Universidad de Buenos Aires

Laboratorio de Fundamentos y Herramientas para la Ingeniería de Software (LaFHIS)

Reconstrucción Arquitectónica en Aplicaciones basadas en frameworks sobre datos de runtime

---

Tesis de Licenciatura

Ivan David Postolski

Director: Dr. Victor Adrián Braberman

Co-Director: Dr. Diego Garbervetsky

Marzo 2016

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

La comprensión de programas es una actividad vital del mantenimiento y evaluación del software. Tanto desarrolladores, testers como arquitectos necesitan comprender aspectos de estructura y comportamiento del software bajo análisis en sus tareas habituales [4] [5]. En particular, estudios recientes sobre las preguntas que se hacen los desarrolladores [6] [7], algunas incipientes técnicas [8] [9] y escuelas de documentación arquitectónica [10] ponen en evidencia que comprender el comportamiento potencial en tiempo de ejecución es especialmente importante.

En este sentido las vistas arquitectónicas del software usualmente se señalan como posibles vehículos facilitadores en tareas de mantenimiento y comprensión global de un sistema. Sin embargo mantener documentación actualizada sobre estas arquitecturas incluso en proyectos de tamaño reducido (70 mil a 280 mil líneas de código) suele ser demasiado costoso [11] y es particularmente relegado en proyectos ágiles donde la prioridad está puesta en el *time-to-market* [12]. Esto motiva la reconstrucción de arquitecturas como herramienta para obtener representaciones afines al sistema a partir del código fuente.

Dentro de este marco varias técnicas se han propuesto para recuperar arquitecturas como vistas de diseño en forma estática (sin necesidad de ejecutar el código) de forma automática y semi-automática [13] [14] [15] [16] [17]. La mayoría de estas técnicas obtienen vistas modulares del código [10]. Es decir, los elementos recuperados tienen presencia de tiempo de diseño y las dependencias entre ellos son básicamente las que se deducen de analizar sintácticamente los archivos que contienen el código. Varios estudios comparativos acerca de la efectividad de técnicas de recuperación [18] [19] [1] arrojan como resultado común que la precisión, en general, es baja al mismo tiempo que suelen resultar abstracciones de grano muy fino.

Más aun, las vistas construidas a partir de ellas solamente muestran una parte de la arquitectura y no permiten por ejemplo, descubrir estructuras derivadas de estilos

arquitectónicos de interacción y en muchos casos no son adecuadas para responder las preguntas vinculadas a los posibles caminos de interacción entre los elementos en tiempo de ejecución y violación de estilos o patrones [11] [1]. Por otro lado, dichas técnicas suelen ser evaluadas sobre programas de infraestructura (habitualmente en C) en donde todo el código está disponible para el análisis. Este escenario es, desde un punto de desafíos conceptuales y técnicos, distinto al que se encuentra en muchas aplicaciones modernas. Estas suelen estar construidas en base a funcionalidades de la orientación a objetos como el polimorfismo, el binding dinámico y el aliasing, usar bibliotecas de código no analizables y frameworks con inversión de control e inyección de dependencias agregando nuevos desafíos tanto para las reconstrucciones como para el análisis de programas [20] [21] [22].

Sin embargo, por su naturaleza la utilización de frameworks podría constituir una oportunidad para la detección automática de entidades con presencia en tiempo de ejecución y flujos de interacción que sean significativos para el usuario. Por ejemplo, los frameworks modernos para sistemas de información basados en web como Spring[1], o para aplicaciones móviles basadas en Android[2] promueven estilos y tácticas arquitectónicas (e.g., MVC, tiers, containers, transacciones, services, activities, etc.) para resolver cuestiones vinculadas a atributos de calidad tales como la modificabilidad, disponibilidad y escalabilidad. Este tipo de frameworks podría entenderse como una máquina virtual que brinda estos atributos de calidad a cambio de ser informado de alguna forma de la configuración de la aplicación en tiempo de ejecución.

A partir de lo expuesto anteriormente presentaremos en este trabajo un camino de reconstrucción arquitectónica con el objetivo de obtener vistas representativas de los componentes presentes en la ejecución para aplicaciones apoyadas sobre frameworks, definiendo como componentes a conjuntos de clases afines tanto por su interacción como por su relación en el dominio del programa y estudiando características propias de los frameworks para un caso específico, el framework Spring. Como materia prima de las reconstrucciones implementaremos un método capaz de analizar este framework y construir por medio de librerías de análisis de código un grafo de llamadas representativo de las posibles interacciones entre clases del sistema. Luego, sobre esta base propondremos dos técnicas novedosas de reconstrucción arquitectónica inspiradas en enfoques hallados en la literatura para acercar sus resultados a las vistas buscadas.

Por último realizaremos una evaluación de las técnicas propuestas y algunas de las técnicas más relevantes de la literatura sobre un caso de estudio representativo, utilizando herramientas de validación preexistentes y proponiendo nuevas dimensiones para verificar la calidad en este tipo de reconstrucciones.

En resumen las contribuciones del presente trabajo son las siguientes:

[1] https://spring.io/
[2] http://developer.android.com/sdk/index.html

- Un método para extraer interacciones y relaciones de tiempo de ejecución para aplicaciones basadas en el framework Spring (capítulo 5).
- Dos técnicas de reconstrucción diferentes apoyadas sobre la información obtenida por el método de análisis de Spring, construidas a partir de enfoques prometedores en la literatura de técnicas propuestas y modificaciones acordes a las vistas buscadas (capítulo 6).
- Propiedades y conceptos novedosos para la evaluación de reconstrucciones arquitectónicas a partir de datos de ejecución que no requieren la participación de expertos en el sistema (capítulos 8-10)

1.1. Estructura del trabajo

La tesis se compone por doce capítulos divididos en tres grandes partes: Antecedentes, Desarrollo y Resultados.

La primera parte del trabajo se encarga de introducir los antecedentes relacionados al estado del arte en técnicas de recuperación arquitectónica (capítulo 2), el estado del arte en la construcción de grafos de llamada con un panorama global de análisis de código (capítulo 3) y una introducción a las características principales de las aplicaciones basadas en frameworks y como son implementadas por Spring (capítulo 4).

La segunda parte presenta el desarrollo de la tesis, y se divide en dos capítulos. El primero abarca la presentación del método implementado para conseguir información del framework Spring y cómo a partir de ella se construyen los grafos de llamadas (capítulo 5) mientras que el segundo capítulo presenta las dos técnicas propuestas sobre la información provista por el método ya mencionado (capítulo 6).

La última parte del trabajo muestra los resultados obtenidos sobre un caso de estudio afín a las características mencionadas en los primeros capítulos. Estas evaluaciones se dividen en cuatro capítulos, el primero introduce las preguntas de investigación surgidas a partir del trabajo realizado y los objetivos de la tesis (capítulo 7), el segundo presenta el caso de estudio sobre el que se realizaran las evaluaciones (capítulo 8), el tercero muestra los resultados de precisión y recall asociados al método presentado en el capítulo 5 (capítulo 9) y finalmente el cuarto capítulo exhibe las evaluaciones propias de las reconstrucciones obtenidas a partir de las técnicas propuestas en el capítulo 6 y las técnicas halladas en la literatura introducidas durante el capítulo 2 (capítulo 10).

Por último se presentan dos capítulos más, correspondientes a las limitaciones en la validez de los resultados reportados y finalmente las conclusiones junto con las perspectivas de trabajo futuro que surgen a partir de esta tesis.

Parte I

Antecedentes

Capítulo 2

Técnicas conocidas de reconstrucción arquitectónica

Como fue mencionado en la Introducción existe un cierto consenso en la comunidad de la ingeniería de software acerca de las bondades en el entendimiento de la arquitectura de un software. Sin embargo también se reconoce la dificultad e imposibilidad de mantener documentación fiel acerca de la misma [11], esto abre la puerta a la reconstrucción de arquitecturas como documentación de sistemas, cuyo propósito es presentar de forma automática o semi-automática una vista del sistema cercana a la concepción mental de sus desarrolladores.

A partir de esta idea numerosas técnicas se han propuesto [16] [14] [18] [13] [17]. Estas toman como entrada dependencias de tiempo de compilación entre entidades (clases y/o archivos) como por ejemplo inclusiones o usos y devuelven una partición o descomposición del sistema. Estas descomposiciones consisten en agrupamientos de clases según los criterios de cada técnica e intentan dar una vista global del sistema, las reconstrucciones se pueden dividir entre *planas* o *jerárquicas* dependiendo si pueden o no existir varios niveles de agrupamiento, en este trabajo abordaremos únicamente el estudio de descomposiciones *planas*.

Entre las bondades del uso de dependencias de compilación se encuentran la performance y escalabilidad de las mismas, pudiendo en ciertos casos analizar proyectos de tamaño considerable como Google Chrome (9.7 millones de líneas de código ó 9.7MLOC, [1]) donde la ejecución de otro tipo de análisis de código como los presentados en el capítulo de desarrollo de esta tesis resultaría inviable. No obstante, resulta esperable que las vistas encontradas a partir de estas dependencias reflejen una visión de módulos del sistema (cercana a una perspectiva de diseño) pero no sean las mejores para explicar comportamientos en tiempo de ejecución.

Durante este trabajo se realizará un esfuerzo considerable por obtener reconstrucciones o vistas más cercanas al comportamiento en tiempo de ejecución por lo que resultará importante revisar aquellos conceptos sobre los cuales se rigen las técnicas conocidas de reconstrucción, en orden de entender las posibilidades de extensión, modificación de las mismas, y encontrar puntos de partida para la propuesta de técnicas nuevas.

Presentaremos en este capítulo una introducción a las técnicas más relevantes halladas en la literatura, sus motivaciones, objetivos y resultados reportados.

2.1. Enfoques de las técnicas

Principalmente las técnicas se dividen en dos enfoques dependiendo de la información con la que trabajan, estos son: *estructurales* o basados en información lexica. El primer grupo, los estructurales, orientan sus esfuerzos en particionar el sistema a partir de interacciones sintácticas y/o elementos propios del lenguaje de programación mientras que el segundo grupo intenta abstraerse de los elementos sintácticos y buscan encontrar elementos del dominio semántico del programa.

Dentro del enfoque estructural se pueden encontrar conceptos como:

- **Clustering sobre *features*:** Probablemente el concepto más estudiado para obtener particiones de un sistema. El mismo consiste en elegir algún vector de cualidades o *features* como por ejemplo: variables globales, macros, tipos de la entidad, inclusiones de archivos, identificadores, nombres de directorio, numero de lineas de código, etc; Luego se toma una definición de distancia entre los vectores, y se agrupan las entidades más cercanas con algoritmos estándar de clustering. Maqbool y Barbri [14] brindan en su estudio un panorama amplio sobre las técnicas que se fundaron sobre estos conceptos.
- **Búsqueda de patrones:** Las técnicas apoyadas sobre este concepto toman como prioridad la búsqueda de patrones clásicos [23] de diseño orientado a objetos argumentando la importancia y frecuencia de los mismos en la visión de alto nivel del programa. [24]
- **Optimización o búsqueda:** Este concepto consiste en definir alguna propiedad deseable de los sistemas, como por ejemplo la modularidad o la cohesión de sus módulos en forma de una función, para después buscar un máximo de la misma mediante herramientas de optimización y búsqueda (usualmente heurísticas) en el espacio de agrupamiento de entidades. [16]
- **Orientadas a la comprensión:** Los primeros tres enfoques engloban casi la totalidad del universo de técnicas disponibles a la hora de reconstruir la visión de

> un sistema, sin embargo existe un enfoque planteado e implementado solamente por una técnica que se diferencia del resto y se basa en condensar las lecciones aprendidas en las recuperaciones manuales previas de los autores. [13]

Mientras tanto el enfoque basado en información léxica se apoya en su totalidad en análisis de frecuencias de palabras a través de diversas técnicas, entre ellas modelado de tópicos mediante *Latent Dirichlet allocation* (LDA) [17] o algoritmos de *Expectation-Maximization* [25].

A veces las técnicas reportadas combinan algunos de estos conceptos en pos de mejorar sus resultados, presentaremos a continuación un breve resumen de las más prometedoras o relevantes en el estado del arte de la recuperación arquitectónica.

2.2. Técnicas prometedoras

2.2.1. Bunch

Bunch es una técnica creada en el año 1999 por Brian S. Mitchell [16], su objetivo es facilitar la mantenibilidad de un sistema por medio de la extracción automática de particiones del sistema. Su aporte es la interpretación de la obtención de estas particiones como un problema de optimización, para ello define una función llamada Modularization Quality (MQ) que intenta representar la calidad de una reconstrucción y luego utiliza algoritmos de hill climbing (NAHC y SAHC) y genéticos para encontrar una partición que maximice dicha función. Por lo tanto, Bunch se puede categorizar dentro de las técnicas de optimización apoyadas sobre información estructural.

Además de la contribución propia del algoritmo de clustering, Bunch propone una herramienta (figura 2.1) alrededor del mismo, comprendiendo otros procesos tales como el pre-procesamiento de archivos de código y la visualización de la descomposición encontrada.

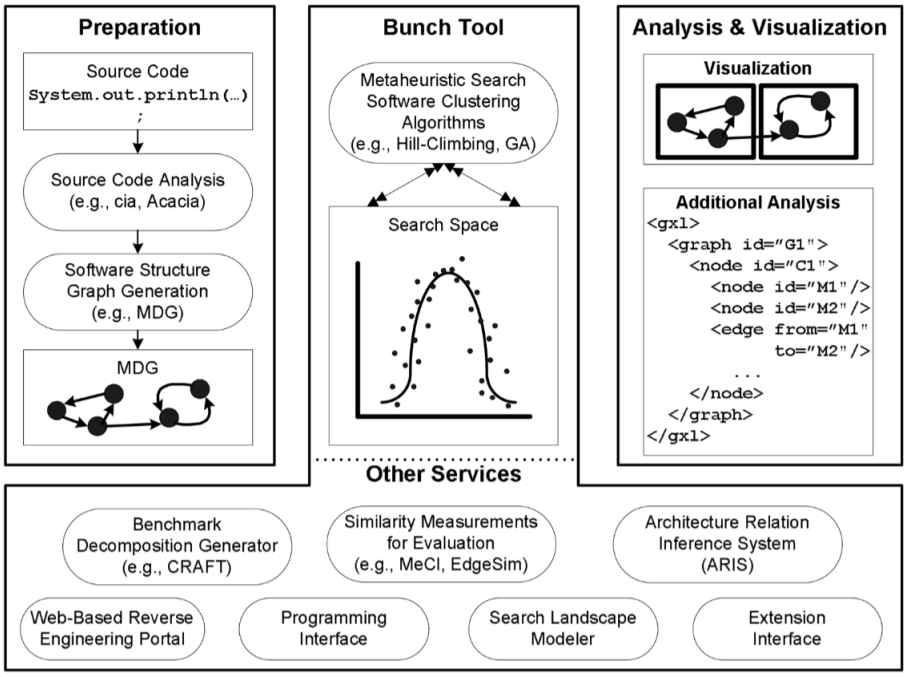


FIGURA 2.1: Arquitectura conceptual de la herramienta Bunch.

Los requerimientos de esta herramienta se enmarcan alrededor de la flexibilidad, portabilidad e integración pues uno de sus objetivos era brindar un instrumento de análisis y estudio para toda la comunidad científica. Esto se ve reflejado en la parte inferior del diagrama donde se explicitan las interfaces por la cual se brindan los distintos servicios asociados.

