente, una serie de plantillas de código previamente escritas son usadas para transformar esta información de diseño en código fuente. Para las tablas del modelo E-R cada campo es mapeado a una línea de código fuente compatible con *SQLalchemy*[2] (un sistema ORM), y también a una sola línea compatible con *WTForms*[3] (un procesador de formularios web). Finalmente, una traducción de T_b es realizada para convertirla en código fuente compatible con *Flask*[4] (un marco de trabajo para aplicaciones web).

Para conseguir generar el código fuente de proceso (S_b) se realizan las siguientes actividades: para cada tarea, subproceso, compuerta o evento en la especificación se generan desde plantillas una serie de vistas (generalmente sólo una) y un conjunto de controladores (también generalmente uno). En la figura **4-3** se puede ver un ejemplo de una plantilla de un controlador y en la figura **4-4** se puede apreciar una plantilla de una vista.

Para las compuertas paralelas es indispensable notar que éstas deben almacenar más de una tarea en los datos del proceso y llevar el control de cada hilo de ejecución y esperar a la finalización de todos los hilos anteriormente iniciados. Esto se puede lograr controlando que al finalizar una tarea, ésta se intercambie con su sucesora hasta que todos los flujos que divergieron tengan como sucesor una compuerta de convergencia.

Para los subprocesos se realiza exactamente el mismo procesamiento que se realiza para los procesos, verificando que las referencias que se hacen en la sentencia *"go back to"* no escapen de las fronteras del subproceso.

Todos los códigos generados para cada elemento de la especificación son entonces concatenados en una sola cadena y se colocan en una plantilla preparada previamente para ejecutar estos controladores. Los archivos de las vistas son creados para luego, en tiempo de ejecución

[2]http://www.sqlalchemy.org/
[3]https://wtforms.readthedocs.io/
[4]http://flask.pocoo.org/

```
@app.route('{{path}}', methods=['GET','POST'])
def {{method_name}}(pid):
    if request.method=='POST':
        update_process(pid,str(request.url_rule),'{{next_task}}')
        return redirect('{{next_task}}'.replace('<pid>',pid))
    template = env.get_template('{{template_name}}')
    ent_list = session.query({{model_name}}).all()
    return template.render(ent_list=ent_list)
```

Figura 4-3: Ejemplo de platilla de controlador

```
<body>
    <div id="wrapper">
        <form role="form" action="" method="post" name="lform" targe="_blank">
            <div class="col-lg-3 col-md-4 col-md-offset-4">
            <fieldset>

                 {% for field in form.__iter__() %}
                    <div class="form-group">
                    {{field.label}} <br/>
                    {{field(class="form-control")}}
                    </div>
                 {% endfor %}
                <input type="submit" value="Marcar como Revisado" name="Revisado"/>
            </fieldset>
            </div>
        </form>
     </div>
</body>
```

Figura 4-4: Ejemplo de plantilla de vista (recortada)

ser consultados por el sistema de plantillas propio del prototipo en ejecución.

Además, en esta primera implementación se intentó llevar a cabo una idea que posteriormente se descartó. Esta idea consistía en aprovechar el hecho de que las tareas de BPMN se deben escribir usando *Verb Phrases*. Este no-terminal del inglés (**VP**) tiene la particularidad de definir una acción sobre un objeto. Un ejemplo de una de estas tareas puede ser "Stack users". Con ayuda de este ejemplo se puede ilustrar el hecho que es posible definir una plantilla para el verbo "Stack" (apilar) y alimentarla con la información de la clase predefinida en el modelo E-R de "User". Se buscaba generar un modelo más cercano al producto final e incluir información semántica sobre la actividad directamente sobre el texto del proceso. Esta idea se implementó y un ejemplo de la plantilla para el verbo *list* se presenta en la figura **4-5**. Sin embargo, la principal desventaja de esta idea es que sería necesario implementar una plantilla para cada verbo.

```
class {{model_name}}_listForm(Form):
    lista = FieldList(FormField({{model_name}}Form), min_entries=5)

@app.route('{{path}}', methods=['GET','POST'])
def {{method_name}}(pid):
    form = {{model_name}}_listForm(request.form)
    if request.method=='POST':
        for ent in form.lista.entries:
            obj= {{model_name}}()
            ent.form.populate_obj(obj)
            session.add(obj)
        session.commit()
        update_process(pid,str(request.url_rule),'{{next_task}}')
        return redirect('{{next_task}}'.replace('<pid>',pid))
    template = env.get_template('{{template_name}}')

    return template.render(form=form)
```

Figura 4-5: Ejemplo de plantilla para *verb phrases*: para el verbo *list*

Cabe notar que debe haber dos mecanismos clave: uno para determinar que tareas ya han sido traducidas a código fuente; y otro para aislar las tareas que se encuentran dentro de la definición de un subproceso. Ambos pueden ser fácilmente tratados con un par de diccionarios adicionales. También, se implementaron métodos adicionales para determinar y almacenar en qué tarea del proceso se encuentra una determinada instancia del proceso y métodos generales para guardar y consultar información sobre cada instancia del proceso.

El sistema de plantillas usado en esta implementación fue *Jinja2*[5]. Las plantillas usadas por este sistema fueron almacenadas en una base de datos SQLite3[6]. Para acceder a esta base de datos y editar las plantillas, fue creado y configurado un servicio web para servir en el puerto 5001. La aplicación web generada queda configurada para funcionar en el puerto 5000. Este servicio web (el prototipo funcional) se recarga automáticamente al alterarse la especificación y generar nuevamente el código. Un diagrama ilustrando este comportamiento se muestra en la figura **4-6**.

En resumen, las herramientas utilizadas para el desarrollo de esta implementación fueron:

- Lenguaje de programación: *Python 2.7*.
- Generador de analizadores sintácticos: *ANTLR v4*.
- Sistema de plantillas: *Jinja 2*.

[5]http://jinja.pocoo.org/docs/dev/
[6]https://www.sqlite.org/

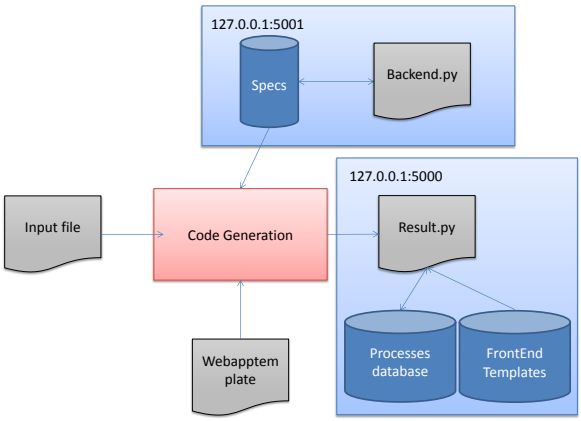


Figura 4-6: Flujo de trabajo para la generación de código

- *Framework* de aplicaciones web: *Flask*.
- Sistema *ORM*: *SqlAlchemy*.
- Procesador de formularios web: *WTforms*.
- Tecnologías de front-end (*HTML*, *CSS* y *JS*): *Bootstrap*[7] y el tema *SB Admin 2*[8] admin theme.
- Base de datos: *SQLite3*.

Es importante aclarar que la gramática de ANTLR que se implementó para los modelos E-R (T_e) difiere de la que se propone en la sección 3.1, pero contiene la misma información de diseño. Esta gramática puede considerarse como la primera versión de la gramática definitiva y se muestra en la figura **4-7**.

```
id:  FID (FID)*;
entity: ('a'|'an') id 'has' ':' proplist;
proplist: prop ((',') prop)* 'and' prop)?;
prop: ('a'|'an') FID FID? | ('a set of') FID;
FID: [A-Z][a-zA-Z0-9]*;
```

Figura 4-7: Gramática para la versión de ANTLR para modelos E-R: T_e

La gramática que se implementó para los modelos BPMN (T_b) en esta versión (que se

[7]http://getbootstrap.com/
[8]https://startbootstrap.com/template-overviews/sb-admin-2/

muestra en la figura **4-8**) carece de algunos de los elementos propuestos en la sección 3.1 ya que ésta era la primera iteración de la implementación. Por ejemplo, los eventos de frontera no se implementan al igual que las compuertas basadas en eventos. Estos elementos fueron agregados en la implementación posterior.

```
process: 'the' id 'process' 'stars' ',' 'then' ':'  tasklist;
task: id #simpleTask
        |'wait' NUMBER ('week'|'weeks') #afterEvent
        |'go' 'back' 'to' id #gotoJump
        |'the' 'process' 'ends' #endEvent
        |'the' 'subprocess' 'ends' #subprocessendEvent
        |'the' id 'is' 'made' #subprocessCall
        |'go' 'to' 'ask' 'if' id #jumpToAsk
        | 'ask' 'if' id #questionredirect
        |'do' 'at' 'the' 'same' 'time' 'the' id 'tasks' #parallelredirect;
control : 'if' 'the' 'answer' 'to' id 'is' (FID|NUMBER) 'then' ':' tasklist #ifpath
        |'for' id 'tasks' ('also')? 'do'':' tasklist #parallelpath;
subprocess: 'the' id 'subprocess' 'stars' ',' 'then' ':'  tasklist;
asignedtask: task ('' id '')?;
tasklist: task ((',' task)* 'and' task)? '.';
```

Figura 4-8: Gramática para la versión de ANTLR para BPMN: T_e

La gramática (T_n) no se había implementado en esta versión puesto que ésta se propuso posteriormente para solucionar el problema de los verbos que se identificó en esta primer implementación.

4.2. Implementación en Xtext + Python

Esta segunda implementación corresponde a un avance con respecto a la implementación de la sección 4.1. En esta implementación se agregan no terminales a la gramática y se reemplazan algunos elementos de la gramática previamente propuesta para mejorar el lenguaje restringido y las capacidades de prototipado y de expresividad de éste.

El esquema de generación de código fuente de esta segunda implementación se muestra en la figura **4-9**. En este esquema se muestran los mismos pasos que se encuentran en el esquema general que se mostró en la figura **3-19**. Varios de los pasos se encuentran integrados en uno solo dado que las capacidades de la herramienta que genera analizadores léxicos y sintácticos (*Xtext*[9]) permiten comprimir varias de estas fases.

Para el proceso de generar el prototipo del sistema de información transaccional se usó el

[9]http://www.eclipse.org/Xtext/

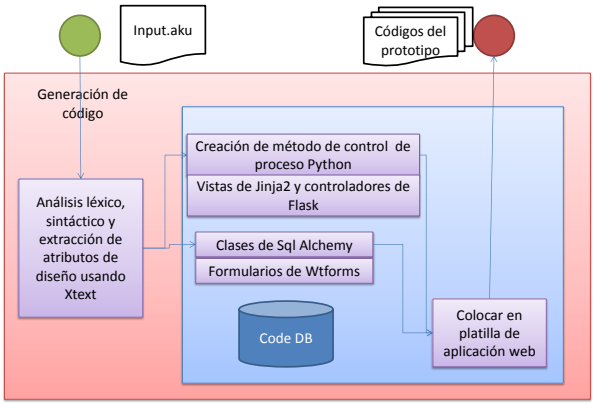


Figura 4-9: Flujo de trabajo para la generación de código

conjunto de herramientas que provee *Xtext*, incluyendo el lenguaje de programación *Xtend*. Este generador de analizadores usa como insumo la definición de una gramática y genera como resultado una serie de elementos:

1. El analizador léxico.
2. El analizador sintáctico.
3. Una serie de clases que contienen las *features* de cada no-terminal.
4. Un código fuente guía para generar el traductor orientado por la sintaxis como se muestra en la sección 2.5.4.
5. Una serie de sub-proyectos entre los cuales se encuentran uno de pruebas unitarias y uno para eclipse que integra esta plataforma con el lenguaje propuesta. Este último integra funcionalidades de autocompletado, marcado de sintaxis y otras funcionalidades para el lenguaje natural restringido.

Aprovechando estas características de Xtext se implementan las tres gramáticas: T_e que se muestra en la figura **4-10** (E-R), T_b que se muestra en la figura **4-12** (BPMN), y por último, T_n que se muestra en la figura **4-11** (tareas).

Estas gramáticas tienen una peculiaridad que vale la pena resaltar: el término que aparece antes del igual es el nombre de la *feature*, en este caso de la información de diseño. Este nombre es un insumo para Xtext y le permite generar una clase para este no-terminal que contendrá la información de diseño que se necesita para generar el código fuente. Un ejemplo de las clases con información de diseño que genera Xtext puede ser visto en la figura **4-13**

```
terminal FID :  '^'?('A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
Identifier : parts+=FID (parts+=FID)*;
DefinitionItem: arity=('a'|'an'|'many') specName=Identifier 'named' intanceName=Identifier
    'with' defaultName=Identifier 'as' 'default';
Definition: childname=Identifier 'is' 'an' specName=Identifier ('which' 'exist' 'in' contextName=Identifier)?
    'and' 'needs' ':' definitions+=DefinitionItem (',' definitions+=DefinitionItem)* '.';
```

Figura 4-10: Gramática de T_e para la segunda implementación

```
TaskRedefinition:taskName=Identifier 'is' 'a' 'task' 'where' ':' partslist=ViewPartsList;
ViewPartsList: list+=AbstractOperation+;
AbstractOperation: operation=(Creation|MultipleView|SingleView|FieldView);
FieldRestrictor : ',' 'only' fields+=Identifier (',' fields+=Identifier)* 'are' 'shown';
Operations :'edition'|'deletion'|'multiple' 'selection'| 'single' 'selection';
OperationAdition: ',' 'with' permisions+=(Operations)
   ((','|'and') permisions+=(Operations))* 'capabilities';
Creation: '-' arity=('multiple'|'a') name = Identifier ('are'|'is')
   'created' (restriction=FieldRestrictor)? '.';
MultipleView: '-' 'all' 'the' child = Identifier ('in' parent = Identifier)? 'are' 'shown'
(restriction= FieldRestrictor)?
(operations=OperationAdition)? '.';
SingleView : '-' 'a' name = Identifier 'is' 'shown' (operations=OperationAdition)? '.';
FieldView : '-' 'the' fields+=Identifier ((','|'and') fields+=Identifier)* 'in' parent = Identifier 'are'
    'shown' (operations=OperationAdition)? '.';
```

Figura 4-11: Gramática de T_n para la segunda implementación

para este caso del no-terminal *ParallelControl*.

En este punto están completas las tres primeras tareas del proceso de propuesto en la figura **3-19**. Xtext otorga un complemento para Eclipse que nos permite mostrar el lenguaje restringido con sintaxis resaltada. Adicionalmente se desarrolla un sistema de auto-completado y ayuda contextual. Una muestra de esta sintaxis resaltada se muestra en la figura **4-14**.

La siguiente tarea en el proceso de generación de código consiste en generar el código de respaldo desde la representación restringida T_e. El primer paso a seguir es resolver las referencias entre tablas del modelo E-R determinando el caso al que corresponden de acuerdo a lo presentado en la sección 3.2. Al igual que en la primera implementación es necesario realizar transformaciones entre las representaciones singulares y plurales de los sustantivos que cumplen la función de nombrar algún elemento de la especificación. Nuevamente, la aproximación implementada es utilizar un diccionario para los sustantivos irregulares y reglas de construcción para todos los demás sustantivos.

Luego de resueltas estas referencias se procede a utilizar las plantillas de la base de datos para generar el código de cada uno de los campos de las tablas. Estas plantillas están desarro-

```
//------Process and Start Event------
Tasklist: taskList+=Asignedtask ((',' taskList+=Asignedtask)* 'and,' taskList+=Asignedtask)? '.';
Process: 'the' name=Identifier 'process' 'starts' ',' 'then' ':' taskList=Tasklist;
//------End Event------
EndEvent: 'the' 'process' name='ends';
//------Tasks------
SimpleTask: name=Identifier;
GotoJump:'go' 'back' 'to' returnTo=Identifier;
Task:task=(SimpleTask|AfterEvent|WaitSignalEvent|WaitMessageEvent|GotoJump|EndEvent|SubprocessendEvent|
    SubprocessCall|JumpToAsk|QuestionRedirect|EventRedirect|ParallelRedirect);
//------Exclusive Gateways------
QuestionRedirect: 'ask' 'if' redirectTo=Identifier;
QuestionControl : 'if' 'the' 'answer' 'to' questionName=Identifier 'is' answer=FID 'then' ':' taskList=Tasklist ;
//------Parallel Gateways------
ParallelRedirect: 'do' 'at' 'the' 'same' 'time' 'the' redirectTo=Identifier 'tasks';
ParallelControl: 'for' redirectName=Identifier 'tasks' ('also')? 'do'':' taskList=Tasklist;
//------Event Gateways------
EventRedirect: 'check' 'the' redirectTo=Identifier 'event';
EventControl : 'if' 'the' 'event' eventName=Identifier 'is'
eventCatched=Identifier ('signal'|'message'|'timer') 'then' ':' taskList=Tasklist ;
//------Subprocess------
SubprocessCall:'the' subprocessName=Identifier 'is' 'made';
Subprocess: 'the' name=Identifier 'subprocess' 'starts' ',' 'then' ':' taskList=Tasklist;
//------Events------
WaitMessageEvent:'wait' 'for' 'the' messageName=Identifier 'message';
AfterEvent:'wait' duration=INT unit=('week'|'weeks'|'hour'|'hours'|'minutes'|
'minute'|'days'|'day');
WaitSignalEvent:'wait' 'for' 'the' signalName=Identifier 'signal';
//------Lanes------
Asignedtask: task=Task ('(by' asignee=Identifier ')')?;
//------Frontier events------
FronterEvent: 'if' 'the' name=Identifier ('signal'|'message'|'timer') 'arrives' 'while' 'doing' task=Identifier
    ('stop' 'it'|'wait' 'to' 'complete') 'and' 'then' ':' taskList=Tasklist;
```

Figura 4-12: Gramática de T_b para la segunda implementación

lladas usando el *framework* Freemarker[10], se almacenan en una base de datos SqlLite[11] y son consultadas por Xtext mediante el ORM OrmLite[12]. Un ejemplo de estas plantillas puede ser visto en la figura **4-15**. Luego se procede a colocar los resultados de estas transformaciones en una plantilla para definir la tabla en una clase de *SQLalchemy*[13]. Adicionalmente, en esta misma plantilla se coloca una definición de una clase auxiliar que contiene el código de un formulario web *WTForms*[14]; esta vez apoyándose en un plugin[15] de esta librería que permite

[10]http://freemarker.org/
[11]https://www.sqlite.org/
[12]http://ormlite.com/sqlite_java_android_orm.shtml
[13]http://www.sqlalchemy.org/
[14]https://wtforms.readthedocs.io/
[15]https://github.com/wtforms/wtforms-sqlalchemy

```
/**generated by Xtext 2.10.0*/

public class ParallelControlImpl extends MinimalEObjectImpl.Container implements ParallelControl
{
  protected Identifier redirectName;
  protected Tasklist taskList;
  protected ParallelControlImpl(){
    super();
  }
  public Identifier getRedirectName() {
    return redirectName;
  }
  public Tasklist getTaskList() {
    return taskList;
  }
} //ParallelControlImpl
```

Figura 4-13: Ejemplo de clase con información de diseño generada por Xtext (recortada)

```
the Project List process stars, then: List Projects, Show Selected Project and, the process ends.

List Projects is a task where:
-all the Projects are shown, with single selection capabilities.

Show Selected Project is a task where:
-a Project is shown.
-all the Tasks in Project are shown.
```

Figura 4-14: Sintaxis resaltada obtenida de Xtext

relacionar la clase original de SQLAlchemy a un formulario sin tener que definir cada campo. La plantilla para estas clases se puede ver en la figura **4-16**.

```
${spec.intanceName.asVariableName()} = Column(String(255))
```

Figura 4-15: Ejemplo de plantilla de campo

Una vez finalizada la creación del código fuente de las clases de dominio se procede a generar el código de los procesos. Nuevamente se realiza para cada compuerta, tarea, subproceso o evento una generación de código basándose en una plantilla. Para la mayoría de estos elementos estos reemplazos simples son suficientes. Por ejemplo, para el no-terminal *"GotoJump"* una plantilla como la que se muestra en la figura **4-17** es suficiente.

Si en la lista de elementos del proceso aparecen en algún momento compuertas o subprocesos, se realiza una búsqueda por la estructura de la especificación buscando todos los posibles flujos o definiciones de subproceso y se genera el código fuente para éstos también. Como se puede ver, aquí no existe la fase de alteración de la información de diseño.

```
class ${spec.childname.asClassName()}(Base):
    __tablename__ = '${spec.childname.asVariableName()}'
    id = Column(Integer, primary_key=True)
    <#list definitions as child>
${child}
    </#list>
    def __str__(self):
        return getattr(self,'name',str(self.id))

class ${spec.childname.asClassName()}Form(ModelForm):
    class Meta:
        model = ${spec.childname.asClassName()}
<#list spec.definitions as d>
<#if d.inSpec()>
    def ${d.specName.asClassName()}_helper_method():
        return session.query(${d.specName.asClassName()}).all()
    ${d.intanceName.asVariableName()}_id = QuerySelectField2(
        query_factory=${d.specName.asClassName()}_helper_method,
        allow_blank=True
    )
</#if>
</#list>
```

Figura 4-16: Ejemplo de plantilla de clases de dominio

```
@app.route('/${r"${procesname}"}/<pid>/redirect/${data.returnTo.asMethodName()}', methods=['GET'])
def ${r"${procesname}"}_redirect_${data.returnTo.asMethodName()}(pid):
    return  redirect(${r"${procesname}"}_resolve_next_task(request.url_rule,pid,None))
```

Figura 4-17: Plantilla para el no-terminal *"GotoJump"*

El proceso de generar estos controladores cambia si existe un símbolo no-terminal *"TaskRedefinition"* cuyo campo *"taskName"* coincida con alguna de las tareas definidas como *"Simpletask"*. Si éste existe entonces el proceso usa plantillas de las actividades a desarrollar en la tarea para generar el respectivo controlador en Python y la vista en HTML. Un ejemplo de estas plantillas de actividades se muestra en la figura **4-18**, en este caso para el no-terminal *"Creation"*. También, se muestra en la figura **4-19** la plantilla usada para generar la parte de la vista que corresponde a esta actividad.

Después de esta generación de código de cada elemento de los procesos, éstos se recorren nuevamente para generar el método de control de flujo de actividades como se mostró en la sección 3.2.

Por último, después de que todo el código fuente es generado, éste se coloca en una única plantilla preparada pare ejecutar este programa y lanzar el servicio web que contiene el

```
<#if creation.arity="a">
form${creation.name.asClassName()} = ${creation.name.asClassName()}Form(
    prefix='${creation.name.asClassName()}',formdata=request.form
)
viewparams["form${creation.name.asClassName()}"]=form${creation.name.asClassName()}
if(request.method=='POST'):
    ${creation.name.asVariableName()} = ${creation.name.asClassName()}()
    form${creation.name.asClassName()}.populate_obj(${creation.name.asVariableName()})
    session.add(${creation.name.asVariableName()})
    session.commit()

</#if>
<#if creation.arity="multiple">
class ${creation.name.asClassName()}_listForm(Form):
    lista = FieldList(FormField(${creation.name.asClassName()}Form), min_entries=5)
${creation.name.asVariableName()}_form_list = ${creation.name.asClassName()}_listForm(
    prefix='${creation.name.asClassName()}_list',formdata=request.form
)
viewparams["${creation.name.asVariableName()}_form_list"]=${creation.name.asVariableName()}_form_list

if request.method=='POST':
    for ent in ${creation.name.asVariableName()}_form_list.lista.entries:
        ${creation.name.asVariableName()}_instance= ${creation.name.asClassName()}()
        ent.form.populate_obj(${creation.name.asVariableName()}_instance)
        session.add(${creation.name.asVariableName()}_instance)
    session.commit()

</#if>
```

Figura 4-18: Plantilla del controlador para actividad *"Creation"*

prototipo. Nuevamente con un solo click basta para lanzar el servicio en el puerto 5001. Este servicio funciona con una base de datos Sqlite que se crea en la primera ejecución. Además, contiene plantillas propias que resultan del proceso de generar las vistas. Estas plantillas están hechas con el *framework Jinja2*[16].

Gracias al plugin generado por Xtext basta con modificar la especificación y guardarla para que automáticamente se vuelva a generar el prototipo y se puedan ver los cambios en el navegador.