2.2.2. Algorithm for Comprenhension Driven Clustering

Algorithm for Comprenhension Driven Clustering (**ACDC**) es la técnica presentada por Tzerpos y Holt [13] en el año 2000 orientada a obtener un agrupamiento para la comprensión del sistema. En su paper de presentación, los autores se afirman críticos de la cantidad de esfuerzo puesto en mejorar técnicas que intentan optimizar funciones como modularidad o cohesión argumentando sobre las importancia de obtener diferentes vistas que agreguen valor y no perseguir la idea de una única vista útil. ACDC se auto-categoriza como una técnica basada en la comprensión, y utiliza información estructural.

El algoritmo propuesto por ACDC se apoya sobre dos conceptos, el primero consiste en la búsqueda de ciertos patrones que según sus autores se repiten en la comprensión de programas confeccionando un *esqueleto* de la recuperación con algunas entidades. Luego, en segundo lugar proponen otros patrones para el completado del esqueleto con las entidades que no pertenecen hasta el momento a ningún agrupamiento esta idea se

presenta como una solución para brindar estabilidad en caso de que se deseé reconstruir varias versiones del sistema.

Entre los patrones mencionados se destacan los siguientes tres,

- **Body Header:** Agrupa los archivos terminados en headers (.h) con las implementaciones (.c) correspondientes. Es parte de la construcción del esqueleto.
- **Subgraph Dominator:** Se basa en el concepto de nodos dominantes en un grafo e intenta encontrar los mismos con una heurística inspeccionando las entidades en orden creciente de grados de salida, en caso de encontrar un conjunto dominado se construye un agrupamiento con los elementos de este conjunto más el nodo dominante y se le asigna al agrupamiento el nombre de este último. En caso de que un conjunto de nodos dominados englobe otros conjunto de nodos dominados, ACDC se queda con el nivel más alto en la jerarquía. Como filtro de componentes utilitarios o sospechosos ACDC no tiene en cuenta para este patrón las entidades que contienen un grado de salida mayor de una constante, por defecto 20. Es parte clave de la construcción del esqueleto.
- **Orphan Adoption [26]:** Es un algoritmo para agrupar entidades que caen fuera del esqueleto de agrupamientos, como es posible que ciertas entidades no hayan sido agrupadas según los patrones mencionados, Orphan Adoption propone decidir en que agrupamiento se agregan las entidades libres a partir de una combinación entre información estructural y en última nombres de clases.

Por último los autores también definen algunos principios generales que según ellos son relevantes para el clustering de software, a saber:

- **Nombres efectivos de clusters** Se resalta la importancia de asignar nombres significativos a los agrupamientos para mejorar los refinamientos y el uso rápido de la partición. Es una de las pocas técnicas que mencionan este problema y lo atacan nombrando a cada agrupamiento según su nodo dominante (más el sufijo *.ss*), excepto aquellas entidades que se consideran como utilitarios y se engloban en el cluster llamado *support.ss*.
- **Cardinalidad limitada** Ciertas técnicas que intentan optimizar métricas como la modularidad o acoplamiento del sistema pueden encontrar soluciones que impliquen agrupamientos de gran tamaño, esto no es aconsejable para la lectura y el entendimiento del sistema. Por lo tanto, ACDC limita los agrupamientos de tamaño a como máximo 20 entidades.

2.2.3. Arquitectural Recovery using Concerns

Arquitectural Recovery using Concerns (**ARC**) es una técnica de agrupamiento presentada por García et al. [17] en el 2011 apoyada en recuperación de información y machine learning para obtener una reconstrucción arquitectónica. El objetivo principal de la investigación en la que se presenta esta técnica trata sobre la confección de una vista arquitectónica que permita comparar diferentes versiones de un programa para hallar de manera automática fenómenos indeseables como *drift* y *erosión*. Esta técnica pertenece a la familia de conceptuales ya que puede no precisar información estructural para realizar la descomposición.

El modelo estadístico en el que se basa ARC se llama *topic modelling* y busca encontrar por medio de análisis sobre la frecuencia de palabras, diferentes tópicos o temas. Un tópico será entonces una distribución probabilística de palabras, en la figura 2.2 se muestra un tópico hallado en un texto relacionado con cuestiones climáticas.

Tópico: Clima	
Palabra	Prob.
temperatura	0.7
viento	0.1
humedad	0.3

FIGURA 2.2: Ejemplo de tópico climático.

Para adaptar esta técnica a la recuperación arquitectónica es necesario un preproceso de los documentos (clases) donde se extraen las palabras consideradas como relevantes al agrupamiento (ie, nombres de intefaces, variables, documentación, etc) mientras que las palabras asociadas al uso común del idioma o palabras propias del lenguaje se excluyen del proceso. Por otro lado, ARC contempla una etapa de extracción de información estructural y una cooperación entre la misma y la información léxica, sin embargo en estudios recientes presentados por García et al. se afirma que implementativamente solamente es tenida en cuenta la parte léxica.

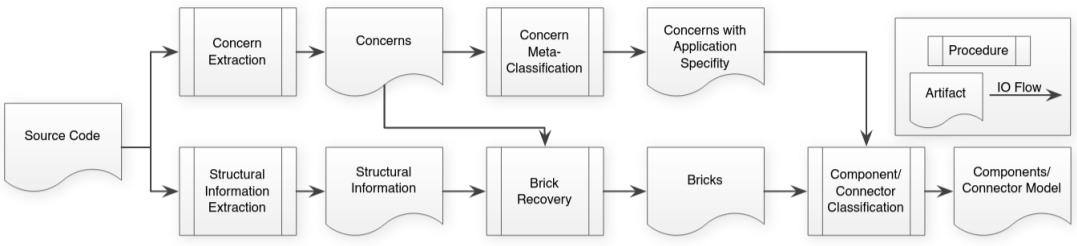


FIGURA 2.3: Esquema de recuperación de ARC.

Además en la publicación que introduce ARC se menciona la oportunidad de encontrar componentes y conectores basándose en diferenciar tópicos según elementos del dominio tecnológico y el dominio de negocios de la aplicación.

2.2.4. Técnicas basadas en clustering

Desde un punto de vista metodológico, algoritmos de clustering jerárquico, como *single-linkage*, *complete-linkage*,[27] [15] [18], no-jerárquicos como *k-means* [28] y clustering de grafos también han sido utilizados para agrupar elementos de software [29]. En este trabajo propondremos una técnica de reconstrucción basada en un algoritmo de esta índole, llamado Walktrap [30] desarrollado por Pons y Latapy para obtener comunidades en redes grandes, el mismo se basa en la idea de que caminos aleatorios en un grafo tienden a quedarse en la misma comunidad. Una diferencia interesante con los algoritmos mencionados anteriormente es que Walktrap no tiene como parámetro la cantidad de clusters esperado, sino que en vez de eso recibe la máxima distancia de caminos estudiados afectando indirectamente la cantidad de clusters pero sin fijar su límite.

2.3. Estudios sobre el desempeño de las técnicas

En cuanto a la evaluación de los estudios sobre técnicas de reconstrucción, el criterio más común se enmarca en la comparación entre el resultado de la técnica y una descomposición validada manualmente por algún experto del sistema usualmente llamada *ground-truth*. Dicha comparación requiere la definición de alguna medida de distancia o similaridad, es por ello que varías han surgido a lo largo del tiempo: MoJoFM [31] por ejemplo trata de contar el mínimo numero de movimientos y uniones requeridos para llevar una descomposición a otra y es la más usada en reportes previos. En caso de reconstrucciones jerárquicas, UpMoJo combina las operaciones de movimiento y unión con "*up*", con el fin de subir agrupamientos a niveles superiores.

Sin dudas el estudio más completo en esta dirección, por cantidad de sistemas analizados y diversidad de los mismos corresponde a García. et al [19]. En su trabajo se reportan varias descomposiciones manuales y se comparan los resultados de técnicas conocidas, entre ellas las mencionadas en la sección previa. En la figura 2.4 se pueden apreciar las características de los sistemas cuya *ground-truth* fue adquirida, para ello se definió un framework [11] (figura 2.5) que involucra tanto análisis funcional de la aplicación como una interacción registrada y controlada con expertos del sistema.

System	Ver	Dom	Lang	SLOC	Comps
ArchStudio	4	IDE	Java	280K	54
Bash	1.14.4	OS Shell	C	70K	25
Hadoop	0.19.0	Data Processing	Java	200K	68
Linux-C	2.0.27	OS	C	750K	7
Linux-D	2.0.27	OS	C	750K	120
Mozilla-C	1.3	Browser	C/C++	4M	10
Mozilla-D	1.3	Browser	C/C++	4M	233
OODT	0.2	Data Management	Java	180K	217

FIGURA 2.4: Características de los sistemas reconstruidos por García. et al. La columna "Comps" indica la cantidad de clusters en la *ground-truth*.

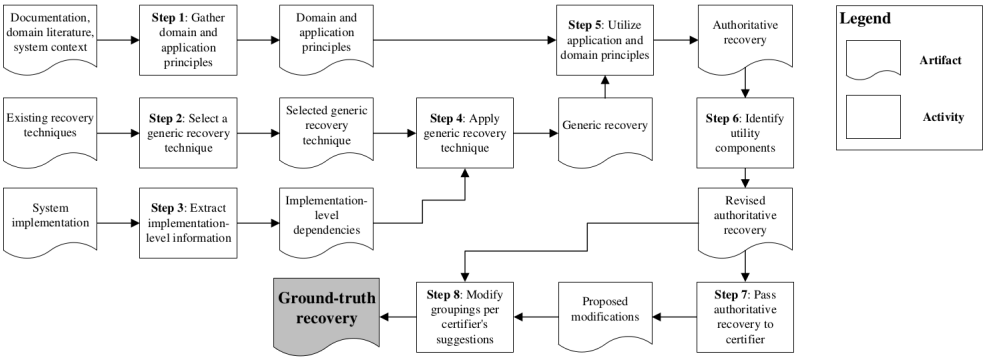


FIGURA 2.5: Framework para construcción de *ground-truth*.

García. et al combina los resultados de varias distancias o métricas, incluída MoJoFM (figura 2.6) con la finalidad de dar mayor certezas en los resultados obtenidos. Quizá la conclusión más importante de su trabajo esta en la notable variabilidad de precisión en las técnicas según el sistema a analizar. En sus evaluaciones ARC y ACDC producen por lo general los mejores resultados, con una distancia MoJoFM promedio de 58.76 % y 55.94 % respectivamente. Por otro lado, el rango de valores varía considerablemente según el sistema: 43 %-73 % para ARC, 36 %-88 % para ACDC, 33 %-74 % para Bunch-NAHC. Según los autores estos resultados indicarían que todavía hay un lugar significativo para mejoras sobre las técnicas actuales.

System	ARC	ACDC	WCA-UE	WCA-UENM	LIMBO	Bunch-NAHC	Bunch-SAHC	Z-Uni	Z-Tok	AVG
ArchStudio	76.28%	87.68%	49.73%	45.87%	31.20%	59.50%	50.07%	48.53%	39.47%	54.26%
Bash	57.89%	49.35%	41.56%	42.21%	27.27%	47.97%	38.51%	36.97%	36.97%	42.08%
Hadoop	54.28%	62.92%	42.15%	39.57%	19.23%	51.24%	46.95%	36.00%	45.91%	44.25%
Linux-D	51.47%	36.31%	33.51%	32.54%	18.46%	32.54%	31.14%	MEM	MEM	33.71%
Linux-C	75.72%	63.76%	61.98%	59.74%	57.70%	73.65%	75.13%	MEM	MEM	66.81%
Mozilla-D	43.44%	41.20%	MJE	MJE	MJE	40.18%	31.65%	MEM	MEM	39.12%
Mozilla-C	62.50%	60.30%	32.49%	32.40%	34.97%	69.02%	64.29%	MEM	MEM	50.85%
OODT	48.48%	46.01%	43.67%	41.97%	MJE	36.65%	31.56%	30.89%	33.57%	39.10%
AVG	58.76%	55.94%	43.58%	42.04%	31.47%	51.34%	46.16%	38.10%	38.98%	**45.15%**

FIGURA 2.6: Distancias MojoFM entre las reconstrucciones y las *ground-truth* por García et al. En gris oscuro se ven los mejores resultados para cada sistema, y en gris claro los segundos mejores. Tanto la última fila (AVG) como la última columna (AVG) representan el promedio.

El estudio más cercano en términos de objetivos perseguidos por esta tesis, es el último reporte de García et al [1], en este trabajo se propone una mejora en los resultados de las técnicas a partir de aumentar la precisión de las dependencias iniciales. Mientras que en su primer trabajo las dependencias consistían meramente en las inclusiones entre archivos, en el segundo trabajo se define la "dependencia de símbolos". Esta dependencia trata de refinar la anterior en base a nombres de funciones y variables, de modo que dos clases `A` y `B` solamente se relacionan entre sí en el caso que exista un acceso de `A` explicito a una variable o método contenido en `B`. En las figura 2.7 se pueden ver las diferencias de magnitud entre las dependencias y en la figura 2.8 la evaluación según la distancia MoJoFM.

Project	Version	Description	SLOC	File	Cluster†	Include Dep.	Symbol Dep.
Chromium	svn-171054	Web Browser	9.7M	18,698	67	1,183,799	297,530
ITK	4.5.2	Image Segmentation Toolkit	1M	7,310	11	169,017	30,784
Bash	4.2	Unix Shell	115K	373	14	2,512	2,481
Hadoop	0.19.0	Data Processing	87K	591	67	1,656	3,101
ArchStudio	4	Architecture Development	55K	604	57	866	1,697

FIGURA 2.7: Características de los sistemas reconstruidos por García. et al en [1].

Algorithm	Bash		ITK		Chrom.		ArchS.		Hadoop	
	Inc	Sym	Inc	Sym	Inc	Sym	Inc	Sym	Inc	Sym
ACDC	41	**59**	59	56	63	**70**	60	**77**	24	41
B-NAHC	44	47	37	41	28	31	52	59	29	29
B-SAHC	45	58	33	**62**	13†	**70**†	62	62	32	40
WCA-UE	28	37	31	32	23	23	32	33	14	17
WCA-UENM	28	34	31	32	23	23	32	33	14	17
LIMBO	27	28	31	31	TO	23	26	25	17	15
ARC	40		59		45		62		**49**	
ZBR-tok	35		MEM		MEM		48		29	
ZBR-uni	39		MEM		MEM		48		38	

FIGURA 2.8: La columnas **Inc** representan los resultados obtenidos por dependencias de inclusión y las columnas **Sym** representan los resultados obtenidos a partir de dependencias de símbolos. Resultados obtenidos por García et al.

Como se ve en la tabla, la precisión de las técnicas mejora utilizando dependencias refinadas, este trabajo motiva tomar el siguiente paso e intentar refinar aún más estas dependencias. Sin embargo, a diferencia de García et al. en esta tesis estudiaremos reconstrucciones basandonos en análisis de código más precisos (grafo de llamadas) con mayores desafíos. Además nos centraremos en un tipo de aplicación en especial no abordado por García et al, como son las aplicaciones basadas en frameworks, en especial sobre Spring.