Para esta segunda implementación de la propuesta de esta tesis se usaron las siguientes herramientas:

- Lenguajes de programación: *Python 2.7, Xtend*
- Generador de analizadores: *Xtext*

[16]http://jinja.pocoo.org/docs/dev/

```
<#if  creation.arity=='a' >
<#if creation.restriction??>
<#list creation.restriction.fields as field>
<div class="col-sm-6"><div class="form-group">
    {{form${creation.name.asClassName()}.${field.asVariableName()}.label}} <br/>
    {{form${creation.name.asClassName()}.${field.asVariableName()}(class="form-control")}}
</div></div>
</#list>

<#else>
{% for field in form${creation.name.asClassName()}.__iter__() %}
<div class="col-sm-6"><div class="form-group">
    {{field.label}} <br/>
    {{field(class="form-control")}}
</div></div>
{% endfor %}
</#if>
</#if>

<#if  creation.arity=='multiple' >
<#if creation.restriction??>
<table>
{% for ent in ${creation.name.asVariableName()}_form_list.lista.entries %}
{% if loop.index0==0 %}
<tr><#list creation.restriction.fields as field><th>${field.asClassName()}</th></#list></tr>
{% endif %}

<tr><#list creation.restriction.fields as field><td>
<div class="form-group">
    {{ent.form.${field.asVariableName()}.label}} <br/>
    {{ent.form.${field.asVariableName()}(class="form-control")}}
</div>
</td></#list></tr>
 {% endfor %}
</table>
<#else>
<table>
{% for ent in ${creation.name.asVariableName()}_form_list.lista.entries %}
{% if loop.index0==0 %}
<tr>{% for prop in ent.form %}<th>{{prop.label}}</th>{% endfor %}</tr>
{% endif %}

<tr>{% for prop in ent.form %}<td>{{prop()}}</td>{% endfor %}</tr> {% endfor %}
</table>
</#if>
</#if>
```

Figura 4-19: Plantilla de la vista para actividad *"Creation"*

- Sistemas de plantillas: *Jinja 2, Freemarker*
- Sistemas ORM: *SqlAlchemy, OrmLite*

- Procesador de formularios web: *WTforms*
- Tecnologías de front-end: *Bootstrap* y el tema *SB Admin 2 Bootstrap*
- Base de datos: *SQLite3*

5 Casos de estudio

En este capítulo se muestran los resultados de la ejecución de la herramienta de prototipado de sistemas de información transaccionales a partir de especificaciones escritas en lenguaje natural restringido. Con el fin de validar y demostrar la aplicabilidad y efectividad de la propuesta se presentan tres casos de estudio. Primero se presentan dos casos de estudio utilizando la herramienta que fue descrita en la sección 4.1. Estos dos ejemplos se denominan "Question Cycle" y "Email Voting". Por último, se muestra un caso de estudio más completo utilizando la herramienta implementada en la sección 4.2. Este ejemplo lleva el título de "Clon de Odoo Ventas". Es importante aclarar que la discusión de los resultados obtenidos se presentará en el capítulo siguiente.

5.1. Caso de estudio 1: Question Cycle

El prototipo presentado en esa sección cumple con una funcionalidad en el contexto académico. Esta funcionalidad busca que un usuario anónimo genere una prueba a partir de una lista de preguntas filtrada. El proceso especificado en BPMN se presenta en la figura **5-1**.

Inicialmente el usuario selecciona una asignatura y un nivel, posteriormente se muestra el conjunto de preguntas cada una con n posibles respuestas. Las preguntas deben ser filtradas basándose en el nivel y la asignatura seleccionados previamente. Cada pregunta también pertenece a una clasificación extra llamada "DBA" o "Derecho Básico de Aprendizaje". El usuario selecciona las preguntas, y a continuación, se le muestra la primera de las preguntas seleccionadas. Las preguntas pueden contener ecuaciones matemáticas y deberían ser mostradas como si se tratara de LaTeX. Cuando no hay más preguntas que responder, se pregunta al usuario si desea finalizar la prueba. Por último, se deberá mostrar un resumen de la prueba con sus respuestas con el fin de retroalimentar al usuario.

La representación en lenguaje natural restringido de este proceso se presenta en la figura **5-2**.

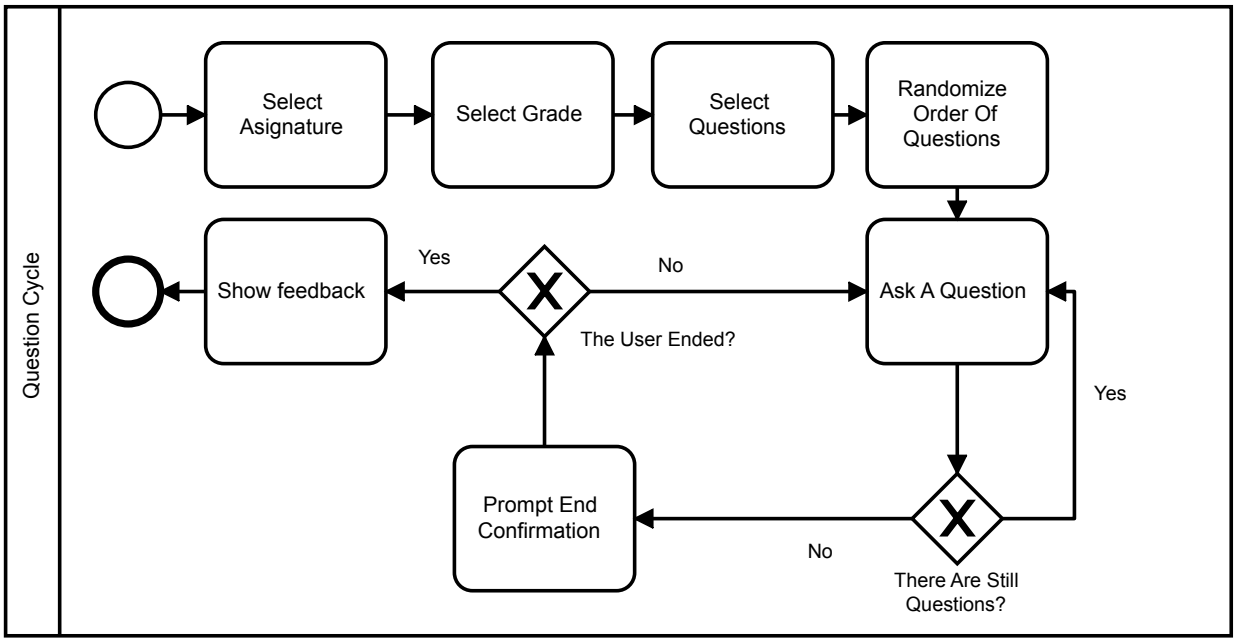


Figura 5-1: Proceso BPMN para "Question Cycle"

Con esta especificación como entrada al proceso de generación de código, el código fuente de la aplicación web fue generado con éxito usando la implementación de la sección 4.1. El resultado final es una aplicación que ejecuta el proceso de esta sección. La figura **5-3** muestra las vistas generadas por la herramienta.

Un par de muestras del código fuente generado se encuentra en las figuras **5-5** y **5-4**. El total de líneas de código generadas en Python fue de 377 y en el total de líneas en HTML fue de $2,239$ incluyendo código extra para interfaces de control.

A pesar de que la mayoría del código fue generado de manera automática, fue necesario el desarrollo de las plantillas para los verbos *ask* (mostrado en la figura **5-5**) y *show*. Esto, por supuesto, afecta la metodología de prototipado y por esta razón se tomó la decisión de proponer el sub-lenguaje T_n para la especificación de tareas.

En este caso, la misma plantilla fue usada para todas las tareas que tuvieran el verbo *select*. Al igual que en los demás casos de estudio y sus prototipos correspondientes, las compuertas exclusivas son vistas que preguntan al usuario cuál es el camino que desea tomar.

No obstante, este primer prototipo tiene limitaciones con respecto a los requisitos originalmente planteados. Esto incluye: los filtros de las preguntas por nivel y asignatura no fueron implementados automáticamente; las compuertas pueden ser reemplazadas por código fuente que determine automáticamente el camino a seguir; la alteración en el orden de

```
a Subject has: a Name, a Code Value and a set of Questions
a Question has: a Dba, a set of Answers, a Level and a Heading Text
an Answer has: a Text Value
a Dba has: a Code Value and a Name Value
a Level has: a Course Number

the Question Cycle process stars, then: Select Subject, Select Level, Select Questions, Randomize Order Of
Questions, Ask A Question and, ask if Are There Unanswered Questions?.

if the answer to Are There Unanswered Questions is Yes then: go back to Ask A Question.

if the answer to Are There Unanswered Questions is No then: Prompt End Confirmation, and ask if The User Ended.

if the answer to The User Ended is Yes then: Show Questions Answer, and the process ends.

if the answer to The User Ended is No then: go back to Ask A Question.
```

Figura 5-2: Especificación en lenguaje natural restringido para *Question Cycle*

las preguntas tampoco fue implementada, ni las ecuaciones se muestran en formato LaTeX. Estas características deberán ser implementadas manualmente por los desarrolladores una vez terminado el ciclo de prototipado.

5.2. Caso de estudio 2: Email Voting

El segundo caso de estudio corresponde a un proceso extraído y adaptado de `bpmn.org`, denominado *Email voting*. El proceso ejecutado en este caso consiste en un proceso de resolución de problemáticas usando una votación por correo electrónico. El proceso es pequeño, pero lo suficientemente complejo para mostrar ejemplos de varias de las características de la herramienta propuesta en esta tesis, y servirá para ayudar a ilustrar que la herramienta puede manejar procesos de negocio simples e inusuales también, y aún así, será fácil de entender para los diseñadores que lean la especificación.

En la figura **5-6** se ilustra el modelo para este caso de estudio usando BPMN. Su respectiva representación en el lenguaje natural restringido se muestra en la figura **5-7**.

El resultado final es el prototipo funcional de una aplicación web que ejecuta el proceso propuesto para este caso de estudio. La figura **5-8** ilustra el resultado de la ejecución de la aplicación prototipada mediante la presentación de sus vistas en un navegador web.

Adicionalmente, vale la pena presentar fragmentos del código fuente generado para esta aplicación. Ésto con el fin de mostrar que el código fuente generado como resultado del

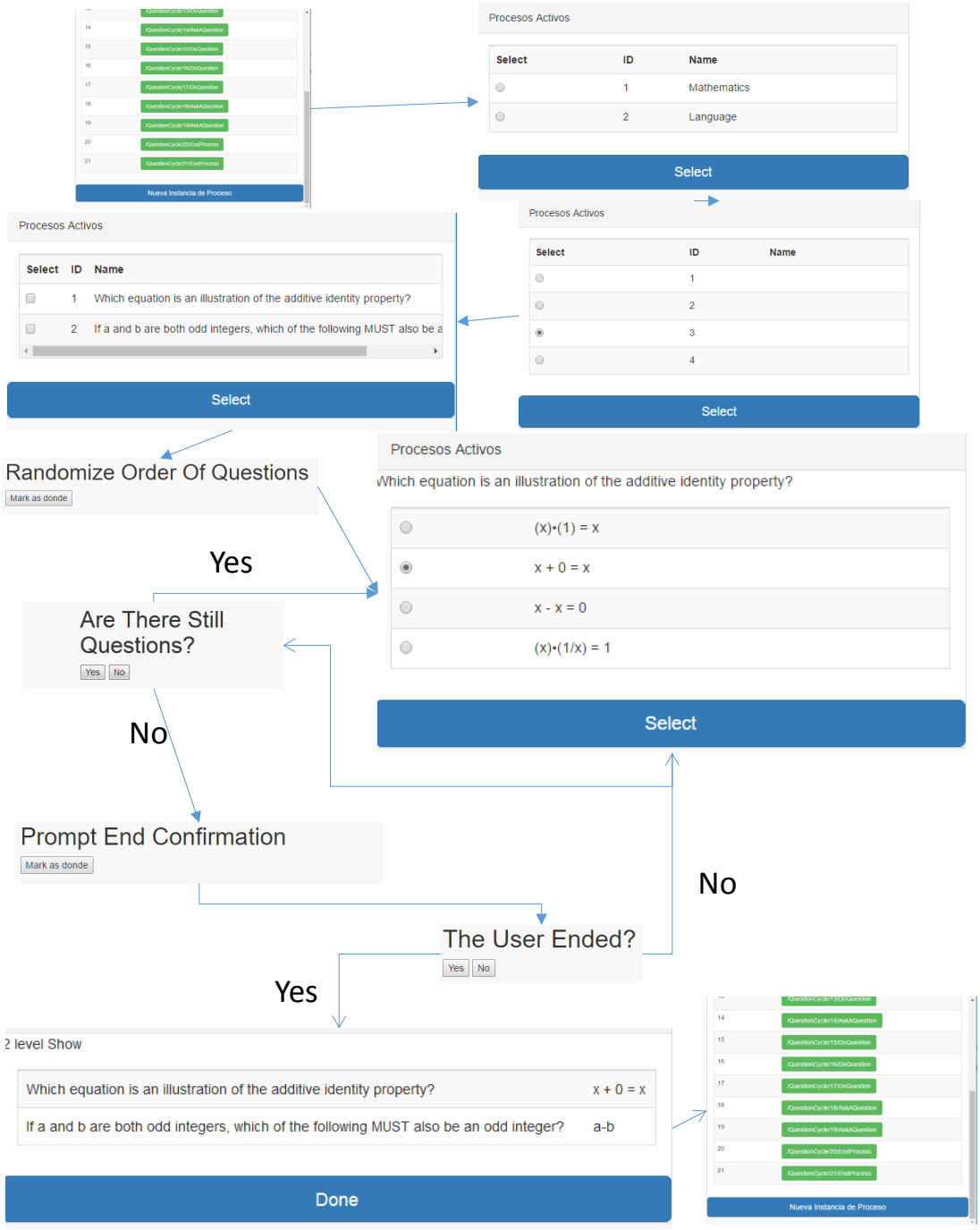


Figura 5-3: Vistas generadas automáticamente para: *Question Cycle*

```python
@app.route('/QuestionCycle/<pid>/AskAQuestion', methods=['GET','POST'])
def AskAQuestion(pid):
    data = get_instance_data(pid)
    Question_index = data.get("current_Question",-1)+1
    if request.method=='POST':
        update_instance_data(pid,"Question__"+request.form.get("instance",0),request.form.get("selected",0))
        update_instance_data(pid,"current_Question",Question_index)
        update_process(pid,str(request.url_rule),'/QuestionCycle/<pid>/ThereAreStillQuestions')
        return redirect('/QuestionCycle/<pid>/ThereAreStillQuestions'.replace('<pid>',pid))

    Question_list = data.get("selected_Question")
    current_Question = Question_list[Question_index]
    Question_instance = session.query(Question).filter(Question.pk==current_Question).first()
    template = env.get_template('AskQuestion.html')
    return template.render(instance=Question_instance)
```

Figura 5-4: Controlador para la tarea "Ask Question" de *Question Cycle*

```
<div>{{instance.heading}}</div>
<form method="post">
<input type=hidden name="instance" value="{{instance.pk}}"/>
<!-- /.panel-heading -->
<div class="panel-body">
<!-- Nav tabs -->
<div class="table-responsive">
<table class="table table-striped">
{%for answ in instance.answers%}
<tr><td><input type=radio name="selected" value="{{answ.pk}}"/></td><td>{{answ.text}}</td></tr>
{%endfor%}
</table>
<!-- /.panel-body -->
</div>
</div>
<input type=submit class="btn btn-primary btn-lg btn-block" value="Select">
<form>
```

Figura 5-5: Vista para la tarea "Ask A Question" de *Question Cycle*

proceso es legible con facilidad por programadores en caso que sea necesario realizar alguna modificación de forma manual. Un ejemplo del controlador para la tarea "Receive Issue List" se presenta en la figura **5-9**. Asimismo, la plantilla de vista para el verbo "Recieve" se muestra en la figura **5-10**. El resultado final del prototipo generado consta de 546 líneas de código en el lenguaje de programación Python y 1,650 líneas en HTML.

Al igual que en el primer prototipo, en el segundo caso de estudio, también se pueden notar varias limitaciones y trabajo extra realizado: varios verbos tuvieron que ser implementados y las funcionalidades de *send email* y *post on website* no se implementaron automáticamente. Por último, cabe resaltar que algunos de los nombres de las tareas pueden ser alterados para

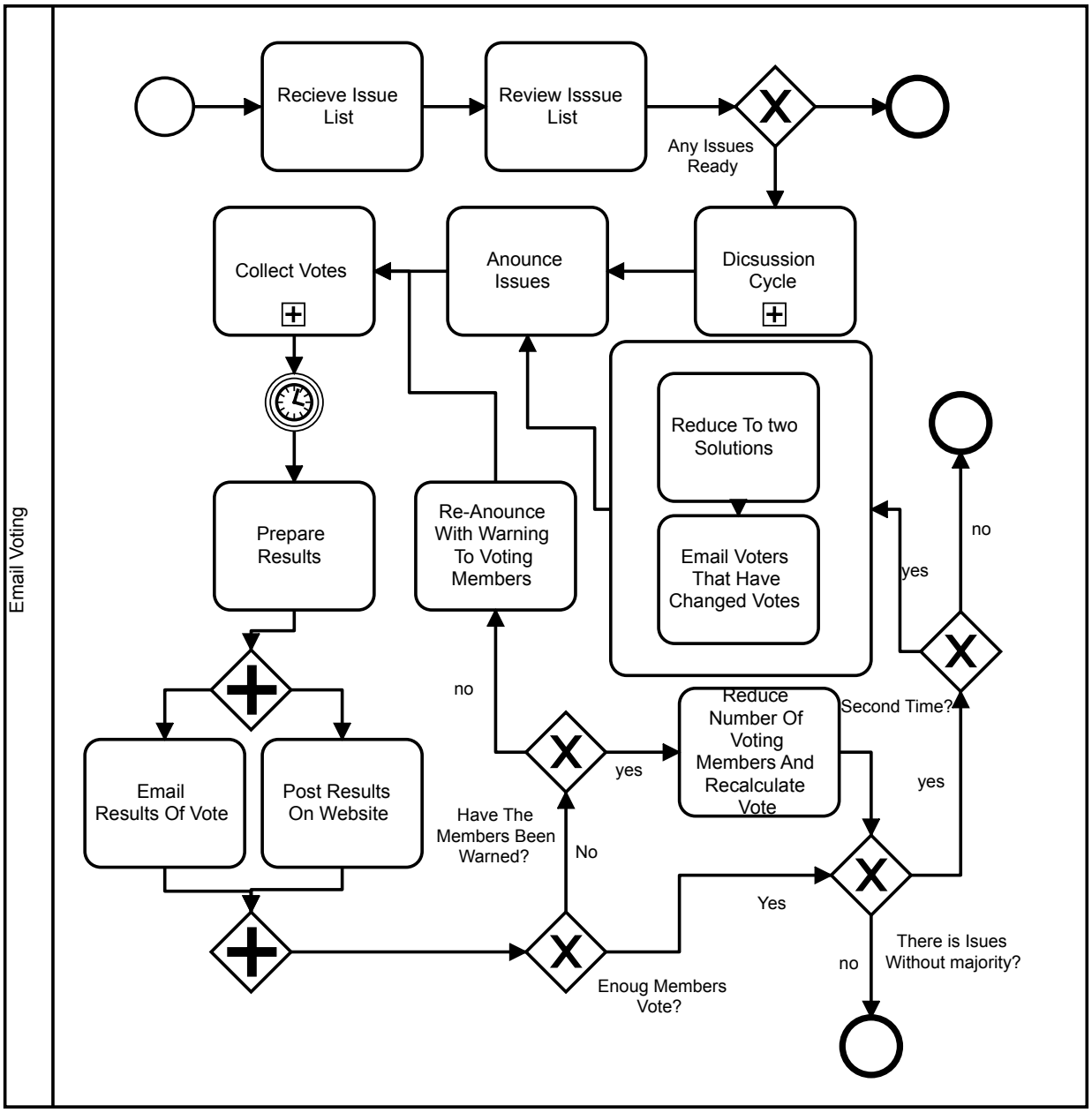


Figura 5-6: Proceso BPMN para *Email Voting*

acercarse más a alguna plantilla definida con anterioridad. Esto se consideró ineficiente y fue reemplazado en la segunda implementación.

```
an User has: a Name, a Password and an Id Number
an Issue has: a Description Text, a Creation Date, an Approved Mark and a Possible Solution
a Solution has: a Description Text
a Vote has: an User, a Description Text and a Voting Date
a Result has: a Name Value, a set of Votes and a set of Issues

the Email Voting process stars, then: Receive Issue List, Review Issue List and ask if There Are Issues.

if the answer to There Are Issues is Yes then: the Discussion Cycle is made, Announce Issue,
Collect Vote, wait 1 week, Prepare Results, do at the same time the Information tasks and ask if Enough
Members Vote.

if the answer to There Are Issues is No then: the process ends.

the Discussion Cycle subprocess stars, then : Announce Issues For Discussion, Moderate Email Discussion and
the subprocess ends.

for Information tasks do: Post Result On Web Site.

for Information tasks also do: Email Result Of Vote.

if the answer to Enough Members Vote is Yes then: ask if There Is Issues Without Majority.

if the answer to There Is Issues Without Majority is Yes then: ask if There Will Be A Second Time.

if the answer to There Will Be A Second Time is Yes then: go back to Discussion Cycle.

if the answer to There Will Be A Second Time is No then:    Reduce To Two Solutions, Email Voters That Have Changed
Votes and go back to Announce Issues.

if the answer to There Is Issues Without Majority is No then: the process ends.

if the answer to Enough Members Vote is No then: ask if Members Have Been Warned.

if the answer to Members Have Been Warned is Yes then: Reduce Number Of Voting Members, Recalculate Vote
and go to ask if There Is Issues Without Majority.

if the answer to Members Have Been Warned is No then: Re-Announce Vote With Warning To Voting Members and
go back to Collect Votes.
```

Figura 5-7: Especificación en lenguaje natural restringido para *Email Voting*

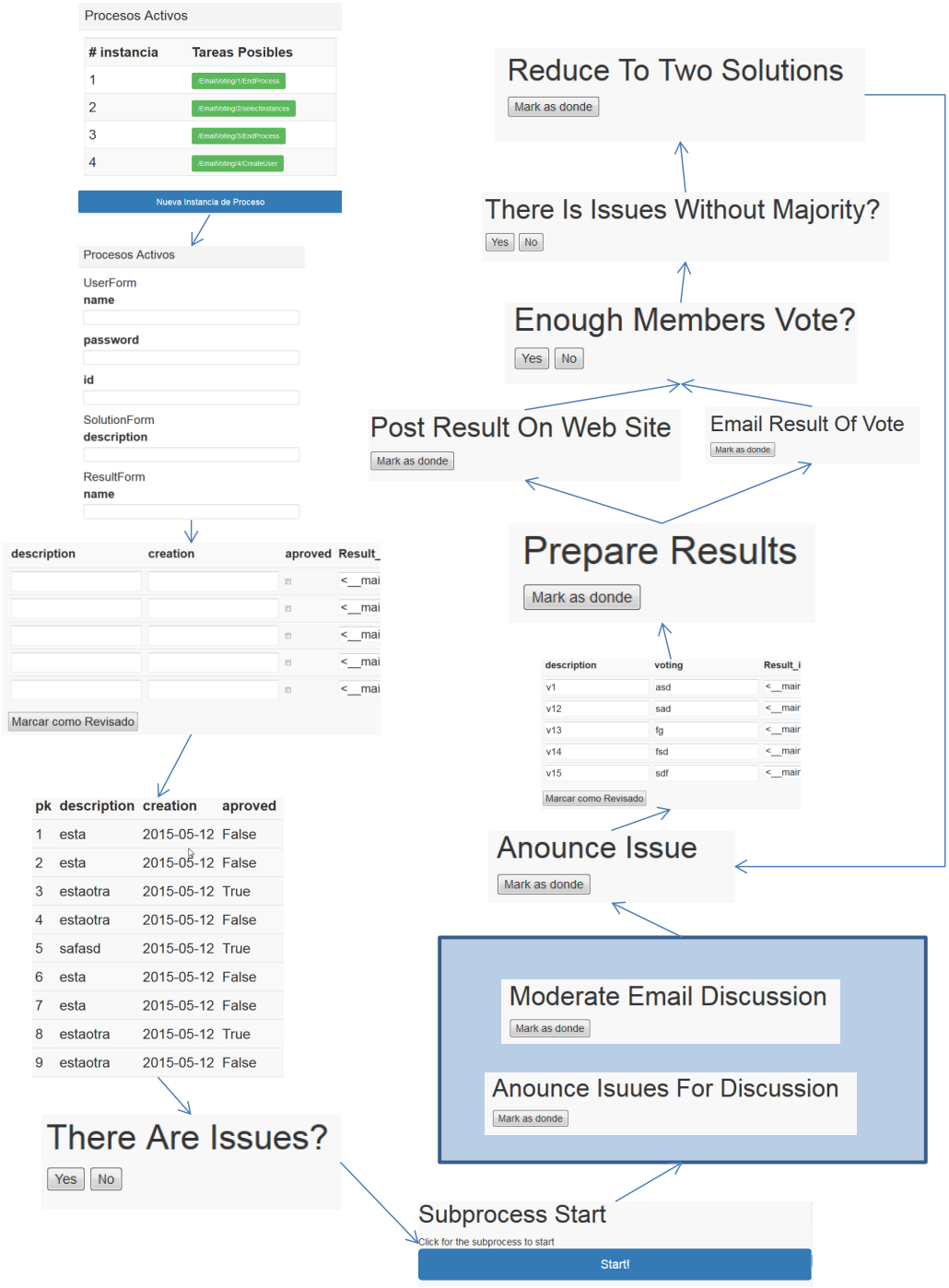


Figura 5-8: Vistas generadas automáticamente para *Email Voting*

```
@app.route('/EmailVoting/<pid>/RecieveIssueList', methods=['GET','POST'])
def RecieveIssueList(pid):
    form = Issue_listForm(request.form)
    if request.method=='POST':
        for ent in form.lista.entries:
            obj= Issue()
            ent.form.populate_obj(obj)
            session.add(obj)
        session.commit()
        update_process(pid,str(request.url_rule),'/EmailVoting/<pid>/ReviewIssueList')
        return redirect('/EmailVoting/<pid>/ReviewIssueList'.replace('<pid>',pid))
    template = env.get_template('RecieveList.html')

    return template.render(form=form)
```

Figura 5-9: Controlador para la tarea "Receive Issue List" de *Email Voting*

```
<body>
    <div id="wrapper">
        <form role="form" action="" method="post" name="lform" targe="_blank">
            <div class="col-lg-3 col-md-4">
            <fieldset>
                 <div class="table-responsive">
                    <table class="table table-striped">
                         {% for ent in form.lista %}
                            {% if loop.index0==0 %}
                                <tr>
                                {% for prop in ent.form %}
                                    <th>{{prop.label}}</th>
                                {% endfor %}
                                </tr>
                            {% endif %}
                            <tr>
                            {% for prop in ent.form %}
                                <td>{{prop()}}</td>
                            {% endfor %}
                            </tr>
                         {% endfor %}
                    </table>
            </fieldset>
                <input type="submit" value="Marcar como Revisado" name="Revisado"/>
            </div>
        </form>
    </div>
</body>
```

Figura 5-10: Vista para la tarea "Receive Issue List" de *Email Voting*

5.3. Caso de estudio 3: Clon de Odoo

Este caso de estudio expone los resultados obtenidos mediante la ejecución de la segunda implementación de la estrategia de prototipado propuesta en la sección 4.2. El sistema seleccionado como tercer caso de estudio se denomina Odoo[1], el cual es un sistema ERP (Enterprise Resource Planning) y CRM (Client Relationship Management) abierto para el uso público. Para los propósitos de esta tesis, algunos de los módulos presentes en Odoo fueron replicados, escribiendo una especificación en lenguaje natural restringido y usando el generador de código fuente implementado. Los módulos seleccionados fueron: ventas y proyectos. La versión de Odoo que se utilizó como objetivo para la especificación es la versión 9.0.