Por otro lado, Shtern y Tzerpos [32] critican la validación a partir de *ground-truth* ya que como la misma parte de una opinion subjetiva del arquitecto o desarrollador esto podría implicar que múltiples reconstrucciones correctas podrían coexistir y que si un algoritmo no se ajusta a una descomposición manual podría ajustarse a otra del mismo sistema. Señalando que diferentes algoritmos de clustering no pueden ser comparados si sus objetivos son distintos, proponen un framework de evaluación con diferentes métricas relacionadas a la estabilidad y confiabilidad de un algoritmo basándose principalmente en sus cualidades estadísticas y en su desempeño bajo distintas versiones del sistema. En la misma línea Wu. et al [18] conduce comparaciones de distintos estudios combinando tres criterios, comparación con *ground-truth*, estabilidad bajo distintas versiones del software y la no-extrema distribución del cluster (muy pocos clusters grandes o demasiados clusters pequeños). Nuestro trabajo toma una parte de los criterios de Wu y Shtern et al. pero debido a que intentaremos encontrar vistas de comportamiento agregaremos validaciones a partir de comportamientos reales independientes de una reconstrucción validada por un experto.

Capítulo 3

Obtención de dependencias más cercanas al tiempo de ejecución

Como se mencionó en los antecedentes relacionados a las técnicas de reconstrucción la visión de las mismas se orienta por lo general a una perspectiva modular y de tiempo de compilación, sin embargo estudios acerca del entendimiento de software [5] [6] indican la necesidad de herramientas orientadas al comportamiento de ejecución. Por otro lado, como veremos en los capítulos siguientes, las aplicaciones que intentaremos analizar por lo general no son triviales en su puesta en marcha, esto surge debido a los lazos que las mismas requieren con otros servicios, entornos y frameworks como pueden ser bases de datos, conexiones de red privadas, software con licencias restringidas o accesos a servicios de terceros. Estas condiciones motivan el estudio de un tipo de análisis que no dependa de estas variables ligadas a la ejecución concreta de la aplicación pero que igualmente refleje propiedades vinculadas al desenvolvimiento de la misma. Veremos en este capítulo una introducción a ciertos análisis que poseen dichas características.

Los análisis estáticos inspeccionan el código fuente para intentar derivar comportamientos de ejecución, como casi todo programa tiene interacciones variables en su ejecución (entradas de usuario, internet, etc) los análisis buscan abstraerse de corridas concretas y en vez de eso buscan cubrir todas las posibilidades mediante suposiciones conservadoras. Las propiedades derivadas de estas suposiciones podrían ser más débiles que las reales del programa, pero están garantizadas de ser aplicables en cualquier corrida del mismo, de esta manera un análisis podría detectar comportamientos espurios que jamás ocurrirán pero no perderá de vista aquellos que si pueden ocurrir, cuando un análisis cumple con las condiciones mencionadas se lo califica como correcto o *sound*.

Históricamente estos análisis surgieron con el objetivo de ayudar a los compiladores en mejorar sus optimizaciones, por ejemplo encontrar código muerto, variables vivas, variables nulas o métodos no alcanzables. Sin embargo en la actualidad la aplicación de mayor interés científico está en la detección automática de vulnerabilidades, esta

temática trae aparejadas algunos obstáculos comunes a los del presente trabajo como el tratamiento de librerías y/o frameworks, particularmente en aplicaciones móviles como se verá en el capítulo siguiente.

Generalmente se dispone de dos enfoques diferentes a la hora de atacar un análisis estático: sistema de tipos y data-flow. Los sistemas de tipos consisten en asignar propiedades a ciertos componentes del programa para luego verificar (en tiempo de compilación) si estas se cumplen o no, por otro lado los análisis de data-flow buscan propiedades a partir de abstraer los estados del programa y sus flujos. Descartaremos para este trabajo los sistemas de tipos ya que los mismos requieren inherente mente un cierto trabajo manual y no se ajustan a los objetivos de automatización perseguidos. Sin embargo existen algunas técnicas que vale la pena mencionar alrededor de este concepto, este es el trabajo de Ducasse et al. [33] donde para brindar una visión de alto nivel y analizar el código primero se definen anotaciones de *ownership* entre objetos.

Volviendo sobre los análisis de data-flow, existen herramientas modernas de análisis de código proveedoras de este tipo de análisis, las mismas convierten la entrada (código fuente o bytecode) a representaciones intermedias desde las cuales les es más fácil operar y para modelar el flujo del programa analizado construyen grafos de control ó *control flow graph* (CFG) y grafos de llamadas ó *call graph* (CG). Mientras el primero representa la secuencia de estados intra-procedurales, el segundo contiene ejes relacionando sitios de llamados y destinos de los mismos. Usualmente no es trivial determinar con exactitud estos destinos debido al polimorfismo, por ejemplo si una clase `A` define el método `m()` y tiene una subclase `B`, el llamado `x.m()` puede referirse a la implementación de `A` ó `B`, dependiendo de la instanciación de `x`, que podría ser no resoluble estáticamente. Dependiendo del campo de aplicación estas situaciones se pueden mitigar de diferentes maneras y el grado de precisión considerado aceptable también puede variar. Comúnmente los grafos menos precisos se pueden generar rápidamente pero contienen más ejes espurios y por lo tanto requieren más tiempo en ser procesados. Además en el ejemplo se evidencia otra cualidad de los grafos de llamadas que es la intima relación entre los mismos y el análisis de referencias o punteros [3] [34].

Merecen un párrafo también los análisis dinámicos, construidos para reportar información detallada sobre corridas específicas del programa. Por lo general los mismos requieren menor cómputo y consumen menos tiempo, pero mientras en los análisis estáticos es complejo encontrar una abstracción con un buen balance entre entre precisión, recall y performance, para los dinámicos es difícil obtener un conjunto pequeño de ejecuciones que cubran todo el comportamiento relevante de la aplicación. Veremos más adelante en esta tesis la utilización de análisis dinámicos para validar resultados y comparar técnicas de agrupamiento.

En el afán de estudiar el comportamiento de las técnicas de reconstrucción cuando las mismas son aplicadas sobre dependencias de tiempo de ejecución, presentaremos el

estado del arte en la construcción de grafo de llamadas que luego utilizaremos para apalancar nuestro desarrollo y finalmente obtener la información de control sobre la cual propondremos reconstrucciones.

3.1. Construcción del grafo de llamadas

Se encuentran en la literatura varios algoritmos para la construcción de grafos de llamadas, sus diferencias varían principalmente el balance ya mencionado entre precisión, recall y performance. Entre ellos se destaca la familia surgida a partir del trabajo de Andersen [35] donde se definen una serie de reglas para cada estado del programa en base a inclusión de conjuntos (ver figura 3.1), luego estas condiciones se ingresan a un algoritmo de punto fijo para obtener su clausura y su resultado es una relación entre referencias y objetos (*points-to*) permitiendo así resolver los destinos de las llamadas polimorficas. Una forma bastante común de categorizar los algoritmos que se desprenden de este formalismo es según su sensibilidad para ciertos aspectos, estos pueden ser: campos (*field-sensitivity*), flujo (*flow-sensitivity*) y contexto (*context-sensitivity*).

Statement	Constraint	
`i: x = new T()`	$\{o_i\} \subseteq pt(x)$	[NEW]
`x = y`	$pt(y) \subseteq pt(x)$	[ASSIGN]
`x = y.f`	$\frac{o_i \in pt(y)}{pt(o_i.\texttt{f}) \subseteq pt(x)}$	[LOAD]
`x.f = y`	$\frac{o_i \in pt(x)}{pt(y) \subseteq pt(o_i.\texttt{f})}$	[STORE]

FIGURA 3.1: Reglas básicas de Andersen.

- **Field-sensitivity:** Se refiere a la capacidad del análisis para distinguir diferentes campos o *fields* para una representación de objeto en memoria. En la tabla se puede ver en las reglas `[LOAD]` y `[STORE]` cuando se define la inclusión en base a la instancia $\{o_i\}$ y el nombre del campo `f`, $pt(o_i.f) \subseteq pt(x)$. Mientras es posible conseguir una mejora de performance ignorando la relación entre instancias y campos (*field-insensitivity*) por ejemplo agrupando todos los campos solamente por su nombre (reemplazando $pt(o_i.f)$ por $pt(f)$), es notable la pérdida de precisión que esto puede ocasionar. Otra alternativa razonable para reducir complejidad es denominada *field based*, la idea radica en diferenciar campos según el tipo del objeto. Es decir, que campos de distintos objetos del mismo tipo se vuelven indistinguibles, perdiendo algo de precisión pero menos que field-insensitivity, esta ultima opción es particularmente prometedora en lenguajes con sistemas de tipos dinámico como JavaScript [36].

- **Flow-sensitivity:** Engloba los análisis que ponderan el orden de las instrucciones a analizar, como se puede ver en las reglas base de Andersen no se define el orden en las que se deben procesar y por ende su resultado es insensible al flujo, sin embargo ciertas representaciones intermedias construidas por las herramientas de análisis como *static single assignment* (SSA) provocan que los algoritmos tengan cierto grado de sensibilidad al flujo [3]. Para nuestro objetivo que radica en obtener las dependencias de llamados este tipo de sensibilidad no es del todo relevante.

- **Context-sensitivity:** Se trata de la sensibilidad más estudiada en la literatura, la misma se enmarca en cuanto o como se distinguen distintas invocaciones a un método. Se define como *contexto de llamada* a una abstracción del estado del programa al momento de invocar un método. Es fácil ver que analizar cada método en un contexto separado resultará en resultados más precisos, pero quizá tenga un impacto en el desempeño o *performance* del análisis. Veamos un ejemplo.

```
1 id(p) { return p; }
2 x = new Object(); // o1
3 y = new Object(); // o2
4 a = id(x);
5 b = id(y);
```

FIGURA 3.2: Ejemplo tomado de [2].

Un análisis insensible a contexto ó *context-insensitive* condensaría todos los posibles resultados de las llamadas en una única representación de `id`. Es decir, asumiría que cualquiera de los dos objetos `o1` y `o2` podrían ser devueltos por cualquier llamado a `id` esto provoca la conclusión que tanto `a` podría apuntar a `o2` y `b` a `o1`, un resultado claramente impreciso. Si en cambio se distinguen los llamados a `id` según las lineas en las que ocurren las invocaciones, el resultado sera que `a` solamente apuntara a `o1` y `b` a `o2`.

Diversas abstracciones fueron presentadas para distinguir los contextos a lo largo de los años, entre ellas se destacan dos: call-site sensitivity y object-sensitivity. La primera, call-site sensitivity [37][38] es la más antigua y por ende más conocida abstracción para armar contextos, la misma consiste en utilizar los sitios de llamadas como contextos. Es decir, cuando se analiza una invocación a un método se la diferencia por la línea de código en la que se encuentra, lógicamente esta distinción se propaga a los métodos que son llamados por el mismo hasta una profundidad definida como k, generalmente se denomina a esta cualidad como k-call-site-sensitive. Para eliminar la imprecisión del ejemplo mencionado basta con tomar un algoritmo con una sensibilidad 1-call-site-sensitive.

Por otro lado, object-sensitivity usa como abstracción los pedidos de memoria (ie, *allocation sites*). Específicamente, califica las variables locales de cada método con el sitio donde se instanció el objeto receptor del llamado al método. Este tipo de

información de contexto no es local, ya que no se puede obtener meramente de la línea en la que se encuentra el llamado al método sino que depende intrínsecamente en la precisión del análisis para deducir que objeto será el receptor. Es importante resaltar que el primer ejemplo (figura 3.2) podría no resolverse con este tipo de contextos ya que el llamado a `id(..)` se hace desde el mismo objeto `this`. Mostraremos a continuación un ejemplo modificado en el que esta elección de contexto resulta beneficiosa.

```
class S {
  Object id(Object a) { return a; }
  Object id2(Object a) { return id(a);}
}
class C extends S {
  void fun1() {
    Object a1 = new A1();
    Object b1 = id2(a1);
  }
}
class D extends S {
  void fun2() {
    Object a2 = new A2();
    Object b2 = id2(a2);
  }
}
```

FIGURA 3.3: Ejemplo tomado de [3].

Un análisis con object-sensitivity mantiene totalmente la precisión en el código del ejemplo incluso si se limitara a soportar solamente una instanciación. El objeto receptor de los dos llamados a `id2` (implícitamente `this`) son objetos abstractos diferentes (`C` y `D`) por lo tanto el llamado a `id` dentro de `id2` es analizado al menos dos veces, una por cada contexto/objeto receptor. El resultado es que el los objetos que fluyen hacia `b1` y `b2` se mantienen en caminos separados. Según Smaragdakis et al. [3] mucho del poder de la sensibilidad a objetos viene de su tolerancia a varios niveles de indirección mientras el objeto receptor siga siendo el mismo. En contraparte un análisis con sensibilidad de call-site de profundidad 1 hubiera perdido precisión justamente por la indirección causada entre `id2` e `id`.

Sridharan [2] especifica estos tipos de sensibilidad extendiendo las reglas base de Andersen como se puede ver en la figura 3.4 mediante la modificación de dos funciones *selector* y *heapSelector*. Selector se encarga de elegir bajo que contexto se analizará el metodo que se esta invocando mientra que heapSelector determina la abstracción utilizada cuando se instancia un objeto nuevo.

Statement in method m	Constraint	
`i: x = new T()`	$\dfrac{c \in contexts(m)}{\langle o_i, heapSelector(c)\rangle \in pt(\langle x, c\rangle)}$	[New]
`x = y`	$\dfrac{c \in contexts(m)}{pt(\langle y, c\rangle) \subseteq pt(\langle x, c\rangle)}$	[Assign]
`x = y.f`	$\dfrac{c \in contexts(m) \qquad \langle o_i, c'\rangle \in pt(\langle y, c\rangle)}{pt(\langle o_i, c'\rangle.\mathtt{f}) \subseteq pt(\langle x, c\rangle)}$	[Load]
`x.f = y`	$\dfrac{c \in contexts(m) \qquad \langle o_i, c'\rangle \in pt(\langle x, c\rangle)}{pt(\langle y, c\rangle) \subseteq pt(\langle o_i, c'\rangle.\mathtt{f})}$	[Store]
`j: x = r.g(a_1,...,a_n)`	$\dfrac{\begin{array}{c} c \in contexts(m) \qquad \langle o_i, c'\rangle \in pt(\langle r, c\rangle) \\ m' = dispatch(\langle o_i, c'\rangle, \mathtt{g}) \\ argvals = [\{\langle o_i, c'\rangle\}, pt(\langle a_1, c\rangle), \ldots, pt(\langle a_n, c\rangle)] \\ c'' \in selector(m', c, j, argvals) \end{array}}{\begin{array}{c} c'' \in contexts(m') \\ \langle o_i, c'\rangle \in pt(\langle m'_{this}, c''\rangle) \\ pt(\langle a_k, c\rangle) \subseteq pt(\langle m'_{p_k}, c''\rangle),\ 1 \leq k \leq n \\ pt(\langle m'_{ret}, c''\rangle) \subseteq pt(\langle x, c\rangle) \end{array}}$	[Invoke]
`return x`	$\dfrac{c \in contexts(m)}{pt(\langle x, c\rangle) \subseteq pt(\langle m_{ret}, c\rangle)}$	[Return]

Figura 3.4: Reglas extendidas de Andersen.

Habiendo introducido las características más sobresalientes en la construcción de grafos de llamadas, podemos ahora dar un poco más de detalle sobre la relación compleja entre precisión, recall y performance. Intuitivamente se puede caer en el error de pensar en que mayor precisión implicará peor performance debido a la necesidad de crear más estados (contextos) y de propagar mayor información. Sin embargo este no siempre es el caso, ya que a veces mejorar la precisión elimina porciones considerables de estados espurios y reduce considerablemente el tiempo de computo [39] [40] [41].