Por un lado, el módulo de ventas consta de cinco procesos. Cada proceso consiste en un par de tareas. Hay tres procesos que se encargan de crear elementos en el sistema: creación de clientes, creación de oportunidades y creación de productos. También, hay dos procesos adicionales que presentan un listado de oportunidades y un listado de productos. Los elementos de las listas pueden ser seleccionados y posteriormente editados.

Por otra parte, el módulo de proyectos permite crear una lista de proyectos, cada uno de estos con una serie de tareas asociadas. También existen procesos para la selección, presentación y creación de tareas y proyectos.

La lista de requisitos resultante puede ser especificada usado el lenguaje natural restringido propuesto en la sección 3.1, como se detalla a continuación:

Language is an Domain Class and needs: a String named Name.

Contact Type is an Domain Class and needs: a String named Type.

Contact is an Domain Class and needs: a Contact Type named Type, a String named Name, a String named Address, a String named Address2, a String named City, a String named State, a Number named Zip Code, a String named Country, a Website named Homepage, a String named Work Desk, a Phone named Personal Phone, a Phone named Mobile Phone, a Phone named Fax, a Email named Personal Email, a String named Title and a Language named Language.

User is an Domain Class and needs: a String named Name and an Email named Username.

Sales Team is an Domain Class and needs: a String named Name, a Number named Code and a User named Team Leader.

Activity is an Domain Class and needs: a String named Message Type, a Number named Number Of Days and a Sales Team named Assigned Sales Team.

Opportunity State is an Domain Class and needs: a String named Name.

[1] https://www.odoo.com/

Opportunity Phase is an Domain Class and needs: a String named Name.

Opportunity is an Domain Class and needs: a String named Description, a Number named Income, a Number named Probability, a Contact named Client, an Activity named Next Activity, an Email named Contact Email, a Phone named Contact Phone, a Date named Provided Closing, a User named Seller, a Sales Team named Responsible Sales Team, a Number named Rating, an Opportunity Phase named Phase and a Opportunity State named State.

Product Type is an Domain Class and needs: a String named Name.

Product is an Domain Class and needs: an Image named Photo, a String named Name, a Product Type named Type, a String named Bar Code, a Number named Price and a Number named Cost.

Project State is a Domain Class and needs: a String named Name.

Project is a Domain Class and needs: a String named Name, a String named Task Label, a Contact named Client and a Project State named State.

Task is a Domain Class and needs: a String named Name, a String named Description, a Date named Limit Date, a User named Assigned and a Project named Project.

the Client Creation process starts, then: Create Contact and, the process ends.

Create Contact is a task where: -a Contact is created.

the Opportunity Creation process starts, then: Create Opportunity and, the process ends.

Create Opportunity is a task where: -a Opportunity are created.

the Opportunity List process starts, then: Show Opportunities, Show Opportunity and, the process ends.

Show Opportunities is a task where: -all the Opportunity are shown, with single selection capabilities.

Show Opportunity is a task where: -a Opportunity is shown, with edition capabilities.

the Product List process starts, then: Show Product List, Product Detail and, the process ends.

Show Product List is a task where: -all the Products are shown, only Name, Price are shown, with single selection capabilities.

Product Detail is a task where: -a Product is shown.

the Product Creation process starts, then: Create Product and, the process ends.

Create Product is a task where: -a Product is created.

the Project Creation process starts, then: Create Project and, the process ends.

Create Project is a task where: -a Project is created.

the Project List process starts, then: List Projects, Show Selected Project and, the process ends.

List Projects is a task where: -all the Projects are shown, with single selection capabilities.

Show Selected Project is a task where: -a Project is shown. -all the Tasks in Project are shown.

the Task Creation process starts, then: Create Task and, the process ends.

Create Task is a task where: -a Task is created.

the Task List process starts, then: List Tasks, Show Task and, the process ends.

List Tasks is a task where: -all the Tasks are shown, with single selection capabilities.

Show Task is a task where: -a Task is shown, with edition capabilities.

A partir de esta especificación, se obtiene de forma automática un prototipo funcional que emula el comportamiento de los módulos de ventas y proyectos de Odoo. Así, se obtiene una serie de vistas enlazadas como se presenta en la figura **5-11**. Estas vistas contienen los procesos de creación, visualización, edición y selección que se pretenden emular del sistema original.

Como ejemplo del código generado por la implementación de la sección 4.2, en la figura **5-12** se muestra el controlador para la tarea "Show Selected Project". Este controlador procesa la vista que se muestra en la figura **5-13**, que también es una plantilla que es compatible con el motor de plantillas Jinja2. En esta tarea cada uno de los campos de la clase de dominio "Proyecto" es mostrado al igual que todas las "Tareas" asociadas al proyecto seleccionado. Para este caso de estudio se generaron 1004 lineas de código fuente en Python y 1671 en HTML.

Por último, es importante mencionar algunas de las limitaciones encontradas en este caso de estudio. Esto incluye: la forma presentación original de la información de Odoo consiste en un sistema de cartas, el cual se perdió en el prototipado, y se presenta la información de forma básica de acuerdo a las plantillas y estilos seleccionados para el prototipo; también se perdieron las listas basadas en columnas que representan estados, por ejemplo, de una tarea; las imágenes de los productos no se muestran, aunque si son subidas; las etiquetas de los proyectos y los campos de auto completado fueron omitidos; y por último, las etiquetas de los campos de los formularios no son configurables sin editar el código fuente.

Figura 5-11: Vistas resultantes en el clon de Odoo

```
@app.route('/projectList/<pid>/showSelectedProject', methods=['GET','POST'])
def projectList_showSelectedProject(pid):
    viewparams={}
    data = get_instance_data(pid)
    formdata = request.form.copy()
    for file in request.files:
        formdata[file] = request.files[file].filename
    request.form = formdata
    project = session.query(Project).filter(Project.id==data.get("selected_Project",0)).first()
    viewparams['project']=project
    task_list = session.query(Task).filter(Task.project_id==data.get('selected_Project',0)).all()
    viewparams['task_list']=task_list
    template = env.get_template('ShowSelectedProject_view.html',globals=labels)
    return template.render(viewparams)
```

Figura 5-12: Ejemplo de controlador generado

```
<div class="panel-heading">Show Selected Project</div>
<div class="panel-body">
{% for field in project.__table__.columns._data.keys()%}
<div class="form-group"><b>{{field}}</b><br /> {{project[field]}}</div>
{% endfor %}
<div class="col-lg-12">
<div class="table-responsive">
<table class="table">
<tr>
{% for prop in task_list[0].__mapper__.columns.keys() %}
<th>{{prop}}</th>
{% endfor %}
</tr>
{% for ent in task_list %}
<tr>
{% for prop in ent.__mapper__.columns.keys() %}
<td>{{ent[prop]}}</td> {% endfor %}
</tr>
{% endfor %}
</table></div></div></div>
```

Figura 5-13: Ejemplo de vista generada

6 Discusión

En este capítulo se discuten los resultados obtenidos en la sección anterior. Esto con el fin de evidenciar ventajas y limitaciones encontradas en la estrategia de prototipado propuesta en esta tesis. Además, se presenta un comparativo del enfoque propuesto con respecto a los trabajos relacionados.

6.1. Discusión de resultados

Esta sección se dividirá en dos partes, una en la que se discutirán los resultados de los casos de estudio como un conjunto y otra en la que se hablara acerca de las herramientas creadas para validar la metodología.

En primer lugar, sobre los resultados de los casos de estudio, se puede ver que se generó satisfactoriamente código fuente ejecutable, que además, es correcto desde el punto de vista de la funcionalidad deseada. La representación textual permite describir el prototipo y editarlo rápidamente para continuar con la validación de los requisitos. Además, el código fuente generado como resultado del proceso de prototipado es legible con facilidad por programadores. Esta es una ventaja importante en caso que sea necesario realizar alguna modificación de forma manual para completar la implementación de los requisitos del sistema final.

En conjunto los resultados presentados en los casos de estudio cumplen con el objetivo de bosquejar el funcionamiento de las herramientas finales, que es lo que se espera de un sistema de prototipado. Por ejemplo, en el tercer caso de estudio, se logró replicar gran parte de la funcionalidad de dos módulos de un ERP comercial, sin hacer uso de nada ajeno a la especificación escrita en lenguaje natural restringido y la herramienta de generación de código.

Por otra parte, las aplicaciones web fueron generadas para ser de fácil despliegue. Tan solo es necesario realizar algunos cambios para desplegar la aplicación en un servidor que use Apache o Nginx. Esto incluye; cambios en los datos de conexión a la base de datos; e instalar

las librerías de Python de las cuales depende el proyecto.

Un elemento que es transversal a todos los casos de estudio son las vistas en las que se visualiza el estado de las instancias de los procesos. En estas vistas se muestran las tareas disponibles para cada instancia de los procesos. En muchos casos esta vista debe ser enseñada al usuario para que reanude las actividades de su quehacer diario. Una ventaja de esta vista, y del método de control de flujo de actividades en general, es que la implementación es igual para todas las especificaciones que se realicen con este sistema de prototipado. Esta misma característica se muestra en la implementación transversal de una plantilla para servicios web.

No obstante, es importante mencionar algunas limitaciones del sistema de prototipado propuesto en esta tesis. El límite de esta propuesta está en la ejecución de los procesos y no se toman en cuenta cosas como el cálculo de estadísticas sobre estos procesos, ni la visualización de éstas. Tampoco se permite en este esquema de prototipado una vista por fuera de un proceso, es decir, todas las vistas generadas deben pertenecer a algún proceso y los datos de la ejecución de éste se almacenaran. También, existen limitaciones a la hora de construir las clases de dominio, por ejemplo, para este sistema es imposible restringir la cantidad de elementos presentes a algún lado de la relación. Tampoco le es posible hacer validaciones complejas sobre las instancias de estas clases de domino.

En segundo lugar, con respecto a las implementaciones realizadas de la herramienta de generación de código, se pueden notar que las dos implementaciones cumplen con mostrar la aplicabilidad de la propuesta. Generan los prototipos de manera rápida y precisa; el código fuente generado cumple con las características deseadas de legibilidad y funcionalidad; y en general, cumplen con las características esperadas de la metodológica propuesta.

En particular, en la primera implementación se evidenciaron carencias que posteriormente fueron corregidas. En ésta se muestra que el uso de una plataforma extra para editar las plantillas de código fuente es una herramienta útil. El uso de una plantilla por verbo demostró ser una alternativa útil pero difícil de mantener, y que podría ser útil un sistema para extender los tipos de especificaciones. Por ejemplo, para los posibles campos de una base de datos.

Esta primera implementación fue útil para corregir varias cosas que se habían pasado por alto. Primero, durante el desarrollo de esta implementación se notó un problema: determinar la instancia de la clase de dominio sobre la cual una tarea de BPMN debía operar. Esto se solucionó en esta primera implementación detectando que instancias eran necesarias y generando el código para crearlas todas en la primera tarea. En la siguiente implementación, se solucionó proponiendo un tipo de tarea que incluyera la posibilidad de seleccionar.