Además las evaluaciones de estos algoritmos en aplicaciones reales a menudo exceden los límites razonables de tiempo y/o memoria dependiendo muchas veces no solo de la sensibilidad y características tecnológicas sino también del tipo de aplicación, tamaño, etc. Este fenómeno sumado a la falta de registros del desempeño de algoritmos frente a escenarios como los que se presentarán en el capítulo 8 de esta tesis (aplicaciones basadas en frameworks), motivará una parte de las evaluaciones presentadas en capítulo 9.

3.2. Herramientas

Presentaremos una breve introducción a las herramientas actuales en materia de análisis de código, estas son: Wala, Soot y Doop.

Wala[1] es una herramienta de análisis de código desarrollada por investigadores del laboratorio de IBM T.J. Watson[2]. Una versión de su código es abierto y se mantiene en desarrollo tanto por investigadores pertenecientes al laboratorio como por el resto de la comunidad. Entre las funcionalidades centrales o *core* de Wala se encuentran el soporte de código binario Java y código JavaScript, herramientas para componer análisis inter e intra procedimientos de tipo data-flow, una representación intermedia instrumentable (SSA) y la implementación de algunos análisis y estructuras de datos que veremos a continuación. Hoy en día el mayor volumen de su desarrollo se enfoca en mejorar los análisis para JavaScript como [42] y proveer herramientas que se adapten bien para analizar aplicaciones móviles en sus respectivas plataformas (iOS, Android, Tizen, Windows Phone and Firefox OS).

Dentro de sus capacidades en materia de grafos de llamadas Wala nos ofrece varios algoritmos.

- **RTA** Rapid Type Analysis. Es un análisis simple insensible al contexto, donde cada parámetro o variable se analiza solamente en base a su tipo declarado y se considera como posible valor cualquier instancia creada previamente de ese tipo. Se podría pensar que mantiene un único contexto por programa.

```
1    I a = new A();
2    I b = new B();
3    b.m();
```

  En el ejemplo RTA considerará ambos objetos (`a` y `b`) como posibles destinatarios del llamado `m` ya que ambos poseen el mismo tipo declarado `I` y fueron instanciados alguna vez en el programa. Como se puede ver este tipo de analisis es sumamente impreciso y en la mayoría de los casos su uso es justificado a partir de la rapidez y proporcionar una primera aproximación.

- **0-CFA** Análisis insensible al contexto, más preciso que RTA ya que en vez de encontrar los destinatarios de métodos por su tipo, 0-CFA guarda para cada variable el tipo concretos de cada `new`, por lo tanto en el ejemplo solamente el tipo `B` sera analizado en la invocación `b.m()`.

- **0-1-Container-CFA** Este análisis ofrece la misma insensibilidad que 0-CFA, cambiando ligeramente la información usada en los sitios de instanciación de objetos (0-CFA utiliza tipos concretos y 0-1-CFA además la línea en la que fue instanciado) por una más precisa y además ofrece *object-sensitivity* en clases contenedoras, es decir colecciones de Java.

- **k-CFA** Brinda k grados de sensibilidad al contexto del tipo *call-site*.

[1] https://github.com/wala/WALA
[2] https://www.research.ibm.com/labs/watson/

Soot[3] es una herramienta de análisis de código implementada por el grupo de investigación Sable en la Universidad de McGill. Su código es abierto y también se mantiene en continuo desarrollo. A diferencia de Wala, Soot esta centrado solamente en el análisis de código Java y no soporta JavaScript. Soot también tiene su propia representación intermedia llamada Jimple y provee marcos de referencia para componer análisis inter e intra procedimiento. A su vez gran parte de su desarrollo actual se focaliza en mejorar sus capacidades frente a los problemas introducidos por aplicaciones móviles. FlowDroid [20] es su componente principal de análisis para aplicaciones Android.

Dentro de sus capacidades en materia de grafos de llamada Soot ofrece los siguientes algorítmos.

- **CHA** Es el más simple de los algoritmos para construcción de grafo de llamada y el más impreciso, Soot lo ofrece como algoritmo por defecto y consiste en por cada método polimórfico que encuentra CHA aproxima su ejecución por todas las clases que lo implementan, a diferencia de RTA no tiene en cuenta las instanciaciones por lo cual es todavía más impreciso.
- **Spark** Soot recomienda ejecutar Spark [34] para obtener un compromiso entre performance y precisión. El algoritmo central de Spark esta basado en inclusión de conjuntos (Andersen) pero también se jacta de implementar mejoras reportadas por otros autores, el mismo es insensible a contexto y su precisión es comparable a 0-1-CFA.

Por último Doop[4] es una herramienta desarrollada por el Departamento de Informática y Telecomunicaciones de la Universidad de Atenas, su objetivo es solamente la construcción de grafos de llamadas y análisis de punteros por lo que no posee tantas funcionalidades generales de análisis como Soot y Wala. De hecho, hace uso de Soot y su representación intermedia como entrada para sus análisis. A diferencia de las herramientas anteriores Doop esta construido sobre la idea de expresar sus análisis declarativamente en lenguaje Datalog para luego resolver los mismos mediante un motor llamado LogicBlox. Doop ofrece una buena variedad de algoritmos con diferentes sensibilidades a contexto, no obstante el mismo requiere una licencia comercial para su motor LogicBlox y por ende no será evaluada en este trabajo.

Todas las herramientas mencionadas poseen una amplia cantidad de publicaciones en las conferencias más prestigiosas de análisis de código e ingeniería de software, y serán tenidas en cuenta para intentar conseguir la información de runtime buscada.

[3] http://sable.github.io/soot/
[4] http://doop.program-analysis.org/

3.3. Desafíos

Dejando de lado los formalismos teóricos de los algoritmos y las cualidades técnicas de las herramientas, existen problemas relacionados con la tecnología y sus usos en la práctica que afectan notablemente los puntos clave de un grafo de llamadas. Enunciaremos a continuación los problemas que son pertinentes para las tecnologías que se engloban en el alcance de esta tesis.

- **Reflexión** o *reflection*, en lenguajes como Java se permite el uso de herramientas de meta-programación basada en nombres, por ejemplo el siguiente código instancia vía reflection un objeto del tipo `x`.

```
class Factory {
  Object make(String x) {
    return Class.forName(x).newInstance();
  }
}
```

  Analizar este código de una manera conservativa o *sound* podría llevar a concebir que `x` podría representar cualquier tipo y terminar en un grafo de llamadas extremadamente impreciso. Por otro lado, intentar aplicar heurísticas como analizar strings y flujos de datos del programa reducirían la impresición pero anularían las garantías de *soundness*. Así es que *reflection* trae el primer problema tecnológico no resuelto a la hora de construir grafos de llamadas.

  Entre los esfuerzos mencionables en este sentido se destaca el trabajo de Smaragdakis et al [43] donde se da una descripción más abarcativa del problema y se proponen nuevas heurísticas con la finalidad de conseguir un "*soundness empírico*" que brinde garantías bajo una imprecisión razonable.

  A los fines prácticos las herramientas actuales tienen diversas formas de manejar estos inconvenientes. Wala ofrece por defecto un análisis de cadenas de strings y por otro lado opciones que incluyen ignorar los llamados reflectivos, casteos e incluso desactivar el análisis de strings. Soot y Doop no proveen por defecto capacidades de soporte para reflection sino que brindan una herramienta separada, TamiFlex [44] para el pre-proceso de las instrucciones que involucran estas características y la posibilidad de integrar su resultado con el análisis global.

- **Librerías.** La mayor parte de las aplicaciones Java construidas en la actualidad se edifican sobre librerías o frameworks complejos, estos usualmente resuelven problemas particulares de manera eficiente, en una forma aceptable según las necesidades de la aplicación en cuestión. Se pueden encontrar varios patrones de interacción entre una aplicación y sus librerías, como por ejemplo: call-return, call-backs o

incluso como veremos más adelante inversión de control. Analizar de forma conservativa o *sound* estas interacciones puede derivar en el análisis de una base de código muchísimo más grande (aplicaciones y librerías) y compleja sin garantías de obtener información relevante debido a falta de precisión y usos de tecnología no soportada. Por ende, casi todas las herramientas de análisis poseen opciones para ignorar librerías o parte de ellas sacrificando la correctitud pero mejorando la performance.

Vale la pena mencionar los trabajos de Tang et al. [22] donde sé propone un método que al parecer mejora la performance para encontrar call-backs entre librerías y código de la aplicación, y Lhoták et al. [45] en el cual se describe en detalle la problemática.

Tanto Wala como Soot (y por ende Doop) tienen parámetros en los cuales se pueden excluir librerías por medio de nombres o máscaras de paquetes.

Capítulo 4

Aplicaciones basadas en frameworks

Durante los últimos años hemos presenciado un auge en el desarrollo de aplicaciones de servicios, móviles y paginas web. Este hito dentro de la historia del software se ve acompañado e impulsado por herramientas orientadas a facilitar y mejorar las condiciones de desarrollo y funcionamiento de este tipo de sistemas. Si bien estos frameworks otorgan facilidades, en su afán de proveer versatilidad en su configuración los mismos incurren en características tecnológicas que plantean un problema serio a la hora de comprender interacciones en tiempo de ejecución y colateralmente dificultan los análisis estáticos de código.

Dichas herramientas se presentan en diferentes formas, entre ellas se encuentran las orientadas a aplicaciones móviles como Android Studio[1] o Xcode para Iphone SDK[2] que proveen un entorno integrado con distintas versiones de su librerías de desarrollo (o *sdk*) y capacidades de emulación de dispositivos, logging y utilidades para el diseño visual. Por otro lado, las orientadas a la construcción de servicios como los frameworks Spring[3], Play[4] o Ruby on Rails[5] se concentran en mejorar la adopción en entornos familiares al programador y ofrecen atributos de calidad y funcionalidades comunes a cambio de una configuración.

Sin embargo existen ciertos puntos en común entre los frameworks relevantes a este trabajo. El primero yace en la instanciación y puesta en funcionamiento de estas aplicaciones, cada uno de los frameworks mencionados implementa alguno de los dos diseños más populares para construir aplicaciones sobre ellos, estos son: la implementación de ciertas interfaces definidas por el framework ó la anotación con algún mecanismo de

[1] http://developer.android.com/sdk/index.html
[2] https://developer.apple.com/ios/
[3] https://spring.io/
[4] https://www.playframework.com/
[5] http://rubyonrails.org/

meta-datos como puede ser escribiendo un archivo .xml, YAML, HOCON, etc. Mediante estas estrategias tecnológicas es posible delegar de forma desacoplada la instanciación, configuración y posterior ejecución al framework, la consumisión de servicios y los accesos a bases de datos.

Otro punto en común es la abstracción de puntos de entrada, es importante señalar que las aplicaciones modernas no se encuadran dentro del modelo mental clásico de programación en el cual uno define un punto de entrada como *main* sino que se desarrollan pensando en la interacción y definiendo que acción se ejecutará según el estímulo originado por agentes externos como interacción de usuarios, pedidos HTTP, etc. Abstrayendo el procesamiento de bajo nivel de estas acciones en los frameworks, como por ejemplo abrir un socket, escuchar un puerto o recibir un pedido. Según su naturaleza y propósito, cada framework elige las abstracciones que considera convenientes para proveer al programador la creación de estos puntos de entrada. Y a su vez será en tiempo de ejecución quien en última instancia delegará el control al código provisto por el programador.

A continuación daremos una descripción de cómo Spring expone estas funcionalidades (inyección de dependencias e inversión de control) y cuales son las consecuencias prácticas para las técnicas actuales de análisis de código.

4.1. Spring

Spring es un framework de código abierto escrito en Java para la construcción de servicios web, el mismo fue creado por Rod Johnson en 2003 y se mantiene en continuo desarrollo hasta el día de hoy por lo que comúnmente es considerado un framework maduro en materia de la confección de servicios web. Es posterior a otras herramientas como EJB y Struts y antecesor de Play (Scala) y Ruby on Rails (Ruby).

Dentro de sus características principales se encuentran: inyección de dependencias, modelo-vista-controlador, transacciones, programación orientada a aspectos, servicios de seguridad y servicios de conectividad para bases de datos, entre otros. Como vimos en la sección anterior pondremos el foco en las dos primeras, la inyección de dependencias y el modelo-vista-controlador debido a que representan la implementación concreta de los conceptos comunes asociados a los frameworks y además fueron responsables de la innovación en este tipo de tecnología.

Presentaremos a continuación un breve ejemplo acerca de cómo es la configuración de un servicio básico, su instanciación y su posterior ejecución.

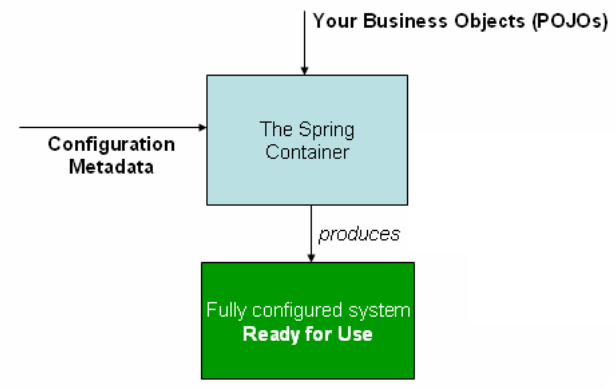
4.1.1. Inyección de dependencias

Resumiremos la inyección de dependencias o inversión de control en Spring en dos aspectos fundamentales, la instanciación de los objetos del sistema y su interconexión, para ello definiremos algunos términos específicos de Spring.

Llamaremos **dependencias** a las relaciones entre objetos que se harán presentes en tiempo de ejecución pero que no son posibles evidenciar desde el código fuente. Los objetos o clases que interactuarán en tiempo de ejecución con el framework se denominan *objetos de negocio* o Plain Old Java Object (**POJOs**), los mismos no deberían tener ninguna dependencia por fuera de aquellas definidas en el modelo de negocio.

La instanciación de un objeto de negocio o POJO se conoce bajo el nombre de **Bean**, este objeto es un Singleton al menos que el desarrollador explicite lo contrario, y poseé por lo general un nombre único que lo identifica en el contexto de ejecución.

Por otro lado, la interfaz `org.springframework.context.ApplicationContext` representará el contenedor o contexto de ejecución y será la responsable de intanciar, configurar y inter-conectar los beans. La misma buscará las instrucciones de qué objetos instanciar, configurar e inter-conectar leyendo una configuración provista por metadatos. Esta configuración podrá ser representada en XML, anotaciones o código Java.



Resulta importante destacar que en la mayoría de los escenarios de uso reales, no es necesaria la instanciación explicita de las clases del framework Spring ya que los contenedores web más comunes como Apache Tomcat, WebLogic, WebSphere, Resin, GlassFish y JBoss se encargan por si mismos de la instanciación, y el aislamiento de cada aplicación. Mostraremos a continuación un ejemplo elemental a partir de dos clases configuradas para utilizar Spring IoC mediante anotaciones Java.