Uno de los hallazgos más significativos de esta primera implementación fue que la aproximación de plantilla por verbo no era suficiente ni realizable, puesto que entorpece el proceso de prototipado y puede requerir una gran cantidad de plantillas predefinidas para alcanzar los resultados de manera sencilla. Esa dificultad dio origen a la parte de la gramática T_n para determinar el contenido de la tarea.

Además, se determinó que para efectos de extender la funcionalidad de la herramienta, también era posible tener plantillas asociadas a un tipo de campo y convertir el "mapeo" de la primera implementación en una consulta a la base de datos de plantillas. Esto permite extender las funcionalidades de la herramienta propuesta.

Para la segunda implementación se usó Xtext, lo que permitió integrar la solución con el IDE Eclipse. Esta integración hizo más amigable la creación de las especificaciones en lenguaje natural restringido, puesto que las especificaciones pueden aprovechar las características de auto-completado y sintaxis resaltada. El uso de un servicio web para editar el código de las plantillas no fue implementado aquí, lo cual dificultó el mantenimiento de las plantillas. El uso de otro sistema de plantillas fue necesario debido al cambio del entorno de desarrollo, pero resultó ser ventajoso puesto que la sintaxis de la plantilla del proceso no interfería con la sintaxis de la plantilla del resultado, haciendo más sencillo su desarrollo. Esta herramienta fue desarrollada usando el lenguaje Xtend y resultó ser un código fuente difícil de mantener.

Por último, estas implementaciones también evidencian varias limitaciones de la metodología de prototipado en sí. Por ejemplo, extender el tipo de visualizaciones de tareas no es posible. Tampoco es posible definir la visualización desde la especificación. Durante el desarrollo de última implementación, surgió una dificultad a la hora de buscar una plataforma configurable como resultado, ya que se requeriría procesar de múltiples maneras un mismo símbolo no-terminal de la gramática.

6.2. Comparativa con trabajos relacionados

La tabla **6-1** presenta una comparativa con trabajos con un objetivo similar al de esta tesis. Esta tabla incluye las diferencias principales con respecto al punto de partida del proceso y los resultados finales obtenidos en cada uno de los trabajos. En cuanto al punto de partida del proceso, éste puede ser: plantillas de requisitos, lenguaje natural sin restricciones, lenguaje de especificación o lenguaje natural restringido. Por otra parte, en relación al resultado final de cada trabajo, éste incluye diversos objetivos como son: programa ejecutable, modelos UML, modelos E-R, modelo BPMN, código en el lenguaje de dominio específico (DSL), etc.

Tabla 6-1: Comparación con trabajos relacionados

Trabajo	**Punto de partida**	**Resultado final**
[Schwitter, 1996]	Lenguaje de especificación	Ejecutable
[Deeptimahanti and Sanyal, 2011]	Lenguaje natural sin restricciones	UML
[Friedrich et al., 2011]	Lenguaje natural sin restricciones	BPMN
[Liu et al., 2004]	Plantillas de requisitos	UML (clases y secuencias)
[Abbott, 1983]	Lenguaje natural sin restricciones	Ejecutable ADA (proceso manual)
[Geetha and Mala, 2013] [Geetha and Anandha Mala, 2014]	Lenguaje natural sin restricciones	E-R
[Popescu et al., 2008]	Plantillas de requisitos	UML (clases)
[Chioaşcă, 2012]	Lenguaje natural sin restricciones	OSMs (*Object System Models*)
[Overmyer et al., 2001]	Lenguaje natural sin restricciones	UML (clases)
[Desai et al., 2016]	Lenguaje natural sin restricciones	DSLs
Este trabajo [Alfonso and Restrepo-Calle, 2017]	Lenguaje restringido + plantillas de requisitos	Prototipo ejecutable de aplicación web

En la mayoría de los casos, el método de entrada al proceso que se escoge es el lenguaje natural sin restringir. Esto se debe a la expresividad y libertad que éste permite, a cambio de una creciente complejidad en el procesamiento automático. La siguiente aproximación más usada es la de las plantillas puesto que permite un fácil procesamiento mientras se mantiene la claridad de la especificación. Esto a cambio de tener que memorizar las plantillas o tener que asistir el proceso de especificación. Por último, la aproximación del lenguaje de especificación construido con elementos de un lenguaje natural es la menos popular puesto que requiere aprender un lenguaje casi artificial y muchas veces el procesamiento que se le debe realizar es mixto. En el enfoque propuesto en esta tesis, se hace uso de un lenguaje restringido a unos pocos constructos, lo que facilita su aprendizaje, y permite implementar herramientas de asistencia al proceso de especificación y procesamiento automático. Además, estos constructos son muy similares a una plantilla de requisitos como se mostró en la tabla **2-9** y preservan la ventaja de mantener la especificación clara y sin ambiguedades.

Como se puede apreciar en la tabla anterior y en las tablas **2-8**, **2-9**, **2-10** y **2-11** de la sección 2.6 (trabajos relacionados), el objetivo de la mayoría de trabajos es producir uno o varios modelos en algún lenguaje de especificación como BPMN, UML (algunos de los posibles diagramas), o E-R. Las traducciones posteriores a código fuente están limitadas por las herramientas externas como Rational Rose[1] o Visual Paradigm[2]. En algunos casos, también se necesita una intervención manual del usuario para seleccionar los atributos de diseño. Solo una pocos trabajos obtienen programas ejecutables como salida. Entre ellos, el trabajo [Abbott, 1983] que propone el proceso manual original y la propuesta de [Schwitter, 1996] que requiere especificaciones y código pre-definido para obtener un ejecutable de línea de

[1] http://www-03.ibm.com/software/products/es/enterprise
[2] https://www.visual-paradigm.com/

comandos. En esta tesis se obtiene un prototipo funcional más cercano al producto final sin intervención humana posterior o el uso de herramientas externas. Además, el prototipo funcional que se obtiene en esta propuesta utiliza tecnologías web.

Aunque dentro del alcance de esta tesis no estaba incluido realizar una evaluación detallada de esta propuesta en comparación con las herramientas de prototipado rápido como Bizzagi, JustInMind y OpenXAVA; es importante tener en cuenta las siguientes consideraciones al respecto. En comparación con Bizzagi, la propuesta de esta tesis es más flexible a la hora de construir el prototipo. Sin embargo, Bizzagi es una herramienta más robusta y segura a la hora de desplegar un ejecutable. Si comparamos la metodología propuesta en esta tesis con JustInMind, es igualmente más rápida y contiene casos de uso que en JustInMind no están presentes (como los procesos en general) ya que éste está orientado principalmente al diseño de las vistas para el prototipado de sitios web y aplicaciones para dispositivos móviles. No obstante, JustInMind tiene mayor libertad a la hora de editar el contenido y el estilo de estas páginas. Por último, si se compara con OpenXAVA, esta herramienta permite avanzar en la misma dirección (sistemas de información), pero el paso inicial del proceso se encuentra en la definición de únicamente las clases de domino, dejando el resto de prototipado en manos de un desarrollador.

7 Conclusiones

7.1. Conclusiones generales

En esta tesis se presentó una metodología para prototipado rápido de sistemas de información transaccionales usando una especificación escrita en lenguaje natural restringido. También se presentaron las partes que componen esta metodología, como la gramática con la que se deben escribir las especificaciones del sistema y el método de generación de código fuente a partir de esta especificación. Además, se realizó la implementación del enfoque propuesto de dos maneras diferentes. Por último, se evaluó la aplicabilidad y efectividad de la propuesta por medio de tres casos de estudio, en los cuales se generaron tres prototipos funcionales de aplicaciones web.

De esta manera, la metodología de prototipado presentada logra cumplir con el objetivo principal de esta tesis "Diseñar, desarrollar y evaluar una metodología para el prototipado automático de sistemas de información transaccionales desde una especificación en lenguaje natural restringido". La metodología se propuso en el capítulo 3 de esta tesis, se implementó en el capítulo 4, se evaluó en el capítulo 5 por medio de tres casos de estudio, y posteriormente, se discutieron los resultados en el capítulo 6. Todo ello, fundamentado y soportado por el marco conceptual y los trabajos relacionados revisados en el capítulo 2.

El uso de esta metodología acompañada de sus herramientas facilita el proceso de prototipado y reduce los riesgos asociados a la elicitación de requisitos y el diseño de un sistema de información. Esto se logra ya que al facilitar el prototipado rápido se pueden obtener validaciones por parte de los *stakeholders* de forma frecuente y ágil. En caso de algún fallo o malentendido en la especificación, se puede corregir rápidamente, reduciendo los costos asociados a los cambios ya que se pueden detectar problemas en etapas tempranas del desarrollo de software.

Para diseñar el lenguaje de especificación denominado lenguaje natural restringido se propusieron dos transformaciones diferentes entre lenguajes de especificación basados en modelos (diagramas de procesos de negocio BPMN y modelos E-R) y un lenguaje natural (en este

caso inglés). Estas transformaciones mantienen la expresividad de los lenguajes originales en la nueva representación. Además, el uso de este nuevo lenguaje permite sobrepasar las capacidades de generación de código de los lenguajes originales ya que en este caso se propuso un tercer fragmento de gramática que complementa el lenguaje de especificación con la expresividad necesaria para especificar tareas. Como beneficio extra, esta especificación podría ser entendida por un *stakeholder* sin conocimientos técnicos.

El uso de plantillas de código fuente para la generación del prototipo resultó ser una aproximación muy útil a la hora de facilitar la mantenibilidad de la herramienta. Esto también facilita el extender, por ejemplo, los tipos de campos disponibles. No obstante, las plantillas pueden llegar a ser complejas y difíciles de leer y entender, sobre todo, en los casos en que hay muchas opciones posibles involucradas. Por ejemplo, para el símbolo no-terminal "Creation", el cual posee múltiples combinaciones entre creaciones en conjunto y singulares, mostrando todos los campos, o solo unos cuantos de estos.

Limitaciones como las que se explicaron en el capítulo anterior evidencian una relación de compromisos entre la complejidad de la especificación y la proximidad que puede llegar a tener un prototipo con respecto al producto final. Este balance afecta directamente la dificultad de generar código fuente. Una especificación más compleja tendrá, por supuesto, mayor cantidad de información de diseño para la cual generar diferentes tipos de código fuente.

Algunas de las limitaciones de este trabajo pueden ser solucionadas de manera similar a como se abordó la generación de código para cada tarea, es decir, proponiendo T_{n2}, un fragmento nuevo de gramática que permita realizar la generación de código para una limitación de estas en particular. No obstante, esto elevaría la complejidad de los procesos de escritura de la especificación y generación de código.

Por otra parte, el procesamiento rápido de la especificación y la generación del prototipo permiten ver cambios en segundos. El hecho de que la especificación sea textual permite realizar cambios rápidamente con su respectiva validación inmediata. Por lo tanto, los cambios a estas especificaciones son económicos y rápidos. Un obstáculo posible podría ser aprender la sintaxis del lenguaje natural restringido, lo cual queda sopesado por el hecho de poder integrar el lenguaje propuesto a un IDE como Eclipse, y de esta forma, poder contar con funcionalidades como resaltado de sintaxis y auto-completado.

7.2. Aportaciones

A continuación se enumeran las principales aportaciones de esta tesis de Maestría en Ingeniería de Sistemas y Computación:

1. Una revisión actualizada de los trabajos relacionados con la generación de código fuente o modelos de software.
2. Una metodología para el prototipado automático de sistemas de información transaccionales.
3. Un lenguaje de especificación basado en texto y en lenguaje natural restringido.
4. Un método de generación de código fuente a partir de esta especificación.
5. Dos herramientas computacionales que implementan estas ideas y demuestran su aplicabilidad.

7.3. Publicaciones

Como resultado del desarrollo de esta Tesis de Maestría en Ingeniería de Sistemas y Computación, se publicó un artículo en una conferencia internacional en el área de ingeniería de software [Alfonso and Restrepo-Calle, 2017]. El artículo publicado es:

- Alfonso, Jean Pierre, & Restrepo-Calle, Felipe (2017). Automatic Source Code Generation for Web-based Process-oriented Information Systems. In Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2017, (pp. 103–113). Porto - Portugal, 28-29 April, 2017.

Posteriormente los organizadores de este evento se comunicaron el autor de esta tesis para solicitar una versión extendida del artículo anteriormente mencionado. Esta versión será publicada como un capítulo de libro bajo la siguiente referencia:

- Alfonso, Jean Pierre & Restrepo-Calle, Felipe (2017). "Fast prototyping of web-based information systems using a restricted natural language specification". In Communications in Computer and Information Science. Springer. (under review)

7.4. Trabajo Futuro

Esta tesis abre caminos interesantes para el prototipado rápido de aplicaciones web. Para continuar con el trabajo presentado se proponen diferentes acciones futuras a tener en cuenta, entre las cuales se encuentran:

- Proponer una fragmento de gramática o método para resolver automáticamente las compuertas exclusivas.
- Extender el lenguaje natural restringido agregando representaciones textuales de otros modelos, además de E-R y BPMN.
- Alterar la ejecución de las tareas basándose en las tareas previas.
- Generar estadísticas de comportamiento de los procesos.
- Generar reportes con visualizaciones de estas estadísticas.
- Generar visualizaciones especiales de campos.
- Colocar restricciones en las relaciones entre clases de dominio.

También, podría realizarse un trabajo que busque romper ese intercambio de compromisos entre complejidad de la especificación y detalle del prototipo funcional. Este trabajo se podría concebir no solo como una metodología de prototipado, sino también como una metodología de construcción de software veloz y a bajo riesgo.

Bibliografía

[Abbott, 1983] Abbott, R. J. (1983). Program design by informal English descriptions. *Communications of the ACM*, 26(11):882–894.

[Aho et al., 2007] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2007). *Compilers: Principles, Techniques and Tools*.

[Aiello et al., 2014] Aiello, G., Di Bernardo, R., Maggio, M., Di Bona, D., and Re, G. L. (2014). Inferring business rules from natural language expressions. *Proceedings - IEEE 7th International Conference on Service-Oriented Computing and Applications, SOCA 2014*, pages 131–136.

[Alfonso and Restrepo-Calle, 2017] Alfonso, J. P. and Restrepo-Calle, F. (2017). Automatic Source Code Generation for Web-based Process-oriented Information Systems. In Damiani, E., Spanoudakis, G., and Maciaszek, L., editors, *12th International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 103–113, Porto - Portugal.

[Ambriola and Gervasi, 1997] Ambriola, V. and Gervasi, V. (1997). Processing natural language requirements. *Proceedings of the IEEE International Automated Software Engineering Conference, ASE*, pages 36–45.

[Anandha and Uma, 2006] Anandha, G. S. and Uma, G. (2006). Automatic construction of object oriented design models [UML Diagrams] from natural language specification. *Trends in Artificial Intelligence*, 4099:1155–1159.

[Arora et al., 2015] Arora, C., Sabetzadeh, M., Briand, L., and Zimmer, F. (2015). Automated checking of conformance to requirements templates using natural language processing. *IEEE Transactions on Software Engineering*, 41(10):944–968.

[Augustine and Martin, 2005] Augustine, S. and Martin, R. C. (2005). *Managing agile projects*. Robert C. Martin series. Prentice Hall Professional Technical Reference, Upper Saddle River, NJ.

[Bajwa and Choudhary, 2012] Bajwa, I. S. and Choudhary, M. A. (2012). From Natural Lan-

guage Software Specifications to UML Class Models. In *Enterprise Information Systems: 13th International Conference, ICEIS 2011, Beijing, China, June 8-11, 2011, Revised Selected Papers*, pages 224–237. Springer Berlin Heidelberg.

[Bellegarda and Monz, 2015] Bellegarda, J. R. and Monz, C. (2015). State of the art in statistical methods for language and speech processing. *Computer Speech & Language*, 35:163–184.

[Ben Abdessalem et al., 2015] Ben Abdessalem, W., Ben Azzouz, Z., Singh, A., Dey, N., S. Ashour, A., and Ben Ghazala, H. (2015). Automatic builder of class diagram (ABCD): an application of UML generation from functional requirements. *Software: Practice and Experience*, 39(7).

[Berenbach et al., 2009] Berenbach, B., Paulish, D., Kazmeier, J., and Rudorfer, A. (2009). *Software &Amp; Systems Requirements Engineering: In Practice*. McGraw-Hill, Inc., New York, NY, USA, 1 edition.

[Beum-Seuk Lee, 2001] Beum-Seuk Lee (2001). Automated conversion from a requirements document to an executable formal specification. *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, page 437.

[Bhatia et al., 2013] Bhatia, J., Sharma, R., Biswas, K. K., and Ghaisas, S. (2013). Using Grammatical knowledge patterns for structuring requirements specifications. *2013 3rd International Workshop on Requirements Patterns, RePa 2013 - Proceedings*, pages 31–34.

[Bongers, 1946] Bongers, H. (1946). *Basic English Grammar*, volume 27.

[Bouquet et al., 2004] Bouquet, P., Giunchiglia, F., Van Harmelen, F., Serafini, L., and Stuckenschmidt, H. (2004). Contextualizing ontologies. In *Web Semantics*, volume 1, pages 325–343.

[Bryant, 2000] Bryant, B. (2000). Object-oriented natural language requirements specification. *Proceedings 23rd Australasian Computer Science Conference. ACSC 2000 (Cat. No.PR00518)*, pages 24–30.

[Bryant, 2002] Bryant, B. (2002). Formal specification of software systems using two-level grammar. In *[1991] Proceedings The Fifteenth Annual International Computer Software & Applications Conference*, volume 137, pages 155–160. IEEE Comput. Soc. Press.

[Bryant and Lee, 2002] Bryant, B. R. and Lee, B. S. (2002). Two-level grammar as an object-oriented requirements specification language. *Proceedings of the Annual Hawaii*

International Conference on System Sciences, 2002-Janua(c):3627–3636.

[Ceri et al., 2000] Ceri, S., Fraternali, P., and Bongio, A. (2000). Web Modeling Language (WebML): A Modelling Language for Designing Web Sites. *9th international World Wide Web conference on computer networks*, 33:137–157.

[Chen, 1976] Chen, P. P.-S. S. (1976). The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36.

[Chinosi and Trombetta, 2012] Chinosi, M. and Trombetta, A. (2012). BPMN: An introduction to the standard. *Computer Standards and Interfaces*, 34(1):124–134.

[Chioaşcă, 2012] Chioaşcă, E. V. (2012). Using machine learning to enhance automated requirements model transformation. *Proceedings - International Conference on Software Engineering*, pages 1487–1490.

[Collins, 2013] Collins, M. (2013). Lexicalized Probabilistic Context-Free Grammars Weaknesses of PCFGs as Parsing Models. *Lecture Notes*, (1):1–22.

[Dagnino, 2002] Dagnino, A. (2002). An evolutionary lifecycle model with Agile practices for software development at ABB. *Eighth Ieee International Conference on Engineering of Complex Computer Systems, Proceedings*, pages 215–223.

[Dahhane et al., 2015] Dahhane, W., Zeaaraoui, A., Ettifouri, E. H., and Bouchentouf, T. (2015). An automated object-based approach to transforming requirements to class diagrams. *2014 2nd World Conference on Complex Systems, WCCS 2014*, pages 158–163.

[Davis, 1990] Davis, A. M. (1990). *Software requirements: analysis and specification*. Prentice Hall, Englewood Cliffs, N.J.

[Deeptimahanti and Sanyal, 2011] Deeptimahanti, D. K. and Sanyal, R. (2011). Semi-automatic generation of UML models from natural language requirements. *Proceedings of the 4th India Software Engineering Conference on - ISEC '11*, pages 165–174.

[Denis and Baldridge, 2007] Denis, P. and Baldridge, J. (2007). A ranking approach to pronoun resolution. In *IJCAI International Joint Conference on Artificial Intelligence*, pages 1588–1593.

[Desai et al., 2015] Desai, A., Gulwani, S., Hingorani, V., Jain, N., Karkare, A., Marron, M., R, S., and Roy, S. (2015). Program Synthesis using Natural Language.

[Desai et al., 2016] Desai, A., Gulwani, S., Hingorani, V., Jain, N., Karkare, A., Marron, M., R, S., and Roy, S. (2016). Program synthesis using natural language. In *Proceedings*

of the 38th International Conference on Software Engineering, ICSE '16, pages 345–356, New York, NY, USA. ACM.

[Dzung and Ohnishi, 2013] Dzung, D. V. and Ohnishi, A. (2013). Evaluation of ontology-based checking of software requirements specification. *Proceedings - International Computer Software and Applications Conference*, pages 425–430.

[Fairley, 2009] Fairley, R. E. (2009). *Managing and leading software projects*. IEEE Computer Society ; Wiley, Los Alamitos, CA : Hoboken, N.J.

[Fatwanto, 2012a] Fatwanto, A. (2012a). Software requirements translation from natural language to object-oriented model. *Proceedings of 2012 IEEE Conference on Control, Systems and Industrial Informatics, ICCSII 2012*, pages 191–195.

[Fatwanto, 2012b] Fatwanto, A. (2012b). Specifying translatable software requirements using constrained natural language. *2012 7th International Conference on Computer Science & Education (ICCSE)*, (Iccse):1047–1052.

[Fatwanto, 2012c] Fatwanto, A. (2012c). Translating software requirements from natural language to formal specification. *Proceeding - 2012 IEEE International Conference on Computational Intelligence and Cybernetics, CyberneticsCom 2012*, pages 148–152.

[Fellbaum, 1998] Fellbaum, C. (1998). *WordNet: An Electronic Lexical Database*, volume 71. MIT Press, Cambridge.

[Ferrari et al., 2016] Ferrari, A., Spoletini, P., and Gnesi, S. (2016). Ambiguity and tacit knowledge in requirements elicitation interviews. *Requirements Engineering*, 21(3):333–355.

[Fockel and Holtman, 2015] Fockel, M. and Holtman, J. (2015). ReqPat: Efficient Documentation of High-Quality Requirements using Controlled Natural Language. *IEEE International Requirements Engineering Conference*, 23:280–281.

[Fowler and Highsmith, 2001] Fowler, M. and Highsmith, J. (2001). The agile manifesto. *Software Development*, 9:28–35.

[Friedrich et al., 2011] Friedrich, F., Mendling, J., and Puhlmann, F. (2011). Process Model Generation from Natural Language Text. *Lecture Notes in Computer Science*, 6741:482–496.

[Geetha and Anandha Mala, 2014] Geetha, S. and Anandha Mala, G. S. (2014). Automatic database construction from natural language requirements specification text. *ARPN*

Journal of Engineering and Applied Sciences, 9(8):1260–1266.

[Geetha and Mala, 2013] Geetha, S. and Mala, G. (2013). Extraction of key attributes from natural language requirements specification text. In *IET Chennai Fourth International Conference on Sustainable Energy and Intelligent Systems (SEISCON 2013)*, pages 374–379. Institution of Engineering and Technology.

[Githens, 2006] Githens, G. (2006). Managing Agile Projects by Sanjiv Augustine. *Journal of Product Innovation Management*, 23(5):469–470.

[Gordon and Harel, 2009] Gordon, M. and Harel, D. (2009). *Generating Executable Scenarios from Natural Language*, pages 456–467. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Granacki and Parker, 1987] Granacki, J. J. and Parker, a. C. (1987). PHRAN-SPAN: a natural language interface for system specifications. *24th ACM/IEEE conference proceedings on Design automation conference - DAC '87*, pages 416–422.

[Harmain and Gaizauskas, 2000] Harmain, H. and Gaizauskas, R. (2000). CM-Builder: an automated NL-based CASE tool. *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*, pages 45–53.

[Harris, 2012] Harris, I. G. (2012). Extracting design information from natural language specifications. *Proceedings of the 49th Annual Design Automation Conference on - DAC '12*, page 1256.

[Herchi and Abdessalem, 2012] Herchi, H. and Abdessalem, W. B. (2012). From user requirements to UML class diagram. *CoRR*, abs/1211.0713.

[Hobbs, 1977] Hobbs, J. R. (1977). Pronoun resolution. *SIGART Bull.*, (61):28–28.

[Ibrahim and Ahmad, 2010] Ibrahim, M. and Ahmad, R. (2010). Class diagram extraction from textual requirements using natural language processing (NLP) techniques. *2nd International Conference on Computer Research and Development, ICCRD 2010*, pages 200–204.

[IEEE, 2008] IEEE (2008). Iso/iec/ieee standard for systems and software engineering - software life cycle processes. *IEEE Std 12207-2008*, pages c1–138.

[Ilić, 2007] Ilić, D. (2007). Deriving formal specifications from informal requirements. *Proceedings - International Computer Software and Applications Conference*, 1(Compsac):145–152.

[Ilieva and Ormandjieva, 2005] Ilieva, M. G. and Ormandjieva, O. (2005). Automatic Transition of Natural Language Software Requirements Specification into Formal Presentation. *Language*, pages 392–397.

[Ishihara et al., 1993] Ishihara, Y., Seki, H., and Kasami, T. (1993). A translation method from natural language specifications into formal specifications using contextual dependencies. *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*, (M):232–239.

[ISO and IEC, 2002] ISO and IEC (2002). Information technology–Z formal specification notation–Syntax, type system and semantics. 2002:101.

[ISO/IEC and IEEE, 2010] ISO/IEC and IEEE (2010). ISO/IEC/IEEE 24765:2010 - Systems and software engineering – Vocabulary. *Iso/Iec Ieee*, 2010:410.

[Johnson, 1998] Johnson, M. (1998). PCFG models of linguistic tree representations. *Computational Linguistics*, 24(4):613–632.