Durante la inicialización del ejemplo, el contenedor de Spring (una implementación de la interfaz `ApplicationContext`) buscará en el class path de la aplicación por aquellas clases del paquete indicado por la etiqueta `context:component-scan` en el archivo de configuración `applicationContext.xml` (figura 4.2) anotadas con `@Component`, encontrará dos clases: `example.A` y `example.Bi`. A continuación las instanciará utilizando

```
1  package example;
2
3  @Component(name="a")
4  public class A {
5
6    @Resource("b")
7    public B bfield;
8
9    public String m() {
10     return bfield.m();
11   }
12 }
13
14 @Component(name="b")
15 public class Bi implements B {
16
17   @Override
18   public String m() {
19     return "static/index.html";
20   }
21 }
22
```

FIGURA 4.1: Beans `a` y `b` configurados a partir de anotaciones.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans:beans ...>
3
4     <!-- Enables the Spring MVC @Controller annotation programming model -->
5     <annotation-driven />
6
7     <!-- Select package to scan for components  -->
8     <context:component-scan base-package="example" />
9
10 </beans:beans>
```

FIGURA 4.2: Archivo de configuración de Spring `applicationContext.xml`.

reflection mediante su constructor por defecto y registrará los objetos en un diccionario bajo sus nombres de bean: `a` y `b`. Luego analizará las dependencias de cada uno de ellos y las inyectará, en este caso particular establecerá la referencia del campo `bfield` de `a` en la instancia ya creada de `b`.

Gracias a la versatilidad de las implementaciones del contenedor será posible instanciar este mismo ejemplo mediante distintas configuraciones, en caso de querer ahondar en los detalles específicos del framework, la documentación de referencia [46] y la guia [47] exponen varias alternativas según las necesidades del programador.

4.1.2. Modelo vista controlador

Vimos en la sección previa como Spring se encarga de instanciar los beans e inyectar sus dependencias entre si. Esta sección mostrará algunas de las facilidades que exhibe para construir servicios web sobre el sistema ya instanciado.

El diseño arquitectónico impulsado por Spring para la construcción de servicios web es el modelo-vista-controlador basado en pedidos o *requests*. La principal ventaja de este modelo es liberar al programador del código relacionado al procesamiento de acciones de bajo nivel mientras se lo proveeé de una plataforma extensible y eficiente.

- **Controladores** Son los objetos responsables del procesamiento de la entrada del usuario presentada en pedidos HTTP, invocando las funcionalidades necesarias en los objetos de negocio y finalmente devolviendo el modelo a visualizar con la vista correspondiente.
- **Modelo** Son los objetos que contienen los datos obtenidos en la ejecución de la capa de negocio y que deben ser mostrados en la respuesta.
- **Vista** Son los objetos responsables de la renderización del modelo en la respuesta, existen varias tecnologías disponibles que se integran con Spring como JSP, templates de Velocity, generadores de PDF o Excel. Las vistas no son responsables de actualizar datos o obtener datos, sino que solamente se encargan de visualizar el Modelo brindado por los Controladores.

La integración de inyección de dependencias con diversas tecnologías modelo-vista-controlador esta incluida en la implementación del framework, ofreciendo libertad al desarrollador para construir diferentes servicios sobre la misma capa de negocios. A los fines de esta tesis nos centraremos en la implementación difundida por Spring llamada **Spring MVC** y más precisamente en los Controladores ya que estos últimos definirán los puntos de entrada para las aplicaciones montadas en Spring MVC.

A continuación mostramos un breve ejemplo de la configuración de un servicio web a partir de Spring MVC.

La anotación `@Controller` indica que una clase tendrá el rol de Controlador, Spring no fuerza al desarrollador a extender o implementar interfaces excepto para casos avanzados que así lo requieran. A su vez `@RequestMapping` permite mapear una URL a un cierto método o clase permitiendo anidaciones [47] y posibilita delegar la invocación a distintos métodos en función del pedido HTTP (“GET”, “POST”, “PUT” o “DELETE”) o alguna condición sobre los cabezales u otras opciones de HTTP.

Durante la etapa de inicialización Spring MVC realiza un escaneo de estas anotaciones en los paquetes indicados por el usuario, y registra los controladores encontrados

```
1  @Controller
2  public OKController {
3
4    @Resource("a")
5    public A a;
6
7    @RequestMapping("/do")
8    public String m(){
9      return a.m();
10   }
11 }
```

FIGURA 4.3: Ejemplo básico de MVC configurado a partir de anotaciones.

para luego buscarlos en tiempo de ejecución. En la figura 4.4 se muestra un diagrama del flujo de control y datos que realiza el framework en tiempo de ejecución, las flechas numeradas representan el orden del flujo que surge al procesar un pedido.[6]

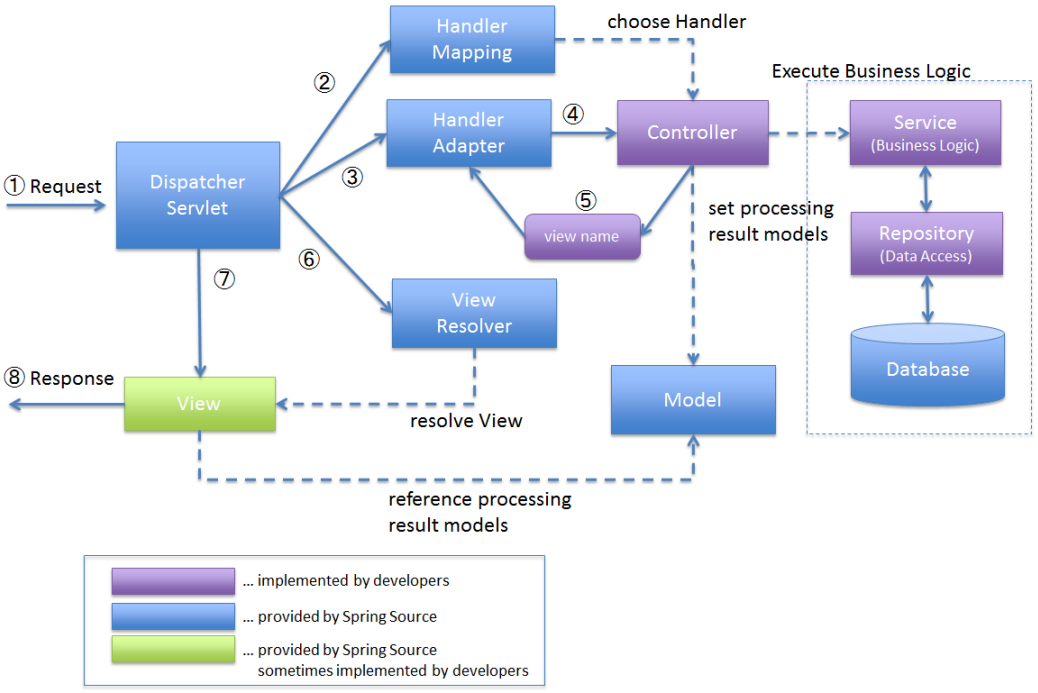


FIGURA 4.4: Gráfico de interacciones de Spring en tiempo de ejecución.

Analicemos entonces como sería la ejecución del ejemplo presentado en la figura 4.3 según el diagrama 4.4:

Todo comenzará con un pedido HTTP dirigido a la URL "/do", Spring recibirá el mismo en su clase Dispatcher Servlet (1). Aquí se consultaran entre los handlers registrados si alguno concuerda con el pedido ingresado, encontrando el Controller previamente

[6]http://terasolunaorg.github.io/guideline/1.0.1.RELEASE/en/Overview/SpringMVCOverview.html

instanciado y registrado (en el ejemplo `OKController`) (2). Luego por medio de un patrón de diseño Adapter (3) se ejecutará el controlador (4), y su resultado, después de las indirecciones correspondientes entre el bean `a` y `b` será "static/index.html" representando el nombre de la vista (5). Spring resolverá por medio de su clase View Resolver la vista por nombre "static/index.html" (6), a su vez en caso de que el Modelo sea alterado por el controlador un motor de *templates* como Velocity o JSP se ejecutará en la vista (7). Para terminar se devolverá la vista al cliente en un objeto de respuesta (8), además dependiendo del pedido HTTP (GET,POST,etc) Spring proveerá capacidades de re-dirección o forwarding entre vistas en caso de que sea requerido (POST exitoso).

Cabe aclarar que en el ejemplo presentado no hay interacción con el modelo, en caso que hubiera, cuando el Handler Adapter ejecuta el controlador (4), el mismo además de retornar el nombre de la vista, recibirá y modificará el modelo por parámetro. Estos datos se cargaran luego por default en una vista dinámica por alguna tecnología de templates como Velocity, JSP o FreeMarker. Comúnmente las tecnologías de visualización cambian y Spring permite adaptar su MVC realizando extensiones al View Resolver.

Como se puede ver en el diagrama el código provisto por la aplicación sobre el framework comienza a partir de los Controladores, y continua su lógica de negocios en otras clases como Services o Repository según lo requiera.

4.2. Desafíos

Habiendo presentado algunas de las características más relevantes a nuestros fines sobre las funcionalidades principales de Spring como Inversión de Control y Modelo-vista-constrolador, podemos ver el uso intensivo de diversas prácticas que engloban buena parte de los desafíos actuales en análisis de llamados presentados en el capítulo 3. Pasaremos a realizar un breve resumen de los mismos.

- Lectura de configuración para la instanciación de beans (IoC). Implica la interacción con el entorno de ejecución como la busqueda en el class path de ciertas clases anotadas y/o la lectura de archivos XML.
- La instanciación de beans (IoC). Implica la ejecución de métodos (setters o constructores) vía reflection, ya que los mismos cambian de firma según cada aplicación y son desconocidos a priori por el framework.
- La inyección de dependencias (IoC). Spring resuelve los lazos entre las entidades que colaboran en runtime en tiempo de ejecución por medio de las configuraciones y reflection.
- La búsqueda y ejecución de métodos de entrada a partir de anotaciones de Controller (MVC). Estos métodos también son buscados e invocados usando reflection.

Además los mismos podrían involucrar polimorfismo a la hora de resolver sus parámetros u objetos destino.

Estas funcionalidades hacen imposible la construcción de un grafo de llamadas *out-of-the-box* con herramientas y algoritmos tradicionales. No obstante, algunos trabajos han surgido proponiendo ideas para lidiar con estos inconvenientes, los dos mas cercanos en esta materia son Frameworks for frameworks o F4F [21] y una técnica de análisis orientada a encontrar vulnerabilidades arquitectónicas de seguridad en servicios web [8].

F4F es parte de la herramienta comercial de IBM Appscan[7] y propone una forma de simular algunos de los comportamientos que vimos en los frameworks con el fin de encontrar vulnerabilidades entendiendo cómo fluyen los datos sensibles para un usuario dentro de un servicio web. La idea más importante de F4F gira alrededor de la definición de un lenguaje de especificación (WAFL) que abstrae ciertas operaciones como invocaciones de llamados, reemplazo de llamados o asignaciones de variables en formularios según el framework a estudiar. WAFL permite la difusión de estos conceptos para diferentes frameworks evitando tener que dar un soporte ad-hoc para cada uno. A nivel implementativo WAFL se traduce en varias clases y un método *main* de Java en donde es posible aplicar análisis y herramientas de tainting conocidas. Sin embargo F4F posee varios puntos no explorados, entre los primeros se encuentra la falta de validación en la precisión del grafo de llamada ya que en el estudio solamente se hace un análisis cuantitativo sobre el crecimiento de tamaño en el grafo y las nuevas vulnerabilidades encontradas pero nunca se contrasta con una ejecución real para entender funcionalidades que podrían escapar al mecanismo. Por otro lado, no es claro cómo F4F resuelve el posible polimorfismo generado a partir de la inyección de dependencias debido a que según lo indica la publicación resuelve la instanciación por medio de inferencia de tipos dentro de los procedimientos, aclarando que posiblemente esto genera una pérdida de precisión pero sin reportar cual es la magnitud de la misma.

Para terminar el código de F4F no es abierto y sólo está disponible comprando una licencia comercial por ende como veremos en las secciones siguientes haremos nuestro propio desarrollo tomando algunos de los conceptos de este trabajo y los someteremos a diferentes validaciones pensadas para explorar las limitaciones de este enfoque mientras se analiza si como el resultado obtenido es relevante para la reconstrucción de arquitecturas.

El segundo trabajo mencionado también tiene como propósito el estudio de falencias de seguridad. A diferencia de F4F, el trabajo de Berger. et al tiene en cuenta la faceta arquitectónica del sistema a la hora de estudiar sus problemáticas no obstante sus vistas requieren el agregado de anotaciones para entender un flujo de datos y mostrar vulnerabilidades asociadas sobre un determinado campo de interés. Este estudio tampoco brinda detalles sobre la precisión de sus análisis sobre los frameworks en los que

[7] http://www-03.ibm.com/software/products/en/appscan-standard

trabaja a pesar de que reconoce la problemática asociada a la imprecisión de los análisis estáticos de código. Por último, a modo de evaluación, el paper contempla las opiniones recibidas por los expertos sobre la re-ingeniería de las aplicaciones analizadas, las diferencias entre los resultados semi-automáticos y los diagramas construidos manualmente, junto a las vulnerabilidades reportadas.

Parte II

Desarrollo

Capítulo 5

Método para la obtención de dependencias de runtime en aplicaciones basadas en Spring

Como fue mencionado en los antecedentes existen historias o vistas en el software relacionadas con el tiempo de ejecución interesantes para el mantenimiento del mismo que no son capturadas por las reconstrucciones arquitectónicas conocidas. En vías de proponer técnicas orientadas a estas problemáticas propondremos un método para conseguir información acerca del flujo de control que será nuestra base para la construcción de las mismas. Realizaremos este análisis para aplicaciones basadas en el framework Spring, debido a su amplia difusión, sus desafíos para los analizadores de código y el interés de su estudio desde una perspectiva estática. La visión global del proceso realizado para la reconstrucción arquitectónica se puede ver en la figura 5.1, en este capítulo ahondaremos en los detalles implementativos de los primeros dos componentes: el analizador de framework y el analizador de código, mientras que en los subsecuentes se abordarán el resto de los mismos.

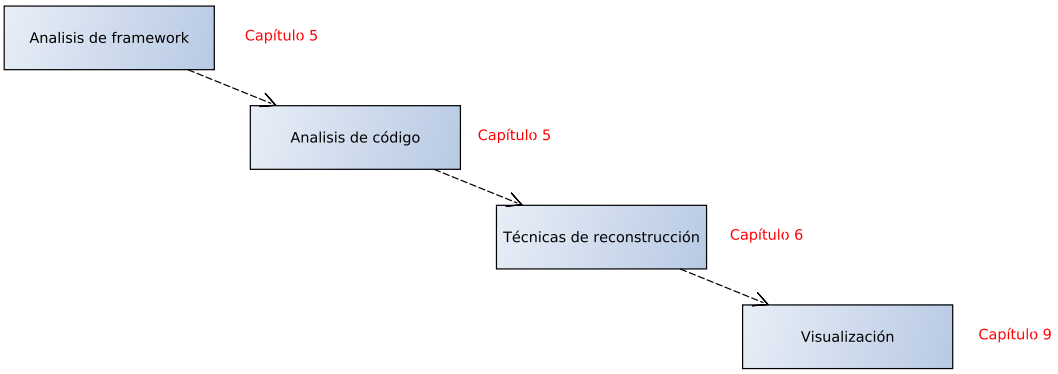


FIGURA 5.1: Workflow de la reconstrucción arquitectónica.

Partiendo de los conceptos introducidos en la sección 4.2 del capítulo anterior, inspiraremos nuestra implementación sobre uno de los aportes de F4F, la generación de un punto de entrada que simule la inicialización y ejecución de un servicio web, llamaremos a este método *dummy main*. Una vez generado (etapa Análisis de framework) podremos, tras una serie de pasos intermedios, construir un grafo de llamadas a partir de herramientas presentadas en 3.2 como WALA, Soot y Doop (etapa Análisis de código). Brindando así una base que permitirá apoyar las técnicas propuestas de reconstrucción arquitectónica (etapa Técnicas de reconstrucción) y finalmente visualizar los resultados.

5.1. Generación del Dummy Main

Para la generación del *dummy main* aprovecharemos la articulación de algunas herramientas de introspección provistas tanto por Spring como por Java (reflection). Si bien esto involucrará levantar *parcialmente* la aplicación, no requerirá la ejecución de la misma y creemos que en un trabajo futuro se podrá cambiar por un mecanismo puramente estático. La razón de este acercamiento es que la magnitud el trabajo necesario para obtener la misma información de forma completamente estática cae fuera del alcance de esta tesis.

El algoritmo para la generación se divide en tres partes: La primer parte consiste en encontrar todos los beans creados por Spring e instanciarlos con su constructor por defecto (sin parámetros). La segunda etapa consulta las dependencias de cada bean ya instanciado y conecta los mismos a través de sus campos públicos y la última etapa busca aquellos beans que cumplen el rol de Controladores (puntos de entrada) e invoca sus operaciones con parámetros simulados. Mostraremos a continuación una representación del pseudocódigo.

Cada una de estas partes involucra el uso de ciertas tecnologías y decisiones para conseguir un balance razonable entre las funcionalidades analizadas y las omitidas, este *tradeoff* es en parte debido a la versatilidad de Spring para su configuración e influirá en la precisión global del método, evaluada durante el capítulo 9.

Algorithm 1 pseudocódigo para generar el Dummy Main