[Kaur and Sengupta, 2010] Kaur, R. and Sengupta, J. (2010). Phased model for component driven approach in software development lifecycle. *2010 2nd International Conference on Computer Engineering and Technology*, pages V4–113–V4–116.

[Kaur and Sengupta, 2011] Kaur, R. and Sengupta, J. (2011). Software Process Models and Analysis on Failure of Software Development Projects. *International Journal of Scientific & Engineering Research*, 2(2):1–4.

[Konrad and Cheng, 2005] Konrad, S. and Cheng, B. H. C. (2005). Facilitating the construction of specification pattern-based properties. *13th IEEE International Conference on Requirements Engineering RE05*, (August):329–338.

[Kordon and Luqi, 2002] Kordon, F. and Luqi (2002). An introduction to rapid system prototyping. *IEEE Transactions on Software Engineering*, 28(9):817–821.

[Krishnan and Samuel, 2010] Krishnan, H. and Samuel, P. (2010). Relative extraction methodology for class diagram generation using dependency graph. *2010 IEEE International Conference on Communication Control and Computing Technologies, ICCCCT 2010*, pages 815–820.

[Kroll and Kruchten, 2003] Kroll, P. and Kruchten, P. (2003). *Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*.

[Landhäußer et al., 2012] Landhäußer, M., Kömer, S. J., Tichy, W. F., Landhauser, M., Ko-

mer, S. J., and Tichy, W. F. (2012). Synchronizing domain models with natural language specifications. *2012 1st International Workshop on Realizing AI Synergies in Software Engineering, RAISE 2012*, pages 22–26.

[Laudon and Laudon, 2012] Laudon, K. and Laudon, J. P. (2012). *SISTEMAS DE INFORMACIÓN GERENCIAL*. Pearson Educación de México, 12 edition.

[Lee and Bryant, 2004] Lee, B.-s. and Bryant, B. R. (2004). Automation of Software System Development Using Natural Language Processing and Two-Level Grammar. In Goos, G., Hartmanis, J., van Leeuwen, J., Wirsing, M., Knapp, A., and Balsamo, S., editors, *Resuscitation*, volume 2941, pages 219–233. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Leffingwell, 2011] Leffingwell, D. (2011). *Agile software requirements: lean requirements practices for teams, programs, and the enterprise*. The Agile software development series. Addison-Wesley, Upper Saddle River, NJ.

[Li et al., 2005a] Li, K., Dewar, R., and Pooley, R. (2005a). Computer-assisted and customer-oriented requirements elicitation. In *13th IEEE International Conference on Requirements Engineering (RE'05)*, pages 479–480. IEEE.

[Li et al., 2005b] Li, K., Dewar, R., and Pooley, R. (2005b). Object-oriented analysis using natural language processing. *Linguistic Analysis*, pages 75–76.

[Liu et al., 2004] Liu, D., Subramaniam, K., Eberlein, A., and Far, B. (2004). Natural language requirements analysis and class model generation using UCDA. *Innovations in Applied ...*, pages 295–304.

[Ma, 2002] Ma, Q. (2002). Natural Language Processing with Neural Networks. *Proceedings of the Language Engineering Conference (LEC '02)*, pages 1–12.

[Meziane and Vadera, 2004] Meziane, F. and Vadera, S. (2004). Obtaining e-r diagrams semi-automatically from natural language specifications. In *Sixth International Conference on Enterprise Information Systems (ICEIS 2004)*, pages 638–642. In volume 1 of conference proceedings ('Databases and information systems integration').

[Mitchell and Seaman, 2009] Mitchell, S. M. and Seaman, C. B. (2009). A comparison of software cost, duration, and quality for waterfall vs. iterative and incremental development: A systematic review. *2009 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM 2009*, (February 2008):511–515.

[Mittal and Chopra, 2011] Mittal, V. and Chopra, C. (2011). Potter Model - A Change Compliant Software Development Lifecycle Model. *2011 Second International Conference*

on Intelligent Systems, Modelling and Simulation, pages 66–70.

[More, 2012] More, P. (2012). Generating UML Diagrams from Natural Language Specifications. *International Journal of Applied Information Systems (IJAIS) – ISSN : 2249-0868 Foundation of Computer Science FCS, New York, USA Volume 1– No.8, April 2012 – www.ijais.org*, 1(8):19–23.

[Mossakowski et al., 2004] Mossakowski, T., Sannella, D., and Tarlecki, A. (2004). *CASL user manual - Introduction to using the common algebraic specification language - Introduction*, volume 2900 of *Lecture notes in computer science*. Springer, Berlin ; New York.

[Nikora, 2005] Nikora, A. P. (2005). Classifying requirements: Towards a more rigorous analysis of natural-language specifications. *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, 2005:291–300.

[Nishida et al., 1991] Nishida, F., Takamatsu, S., Fujita, Y., and Tani, T. (1991). Semiautomatic program construction from specifications using library modules. *IEEE Transactions on Software Engineering*, 17(9):853–871.

[Njonko and El Abed, 2012] Njonko, P. B. F. and El Abed, W. (2012). From natural language business requirements to executable models via SBVR. *2012 International Conference on Systems and Informatics, ICSAI 2012*, (Icsai):2453–2457.

[Object Management Group, 2015] Object Management Group (2015). Interaction Flow Modeling Language v1.0. (February).

[Object Management Group (OMG), 2011] Object Management Group (OMG) (2011). Business Process Model and Notation (BPMN) Version 2.0. *Business*, 50(January):170.

[Oliveira et al., 2006] Oliveira, A., Seco, N., and Gomes, P. (2006). A CBR Approach to Text to Class Diagram Translation. In *8th European Conference on Case-Based Reasoning*.

[Omar and Razik, 2008] Omar, N. and Razik, N. a. (2008). Determining the basic elements of object oriented programming using natural language processing. *2008 International Symposium on Information Technology*, pages 1–6.

[OMG, 2008] OMG (2008). Semantics of Business Vocabulary and Business Rules (SBVR), v1.0. OMG Available Specification. http://www.omg.org/docs/formal/08-01-02.pdf. (May):392.

[OMG, 2010] OMG (2010). OMG Unified Modeling Language TM (OMG UML), Superstructure v.2.3. *InformatikSpektrum*, 21(May):758.

[Overmyer et al., 2001] Overmyer, S., Benoit, L., and Owen, R. (2001). Conceptual modeling through linguistic analysis using LIDA. *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 401–410.

[Patel, 2014] Patel, G. A. (2014). Resolve the Uncertainity in requirement Specification to generate the UML Diagram. *2014 International Conference on Advances in Engineering and Technology (ICAET)*, (Icaet):1–7.

[Popescu et al., 2008] Popescu, D., Rugaber, S., Medvidovic, N., and Berry, D. M. (2008). Reducing ambiguities in requirements specifications via automatically created object-oriented models. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5320 LNCS:103–124.

[Project Management Institute, 2013] Project Management Institute, editor (2013). *A guide to the project management body of knowledge ({PMBOK} guide)*. Project Management Institute, Inc, Newtown Square, Pennsylvania, fifth edit edition.

[Raghothaman et al., 2016] Raghothaman, M., Wei, Y., and Hamadi, Y. (2016). Swim: Synthesizing what i mean: Code search and idiomatic snippet synthesis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 357–367, New York, NY, USA. ACM.

[Renssen, 2005] Renssen, A. V. (2005). *Gellish: a generic extensible ontological language - design and application of a universal data structure*. PhD thesis, TU Delft, Delft University of Technology.

[Rui, 2013] Rui, S. (2013). Translating software requirement from natural language to automaton. *Proceedings 2013 International Conference on Mechatronic Sciences, Electric Engineering and Computer (MEC)*, pages 2456–2459.

[Saeki et al., 1989] Saeki, M., Horai, H., and Enomoto, H. (1989). Software Development Process from Natural Language Specification. *Proceedings of the 11th International Conference on Software Engineering*, pages 64—-73.

[Sampson, 2003] Sampson, G. (2003). The Oxford Handbook of Computational Linguistics. *Literary and Linguistic Computing*, 18(3):333–336.

[Sannella and Wirsing, 1999] Sannella, D. and Wirsing, M. (1999). 8 Specification Languages. pages 243–272.

[Schwitter, 1996] Schwitter, R. (1996). Attempto-from specifications in controlled natural language towards executable specifications. *Arxiv preprint cmp-lg/9603004*.

[Selway et al., 2013] Selway, M., Grossmann, G., Mayer, W., and Stumptner, M. (2013). Formalising natural language specifications using a cognitive linguistics/configuration based approach. *Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC*, pages 59–68.

[Selway et al., 2015] Selway, M., Grossmann, G., Mayer, W., and Stumptner, M. (2015). Formalising natural language specifications using a cognitive linguistic/configuration based approach. *Information Systems*, 54:191–208.

[Shukur et al., 2002] Shukur, Z., Zin, A. M., and Ban, A. (2002). M2Z: A tool for translating a natural language software specification into Z. *Formal Methods and Software Engineering, Proceedings*, 2495:406–410.

[Sivaji et al., 2013] Sivaji, A., Abdullah, M. R., Downe, A. G., and Ahmad, W. F. W. (2013). Hybrid Usability Methodology: Integrating Heuristic Evaluation with Laboratory Testing across the Software Development Lifecycle. *2013 10th International Conference on Information Technology: New Generations*, pages 375–383.

[Smirnov, 2008] Smirnov, I. (2008). Overview of stemming algorithms. *Mechanical Translation*.

[Smith et al., 2003] Smith, R., Avrunin, G., and Clarke, L. (2003). From natural language requirements to rigorous property specifications. *Workshop on Software Engineering for Embedded Systems (SEES 2003) From Requirements to Implementation*, pages 40–46.

[Soares and Moura, 2015] Soares, H. A. and Moura, R. S. (2015). A methodology to guide writing Software Requirements Specification document. In *Proceedings - 2015 41st Latin American Computing Conference, CLEI 2015*.

[Song et al., 2004] Song, I., Yano, K., Trujillo, J., and Luján-Mora, S. (2004). A Taxonomic Class Modeling Methodology for Object-Oriented Analysis. *Information Modeling Methods and Methodologies, Advanced Topics in Databases Series*, pages 216–240.

[Steen et al., 2010] Steen, B., Pires, L. F., and Iacob, M.-e. (2010). Automatic generation of optimal business processes from business rules. pages 117–126.

[Takeuchi and Nonaka, 1986] Takeuchi, H. and Nonaka, I. (1986). The new new product development game. *Harvard business review*, 64(1):137–146.

[Tetreault, 2001] Tetreault, J. R. (2001). A Corpus-Based Evaluation of Centering and Pronoun Resolution. *Computational Linguistics*, 27(4):507–520.

[Turpin, 1996] Turpin, R. (1996). A progressive software development lifecycle. *Proceedings of ICECCS '96: 2nd IEEE International Conference on Engineering of Complex Computer Systems (held jointly with 6th CSESAW and 4th IEEE RTAW)*, pages 208–211.

[Videira et al., 2006] Videira, C., Ferreira, D., and Da Silva, A. R. (2006). A linguistic patterns approach for requirements specification. *Proceedings - 32nd Euromicro Conference on Software Engineering and Advanced Applications, SEAA*, 2004:302–309.

[Vidya Sagar and Abirami, 2014] Vidya Sagar, V. B. R. and Abirami, S. (2014). Conceptual modeling of natural language functional requirements. *Journal of Systems and Software*, 88(1):25–41.

[Walia and Carver, 2009] Walia, G. S. and Carver, J. C. (2009). A systematic literature review to identify and classify software requirement errors. *Information and Software Technology*, 51(7):1087–1109.

[Willett, 2006] Willett, P. (2006). The Porter stemming algorithm: then and now. *Program Electronic Library And Information Systems*, 40(3):219–223.

[Yan et al., 2015] Yan, R., Cheng, C.-H., and Chai, Y. (2015). Formal consistency checking over specifications in natural languages. *Design, Automation Test in Europe Conference Exhibition (DATE), 2015*, pages 1677–1682.

[Yuan, 2010] Yuan, L. (2010). Improvement for the automatic part-of-speech tagging based on hidden Markov model. *2010 2nd International Conference on Signal Processing Systems*, (1):V1–744–V1–747.

[Zapata, 2006] Zapata, C. M. (2006). UN – Lencep : Obtención Automática de Diagramas UML a partir de un Lenguaje Controlado. *Memorias del VII Encuentro Nacional de Computación ENC'06*, pages 254–259.

[Zeaaraoui et al., 2013] Zeaaraoui, A., Bougroun, Z., Belkasmi, M. G., and Bouchentouf, T. (2013). User stories template for object-oriented applications. *2013 3rd International Conference on Innovative Computing Technology, INTECH 2013*, pages 407–410.

[Zhou, X; Zhou, 2008] Zhou, X; Zhou, N. (2008). Auto-generation of Class Diagram from Free-text Functional Specifications and Domain Ontology. (2):1–20.