```
1: dummyMain ← genMainClassPrelude()
2: definedBeans ← getDefinedBeans()
3: for each bean in definedBeans do
4:     beanConstructor ← getConstructor(bean)
5:     dummyMain ← dummyMain.append(genInitCode(bean.name, beanConstructor))
6: end for
7: for each bean in definedBeans do
8:     deps ← getDependencies(bean)
9:     for each dep in deps do
10:        field ← getBeanField(bean, dep)
11:        dummyMain ← dummyMain.append(genRefCode(bean, field, dep))
12:    end for
13: end for
14: for each bean in definedBeans do
15:    if isController(bean) then
16:        entryPoints ← getEntryPoints(bean)
17:        for each entryPoint in entryPoints do
18:            dummyMain ← dummyMain.append(genMethodCall(bean.name, entryPoint))
19:        end for
20:    end if
21: end for
```

Para la búsqueda de beans y dependencias (*getDefinedBeans* y *getDependencies* en el pseudocódigo) se usaron las capacidades de introspección que brinda Spring bajo las interfaces `ConfigurableApplicationContext` y `ConfigurableBeanFactory`, en

particular los métodos `getDependenciesForBean`[1] y `getBeansDefinitionsNames`[2] respectivamente. Mientras que para saber si la clase de un bean tiene constructor por defecto o no (*getConstructor*), conseguir el campo público que referencia a otro bean (*getBeanField*), decidir si un bean es Controlador (*isController*) y tomar sus métodos de entrada (*getEntryPoints*) la herramienta fue la reflexión de Java y sus métodos de introspección básicos de `Class`, `Method` y `Field`.

En cuanto a la generación, la misma fue construida sobre un String o `StringBuffer` en donde se fue agregando según las etapas el código correspondiente. Se comenzó generando el preludio con la estructura del método *main* de Java, luego en la instanciación (*getInitCode*) se agregó una línea con un `new` por cada bean. Después en el cableado de dependencias se referenció y asigno a los campos correspondientes los beans ya instanciados y por último en los llamados se generaron invocaciones a los métodos con parámetros asignados de la siguiente manera: para las interfaces propias de los controladores de Spring (`Request`, `Response` y `Model`) un objeto fue creado para cada una, para clases con constructores por defecto (sin parámetros) una nueva instancia fue creada in-situ y en el caso de tipos primitivos se asignaron valores por defecto. Cualquier parámetro que caiga fuera de estas categorías será invocado con *null*, durante las evaluaciones (capítulo 9) se verá como esta última decisión afecta el desempeño de los algoritmos de construcción de grafo de llamadas.

Ejemplo

Tomando el ejemplo de la sección 4.1.2 (figura 4.3), los pasos del algoritmo para analizar esta aplicación serán los siguientes. Inicialmente encontrará tres beans definidos `a`, `b` y el controlador `OKController`. Todos ellos disponen de constructor por defecto (sin parámetros), por lo tanto se generará el siguiente código

```
4                //Intanciacion
5                A a = new A();
6                Bi b = new Bi();
7                OKController okController = new OKController();
```

Luego se consultaran las dependencias de cada uno y se referenciaran o cablearan según los campos correspondientes, generando el código que se ve a continuación

[1] Clase org.springframework.beans.factory.config.ConfigurableBeanFactory
[2] Clase org.springframework.beans.factory.ListableBeanFactory

```
8               //interconexion
9               a.bfield = b;
10              okController.a = a;
```

Por último se generará el código para invocar los métodos discriminados como puntos de entrada. Para Spring MVC esto quiere decir, como vimos en 4.3, los métodos anotados con `@RequestMapping`.

```
11              //invocacion
12              okController.m();
```

```
1public class DummyMain {
2
3       public static void main(String[] args){
4               //Intanciacion
5               A a = new A();
6               Bi b = new Bi();
7               OKController okController = new OKController();
8               //interconexion
9               a.bfield = b;
10              okController.a = a;
11              //invocacion
12              okController.m();
13      }
14}
```

FIGURA 5.2: Método y clase *dummy main* completo generado para el ejemplo 4.3.

Es importante notar que a los fines prácticos no importa el orden de las invocaciones y las intanciaciones en el código generado ya que el análisis de llamadas realizado posteriormente buscará entender todas las posibles ejecuciones del programa.

Para terminar las tareas realizadas en el paso correspondiente al análisis del framework, también se expone el grafo de dependencias entre beans con los datos referidos al nombre de cada uno y su clase concreta. Siguiendo con el ejemplo 4.3.

```
digraph "DirectedGraph" {
        "okController:OKController" -> "a:A";
        "a:A" -> "b:Bi";
}
```

FIGURA 5.3: Grafo de dependencias de Spring en formato Graphviz.

5.1.1. Limitaciones conocidas

En cuanto a las limitaciones de nuestro análisis, Spring permite la configuración de dependencias en un bean en campos públicos, protegidos o privados sin ninguna distinción, esto trae nuestra primera dificultad ya que para nuestro *main* generado solamente se podría realizar estas inyecciones por medio de reflexión, que es justamente lo que tratamos de evitar, el análisis de código reflexivo. Por lo tanto, realizaremos la inyección solamente para los campos públicos, esto no presentará diferencias a los fines prácticos de revelar las interacciones propias de la capa de negocios de la aplicación, distinto sería el caso si nuestra investigación estuviera orientada a problemáticas de seguridad o visibilidad de la información.

Restringiremos además la instanciación de Beans a aquellos que presenten un constructor por defecto (caso más común), ya que el análisis requerido para integrar los casos donde la inyección se produce por parámetros cae fuera del alcance del trabajo.

Además dentro de los casos que decidimos no soportar se encuentran usos avanzados de Spring donde un bean inspeccionara el contexto de ejecución en busca de otro, es decir no explicita su dependencia en forma estática mediante un archivo de configuración sino que consulta en tiempo de corrida por un nombre o interfaz en el registro. Este uso de la interfaz de Spring permite interacciones entre servicios en forma parecida a callbacks de librerías pero con un grado mayor de indirección. Funcionalidades de esta índole no están contempladas dentro de la especificación de F4F o ningún otro reporte y podrían ser motivo de técnicas nuevas de análisis.

5.2. Construcción del grafo de llamadas

Una vez generado el *dummy main*, se avanzará en la construcción del grafo de llamadas según el workflow presentado en la siguiente figura.

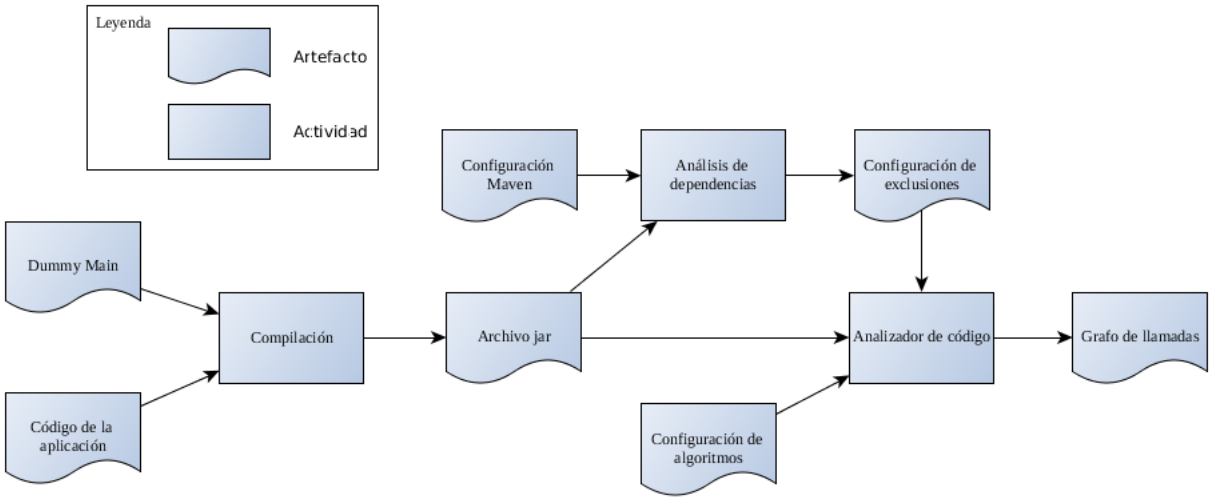


FIGURA 5.4: Workflow para la construcción de grafo de llamadas.

El primer paso será compilar el código generado junto al resto del código de la aplicacion, ya que los analizadores de código que utilizaremos trabajan a partir de bytecode. A partir de allí la entrada del analizador de código constará de tres partes, el código binario de la aplicación (Archivo jar en la figura), las dependencias a excluir y la configuración de que algoritmo ejecutar.

Como fue mencionado en los antecedentes (3.2) tanto las herramientas de análisis ofrecen funcionalidades para incluir o excluir clases y/o paquetes, sin embargo las aplicaciones modernas compiladas por sistemas como Ant o Maven disponen de una gran cantidad de dependencias especificadas como artefactos (archivos `.jar`), cada uno de estos archivos puede contener una cantidad variable de paquetes y de diferentes nombres por lo que se necesito de una actividad para entender a partir de un artefacto los paquetes contenidos en el mismo. Luego una configuración acerca de que artefactos y algoritmo se desea correr deben ser proporcionadas al analizador que retornara finalmente un grafo de llamadas.

Según la cantidad de librerías o paquetes incluidos en el análisis se espera que su desempeño y resultado se vea afectado ya que varias funcionalidades provistas por las clases de la aplicación requieren de sus dependencias para ser analizadas (i.e., extienden una clase, implementan una interfaz, etc), las preguntas relacionadas con este tema se plantearan oportunamente en las evaluaciones del trabajo.

Realizaremos la construcción del grafo de llamadas mediante dos de las herramientas mencionadas en los antecedentes (3.2) Soot y Wala. Doop no será utilizada debido a su restricción de licencias.

Soot y Wala difieren considerablemente en su arquitectura y modificabilidad por lo que su uso implicó dos implementaciones diferentes. Soot provee un sistema de workflow

en donde uno puede registrar componentes de análisis, en particular durante una fase llamada *whole-program* y luego mediante un call back el mismo retorna al usuario el grafo de llamadas, por otro lado Wala brinda un protocolo de objetos o API que entre otras cosas pueden ser usados para la construcción de un grafo de llamadas.

Una vez construido el grafo de llamadas bajo los distintos algoritmos de cada herramienta el mismo será exportado en formato `dot` para su visualización, análisis y utilización por técnicas de reconstrucción.

Ejemplo

Partiendo del *dummy main* generado en la sección anterior (figura 4.3), dado que el mismo no contiene ningún impedimento como los mencionados los grafos construidos a partir de Soot y Wala, y sus algoritmos no diferirán en su resultado (figura 5.5). Para simplificar la visualización cada nodo solamente muestra su clase y nombre del método.

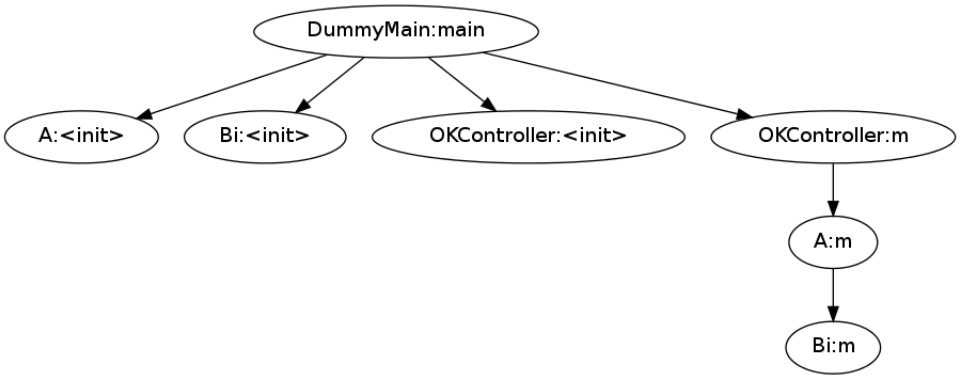


FIGURA 5.5: Grafo de llamadas construido a partir del ejemplo 4.3.

Como se puede ver el grafo contempla la línea de llamados `OKController.m() ->A.m() ->B.m()` así como la inicialización de los beans. Para ilustrar el impacto que podría tener la exclusión de dependencias supongamos que el bean `a` implementa alguna interfaz cuyo paquete se excluye durante el análisis, de este modo tanto el eje de invocación hacia `A:m`, `Bi:m` y `A:init` no serian representados en el grafo resultante (figura 5.6) resultando en una pérdida considerable de precisión. Sin embargo, dicha perdida podría acelerar la construcción del grafo por lo que será considerada como una variable en las evaluaciones propias de la precisión (capítulo 9).

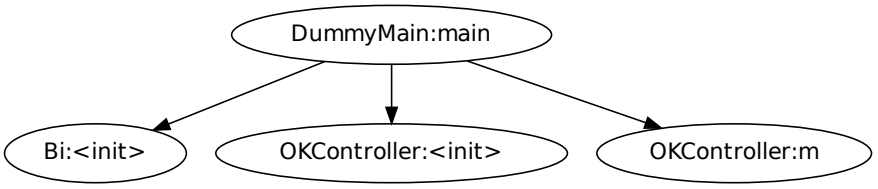


FIGURA 5.6: Grafo de llamadas con exclusión de interfaz implementada por el bean `a`.

Capítulo 6

Técnicas de reconstrucción basadas en información del framework

Una vez conseguida la información del framework y con ella el grafo de llamadas se comenzó en la construcción y modificación de técnicas de reconstrucción a partir de los mismos. Para la construcción de técnicas nuevas contemplamos los dos grandes enfoques mencionados en (2.1) (estructural y de información léxica) y ofrecimos una propuesta para cada uno. Según la técnica la información obtenida por el método propuesto en el capítulo 5 se deberá integrar al flujo de cada una.

6.1. Propuesta estructural

Presentaremos en esta sección los detalles de la propuesta estructural, la misma se apoya fundamentalmente sobre la clusterización del grafo de llamadas. Sin embargo el mismo debe ser procesado cuidadosamente debido a ciertas características de recall del grafo de llamadas y completitud en las reconstrucciones que detallaremos a continuación.

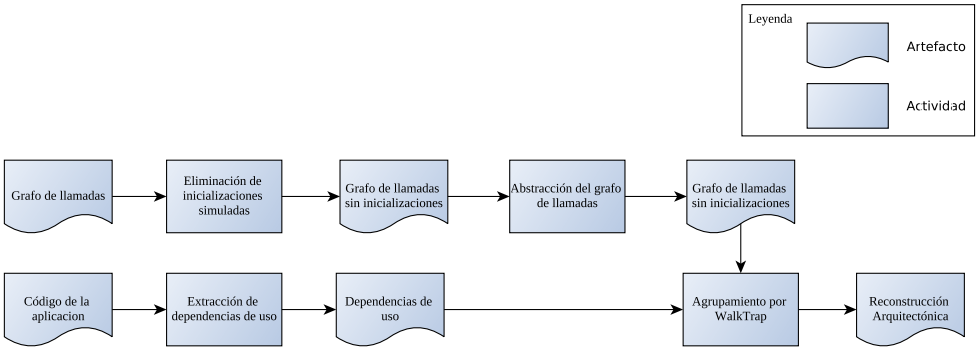


FIGURA 6.1: Worflow de la técnica estructural.

El procesamiento de la técnica comienza a partir de la eliminación de inicializaciones del grafo, las mismas son producto de la simulación de la puesta en marcha del sistema explicada en la sección 5.1 y como ilustraremos luego en un ejemplo, solamente aportan información propia del tiempo de diseño, cuya influencia en las reconstrucciones intentamos disminuir.

Por otro lado, debido a que el resultado buscado es una descomposición de entidades (clases) se necesitará una traducción o abstracción de los nodos del grafo de llamadas (métodos) para ello tomaremos los ejes `A.a() ->B.b()` (el método `a` invoca al método `b`) y los condensaremos en un eje `A ->B` asignándole un peso entero según la cantidad de métodos de `A` que invocan a métodos de `B`.

Como hemos mencionado oportunamente en los antecedentes del trabajo, los grafo de llamadas obtenidos podrían no ser *sound* debido a características tecnológicas (reflection), exclusión de dependencias (sección 5.2) y limitaciones en la generación del *dummy main* (sección 5.1.1) esto muy probablemente resulte en que las entidades y relaciones encontradas en un grafo de llamadas no sean completas respecto al sistema (recall), desembocando en reconstrucciones parciales del mismo. En el afán de mantener nuestra línea de investigación sobre reconstrucciones completas del sistema sera necesario añadir las dependencias de compilación para contemplar aquellas entidades que no se fueron reconocidas en el grafo de llamadas por las limitaciones mencionadas. No obstante, esta adición se realizará contemplando las relaciones de diseño como de menor prioridad frente a interacciones de llamados.

La última actividad de esta técnica consiste en la ejecución de la etapa de clustering, como fue mencionado en la sección 2.2.4 utilizaremos el algoritmo Walktrap, implementado por la librería igraph [1] en python. Esta implementación brinda la capacidad de ejecutar el mismo sobre un grafo con pesos cuyo uso adoptaremos para dar más prioridad a las dependencias producto del grafo de llamadas, y menor prioridad a las dependencias de uso.

[1] http://igraph.org/

La hipótesis es que esta técnica construirá agrupamientos más relacionados con una vista de runtime y simplificará el seguimiento de ejecuciones reales abstrayendo componentes que tienen mayor interacción entre si pese a que en tiempo de diseño su relación no sea directa.

Ejemplo

Presentaremos un ejemplo sintético a los fines de ilustrar la técnica propuesta y mostrar las diferencias con las técnicas actuales. En la figura 6.2 se muestra un esquema tradicional de clases, las mismas son: `A,B,C,D,E,F` y la interfaz `I`, las flechas con puntos representan herencia mientras que las flechas normales representan uso, este esquema constituye lo que hemos llamado a lo largo del trabajo como dependencias de tiempo de compilación. La idea detrás del diseño en estas clases se acerca al patrón composite definido en [23], donde al compartir una interfaz común las implementaciones concretas pueden ir componiéndose para lograr un objetivo común.

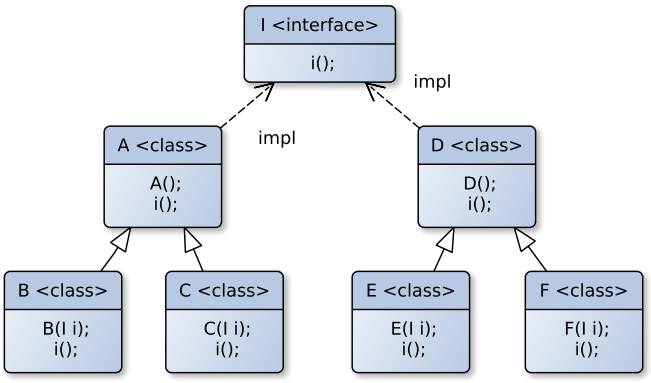


FIGURA 6.2: Diagrama de clases del ejemplo mencionado.

```
1  interface I{
2      void i();
3  }
4  class A implements I {
5      public I i;
6      public void i() {}
7  };
8  class B extends A {
9      public B(I i) {this.i = i;}
10     public void i() {this.i.i();}
11 }
12 class C extends A{
13     public C(I i) {this.i = i;}
14     public void i() {this.i.i();}
15 }
16 class D implements I {
17     public I i;
18     public void i() {}
19 }
20 class E extends D {
21     public E(I i) {this.i = i;}
22     public void i() {this.i.i();}
23 }
24 class F extends D {
25     public F(I i) {this.i = i;}
26     public void i() {this.i.i();}
27 }
28
29 public class Main{
30     public static void main(String[] args) {
31         C c = new C(new B(new D()));
32         c.i();
33         F f = new F(new E(new A()));
34         f.i();
35     }
36 }
```

FIGURA 6.3: Código fuente del ejemplo mencionado.

Por otro lado, en la figura 6.4 vemos el grafo de llamadas construido para el ejemplo, en el se pueden observar dos grupos bien definidos: la inicialización de las

clases (método `<init>`) y la ejecución de los llamados (método `i`). Si prestamos atención a la inicialización veremos que el grafo que se desprende de ellas es bastante similar al UML, exceptuando la interfaz `I` que no juega ningún rol en el camino de ejecución, esto aparecerá notoriamente en el grafo de llamadas construido a partir del *dummy main* ya que toda la simulación correspondiente a levantar el contexto de Spring se basa en instanciaciones. Por ello, decidimos remover de la técnica las inicializaciones producidas en el `Main` como se ve en la figura 6.5.

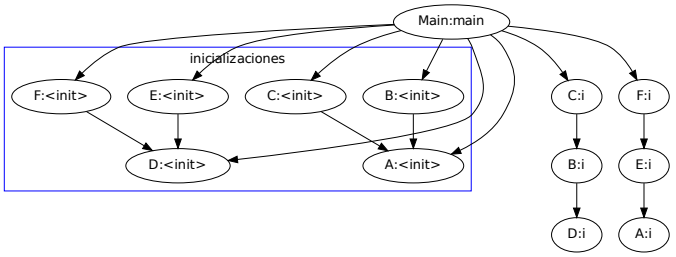


FIGURA 6.4: Grafo de llamadas del ejemplo mencionado.

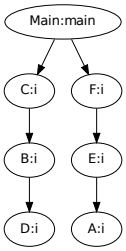


FIGURA 6.5: Grafo de llamadas luego de la eliminación de inicializaciones.

Habiendo realizado este filtrado procedemos a la etapa de abstracción del grafo de llamadas, el orden de estos pasos es fundamental ya que al traducir el grafo se vuelven indistinguibles los llamados de inicialización (`<init>`). El resultado de la abstracción en el ejemplo será el siguiente. Para simplificar el ejemplo no adicionaremos los ejes provenientes de las dependencias de uso, ya que por su baja prioridad no afectan el resultado de la técnica.

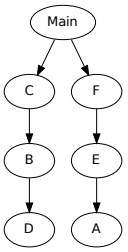


FIGURA 6.6: Abstracción del grafo de llamadas.

Finalmente, después de este procesamiento ejecutaremos el algoritmo Walktrap, su resultado se puede apreciar en la figura 6.7, separando dos comunidades participantes en la ejecución.

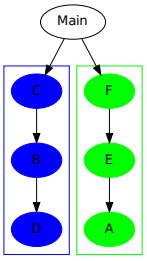


FIGURA 6.7: Grafo de llamadas colapsado y clusterizado por walktrap.

En contraste, el resultado para este mismo ejemplo de otras técnicas estructurales basadas solamente en dependencias de uso como ACDC o Bunch da como resultado un único componente sin diferenciar las dos ramas de ejecución, ya que desde un punto de vista modular todas las clases implementan la misma interfaz.

6.2. Propuesta basada en información léxica

La segunda técnica que propondremos se sitúa fuertemente sobre los conceptos de la técnica ARC (mencionada en 2.2.3), en ella se combinan elementos de información léxica (una clase se considera como una bolsa de palabras) con elementos de la estructura del programa en tiempo de compilación. Dentro del procesamiento de la información léxica se encuentra el método estadístico LDA, este método crea conjuntos de palabras relacionadas o *tópicos* a partir de la frecuencia de las mismas, que luego pueden ser usados para la construcción de agrupamientos clasificando según la probabilidad de pertenencia a cada conjunto. Para la implementación de modelado de tópicos se utilizó la

librería `MALLET`[2], que provee varios algoritmos relacionados con: document classification, sequence tagging y topic modelling. A continuación presentamos el workflow propuesto para la técnica.

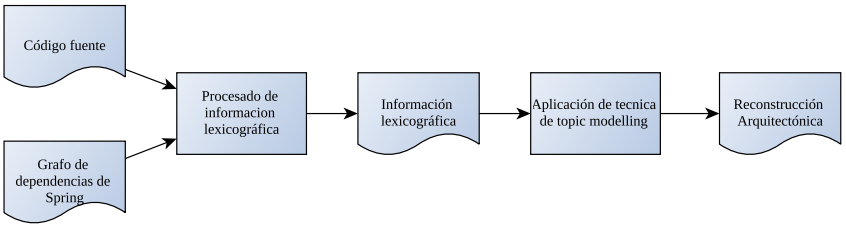


FIGURA 6.8: Worflow de la técnica basada en información léxica.

La propuesta comenzará entonces combinando la información contenida en las clases (código fuente) con la información del grafo de dependencias de Spring obtenida por nuestro análisis en la sección 5.1. La idea es intentar de sobreponer la estructura de las dependencias entre beans en tiempo de ejecución con el análisis léxico con la intención de mejorar el resultado en casos donde topic modelling por si solo pierde precisión. Esto se logrará en la primera actividad de la siguiente manera: si el bean `b`, de clase `B` depende del bean (utilizando la definición de Spring de dependencia, sección 4.1.1) `a` de clase `A`, entonces en la información textual de `B` se agregará el nombre y clase del bean `a`[3]. Además, la importancia que tendrá esta información del framework será un parámetro en la actividad de procesamiento.

Existen varias razones que motivan este acercamiento, la primera radica en que la relación mantenida entre los beans usualmente está ligada a la consecución de su fin, y que cada bean se encarga de un propósito específico, por lo tanto los beans dependientes tienen inherentemente una relación en la realización del objetivo de negocio. Por otro lado, es una práctica habitual el nombrado descriptivo de beans como actores de runtime, por lo que sus nombres probablemente sean valiosos a la hora de un análisis de información léxica.

Continuando en esta misma etapa se deberán filtrar palabras propias del lenguaje de programación Java, palabras comunes del idioma inglés, nombres de autores (presentes en comentarios) y otras palabras tecnológicas como nombres de paquetes, anotaciones, etc. Para terminar, en la última etapa se ejecutará el algoritmo de topic modelling (LDA) provisto por MALLET con la información léxica procesada como parámetro y la cantidad de tópicos o clusters deseada.

La hipótesis es que esta técnica servirá para discriminar casos donde la estructura en tiempo de ejecución por razones de implementación o de la abstracción elegida para

[2]http://mallet.cs.umass.edu/
[3]Los textos escritos en *camel-case* se traducirán en espacios apropiadamente

las reconstrucciones (clases) mantiene diferencias respecto a la información textual del dominio de las clases.

Ejemplo

Supongamos que el sistema bajo análisis se encarga de monitorear condiciones climáticas, para ello cuenta con dos componentes (beans) un medidor atmosférico y un medidor de precipitaciones. A su vez, cada uno contiene diferentes instrumentos que se ejecutan para obtener una medición completa. Mientras el atmosférico utiliza el barómetro, el piranómetro y la veleta, el medidor de precipitaciones hace uso del higrómetro, pluviómetro y la almohadilla de granizo. Ya que el comportamiento de cada medidor es a priori análogo (interactúa con sus instrumentos en algún orden y luego devuelve un resultado) una implementación de este sistema podría implicar que ambos medidores sean instancias distintas de una misma clase. Por otro lado cada instrumento contiene su propia lógica y podrían encapsularse en clases diferentes. Esto llevaría al siguiente diagrama de clases y de objetos.

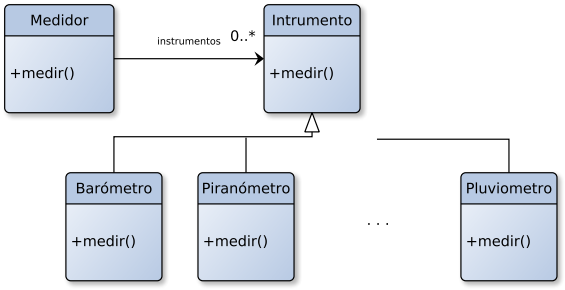


FIGURA 6.9: Diagrama de clases del ejemplo mencionado.

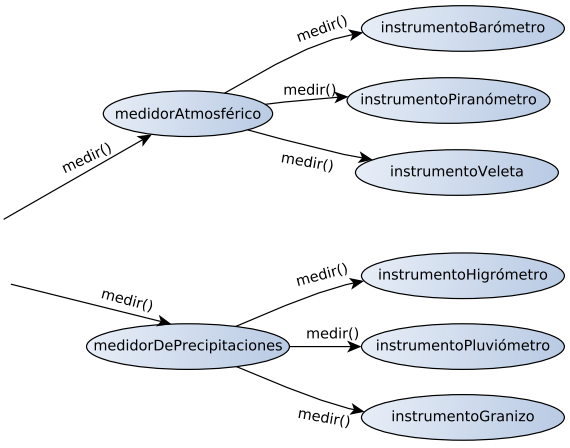


FIGURA 6.10: Diagrama de objetos del ejemplo mencionado.

Este escenario plantea un desafío a la abstracción propuesta (clases vs. objetos) para la reconstrucción arquitectónica ya que ambos medidores serán abstraídos por su clase aunque sus dos instancias tengan propósitos diferentes, además sus llamados hacia los instrumentos serán combinados (ver figura 6.11) por lo que probablemente todos los instrumentos sean abstraídos en el mismo agrupamiento. Desde una perspectiva de interacción, esto a pesar de ser correcto no es preciso en cuanto a los dominios de negocio de la aplicación, que podrían ser de interés en una reconstrucción.

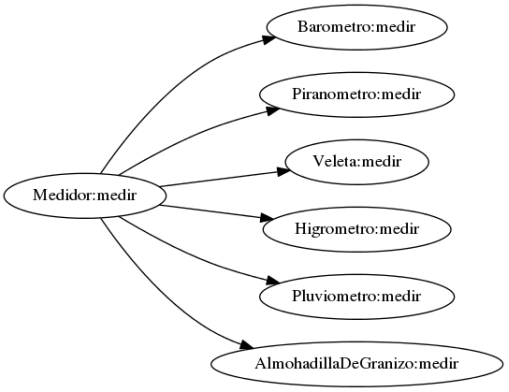


FIGURA 6.11: Grafo de llamadas del sistema climático.

Finalmente el resultado de la agrupación por la técnica propuesta sera el exhibido en la figura 6.12, notar que además de diferenciar los dos agrupamientos, se recuadra ademas una parte del tópico de cada uno. Cabe destacar que en nuestro ejemplo las interacciones propias del diagrama de objetos serian las mismas en el diagrama de dependencias de Spring, esto podría no ser el caso general, sin embargo nuestra hipótesis es que ambos están relacionados.

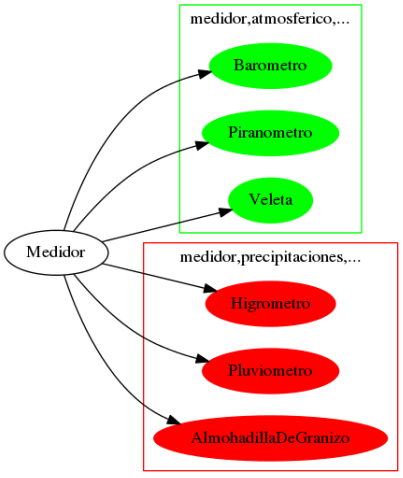


FIGURA 6.12: Agrupamientos para el sistema climático con técnica de tópicos.

Si bien este ejemplo cuestiona la abstracción de clases para mostrar comportamientos del sistema, proponer un cambio en ese sentido caería fuera del alcance del

trabajo por lo que nos concentraremos en mejorar esta misma vista de clases. Sin embargo es imposible no plantearse distintos caminos que serían posibles para abordar esta problemática en un eventual trabajo futuro, como puede ser la utilización de contextos (como los presentados en 3.1) para diferenciar clases, o la eventual representación de beans en vez de clases para casos similares al del ejemplo.

Evaluaremos durante el capítulo 10 el desempeño de la técnicas propuestas y las técnicas relevantes halladas en la literatura.

Parte III

Resultados

Capítulo 7

Preguntas

Presentaremos en esta parte de la tesis las preguntas que surgen a partir del trabajo realizado y las evaluaciones conducidas en el afán de buscar respuestas a las mismas. La primer pregunta de interés surge a partir del método propuesto en el capítulo 5 y las características y desafíos mencionados en los antecedentes vinculados a los análisis de código, allí se indicaron limitaciones conocidas del método propuesto en cuanto al análisis del framework como también diferencias entre los algoritmos disponibles para la construcción del grafo de llamadas. Esto abre la puerta al primer interrogante del trabajo.

- **RQ(1)** ¿Cual es la precisión y el recall del método propuesto en el capítulo 5 para la construcción del grafo de llamadas?

Nuestra hipótesis es que la precisión aumentará según aumente la sensibilidad del algoritmo de grafo de llamadas utilizado, pero que en general se vea amenazada por características tecnológicas (reflection) y limitaciones propias del análisis del framework (casos no soportados).

Por otro lado, a partir de las técnicas propuestas en la sección 2 y los antecedentes vinculados a reconstrucción arquitectónica creemos de interés preguntarse acerca de la fidelidad que pueden tener los agrupamientos obtenidos por diferentes técnicas (conocidas y novedosas) con una vista arquitectónica que refleje interacciones propias del tiempo de ejecución, diferenciando claramente unidades de colaboración (definidos en la introducción como componentes) que tengan propósitos distintos. La pregunta de alto nivel en este sentido es la siguiente.

- **RQ(2)** ¿Es posible obtener agrupamientos que den valor en la reconstrucción y visualización de comportamientos arquitectónicos en tiempo de ejecución a partir de las técnicas propuestas en el capítulo 6?

Nuestra hipótesis en esta pregunta yace íntimamente relacionada con el objetivo del trabajo, y consiste en pensar que agrupamientos basados en dependencias de flujo de control más o menos precisas probablemente ayuden a reconstruir componentes con vínculos de tiempo de ejecución.

A su vez estas dos preguntas podrían tener un punto de intersección ya que el resultado de las técnicas apoyadas en grafos de llamada posiblemente varíe de acuerdo a la precisión del método de obtención de dependencias, brindando mayor o menor valor a la descomposición obtenida. La probabilidad de este fenómeno da lugar a la tercer y última pregunta estudiada en este trabajo.

- **RQ(3)** ¿Existe una relación entre la precisión del método propuesto en el capítulo 5 y el valor de las técnicas apoyadas en la reconstrucción y visualización de comportamientos arquitectónicos presentadas en el capítulo 6?

En este caso nuestra hipótesis estará dada por la intuición de que cuanto más preciso es el grafo de llamadas más cercano será el agrupamiento a una representación afín de la vista arquitectónica buscada.

En los capítulos siguientes (9 y 10) presentaremos los refinamientos y estrategias para intentar resolver estas preguntas, para ello conduciremos la evaluación sobre un caso de estudio real (capítulo 8) en el afán de obtener evidencia empírica significativa, e ideas y preguntas que lleven a nuevas vistas y estudios comparativos.

Capítulo 8

Caso de estudio

8.1. Broadleaf - DemoSite

El caso de estudio sobre el que se trabajará durante las evaluaciones es un proyecto de código abierto llamado BroadleafCommerce[1][2] el mismo se presenta como una librería orientada a facilitar la construcción de servicios web *e-commerce* con varios casos de aplicaciones en entornos reales. Sus funcionalidades apuntan a resolver las problemáticas comunes en el dominio de los *e-commerce* y brindar extensibilidad para la adopción de este sistema en diversos ámbitos. Existen en esta línea varios productos comerciales con el mismo fin como Shopify[3] y Bigcommerce[4] pero estos no exponen públicamente su código.

Entre las funciones que ofrece BroadleafCommerce se encuentran el manejo de un carro de compras, proceso de checkout, herramientas de búsqueda, distintas opciones para soportar sistemas de pago, un módulo de ofertas, descuentos y promociones, un subsistema manejador de productos del catálogo, un workflow modificable de aprobación de cambios en productos, capacidades para múltiples usuarios y pedidos, un subsistema de tracking para órdenes y extensibilidad para modificar las entidades o funciones del sistema como se lo requiera.

Tecnológicamente el mismo está programado en el lenguaje Java 7 y se apoya sobre cinco frameworks, estos son:

- **Spring Framework (versión 4.1.6).** Proveé la infraestructura base de los componentes del sistema tales como su inicialización (IoC), el manejo de requests (MVC) y es la base de su modificabilidad por terceros. Además para funcionalidades más avanzadas

[1] http://www.broadleafcommerce.com/
[2] https://github.com/BroadleafCommerce/BroadleafCommerce
[3] https://www.shopify.com
[4] https://www.bigcommerce.com

- **Spring Security (versión 3.2.6)**. Utilizado para la autenticación y autorización de los usuarios del sistema.
- **Hibernate (versión 4.1.11).** Encargado de la interfaz para la persistencia de datos.
- **Thymeleaf (versión 2.1.4).** Responsable del motor de vistas, recomendado por Broadleaf para brindar contenidos en HTML5.
- **SOLR (versión 4.10.3).** Motor de búsqueda de apache usado por BroadleafCommerce.

El proyecto se constituye por aproximadamente dos mil clases Java que aportan alrededor de trescientas mil líneas de código ó 300 KLOC junto a por lo menos cien archivos de configuración XML. Creemos que todas estas cualidades convierten a BroadleafCommerce en un buen exponente de los sistemas que buscamos estudiar en este trabajo.

No obstante debemos tener en cuenta que BroadleafCommerce no representa un sistema sino una librería, por ende utilizaremos para los experimentos su aplicación de prueba DemoSite[5][6] donde se hacen uso de varias de sus funcionalidades para poder estudiar y reproducir escenarios del sistema. Las reconstrucciones entonces consideraran a la aplicación como ambos proyectos juntos DemoSite y BroadleafCommerce.

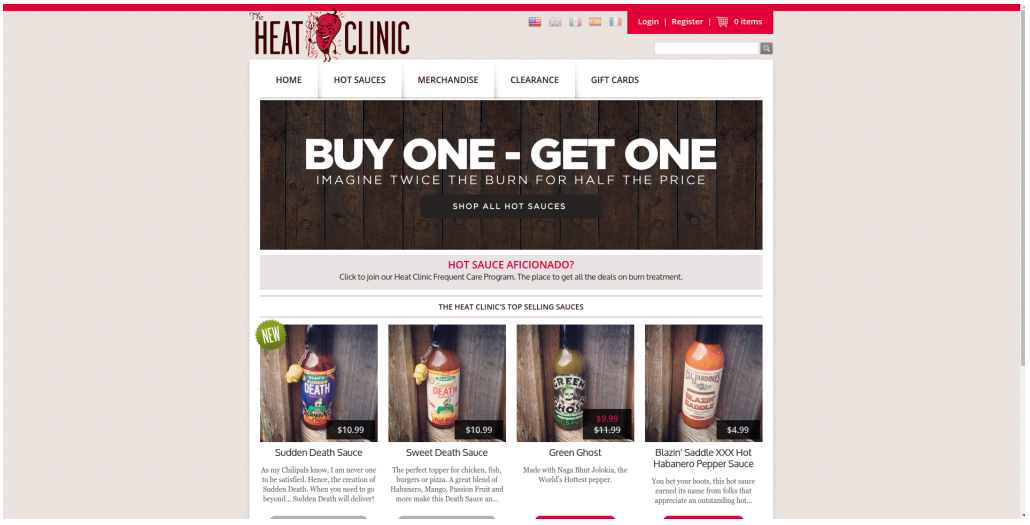


FIGURA 8.1: Aplicación de estudio construida sobre BroadleafCommerce (portal de compras).

Se pueden ver en la figura 8.1 capturas de pantalla de la aplicación de prueba, esta tiene como nombre Heat Clinic y su dominio para la demostración gira en torno a la compra-venta de salsas picantes.

[5]https://demo.broadleafcommerce.org
[6]https://github.com/BroadleafCommerce/DemoSite

Broadleaf y DemoSite ofrecen además del portal principal una consola de administración pero por una cuestión de alcance nos limitaremos a estudiar solamente los comportamientos propios del portal principal.

8.1.1. Casos de uso

Dentro de las funcionalidades que provee la aplicación tomaremos algunos casos de uso representativos del mismo, en cuanto a su comportamiento en tiempo de ejecución.

El primer caso de uso elegido se ejercita por un usuario externo (cliente) y consiste en el ingreso del mismo en el sistema, log-in y edición de su perfil. Denominaremos a este caso de uso como **caso-login**.

1. El usuario hace click en “registrarse”
2. Ingresa los datos de registro (email,nombre y password)
3. El usuario hace click en “submit”
4. El usuario ingresa en el sistema con las credenciales y el usuario ingresados previamente
5. El usuario registrado navega las secciones de edición de perfil
6. El usuario registrado modifica su nombre y su contraseña
7. El usuario hace click en *logout*

El segundo caso de uso elegido trata de un procesamiento de orden involucrando operaciones alrededor del carrito de compras, lo llamaremos **caso-checkout**.

1. Un usuario no registrado hace click en ofertas
2. Aplica filtros relacionados con la marca de la salsa
3. Ordena las publicaciones por precio
4. Entra en el detalle de una publicación
5. La agrega al carrito de compras
6. Entra al carrito
7. Modifica la cantidad del producto y procede al checkout
8. Se loguea como invitado e Ingresa los datos de facturación y pagos
9. Termina el checkout

Como tercer caso de uso tomaremos el uso de la herramienta de búsqueda y filtros para encontrar un item y hacer un review, este se referenciará como **caso-review**.

1. Un usuario no registrado ingresa un texto en la barra de búsqueda
2. Hace un ordenamiento por nombre
3. Ingresa a los detalles de un item
4. Elige la opcion de ingresar para hacer un review

5. Ingresa un rating y el texto del review

Por último dedicaremos un caso en particular para el uso propio de la wishlist y su interacción en el proceso de compra, este será nombrado como **caso-wishlist**.

1. Un usuario registrado elige un producto desde la pagina inicial
2. Lo agrega a su wishlist
3. Luego elige otro producto y lo agrega a su wishlist
4. Ingresa a su wishlist
5. Remueve el primer item de la lista
6. Mueve el segundo item al carrito

Además consideraremos el caso de uso que sale de la combinación de todos los anteriores como **caso-completo**, el mismo será de utilidad para retratar la completitud de estos casos en el ejercicio de las funcionalidades provistas por la aplicación.

8.1.2. Workflows

Una funcionalidad muy importante para la aplicación es la posibilidad de modificar fácilmente los *workflows* principales de la misma a saber: agregar un producto al carrito, actualizar la cantidad de un producto, remover un producto del carrito, cargar una orden y procesar un pago. Si bien este atributo está relacionado generalmente con el sistema desde una perspectiva estática, en este caso resulta de particular interés ya que estos workflows definirán componentes o actores que tendrán roles claros y determinantes en tiempo de ejecución.

Como se puede ver en la documentación de Broadleaf[7] en si los workflows no son muy interesantes con respecto a su estructura, ya que se componen tan solo de una lista de actividades, un gestor de errores y un contexto que se va pasando entre las actividades según se vayan ejecutando. Su interfaz de ejecución es prácticamente el método `doActivities(Context)` que inicia el procesamiento de las actividades bajo el contexto inicial dado. Sin embargo como es posible apreciar en la configuración (figura 8.2) línea 8, el workflow encargado de agregar un item al carrito implementativamente consiste en una instancia de la clase `SequenceProcessor`, esta decisión de implementación es común a todos los demás workflows (i.e., son todos instancias diferentes de la misma clase) y por ende componen un caso similar al presentado como ejemplo en la sección 6.2.

[7] http://www.broadleafcommerce.com/docs/core/current/broadleaf-concepts/workflows-and-activities

```
1  <bean p:order="1000" id="blValidateAddRequestActivity" class="org.broadleafcommerce...
     ValidateAddRequestActivity"/>
2   <bean p:order="2000" id="blCheckAddAvailabilityActivity" class="org.broadleafcommerce...
     CheckAvailabilityActivity"/>
3   <bean p:order="3000" id="blAddOrderItemActivity" class="org.broadleafcommerce...
     AddOrderItemActivity"/>
4   <bean p:order="4000" id="blAddFulfillmentGroupItemActivity" class="org.
     broadleafcommerce...AddFulfillmentGroupItemActivity"/>
5   <bean p:order="5000" id="blAddWorkflowPriceOrderIfNecessaryActivity" class="org.
     broadleafcommerce...PriceOrderIfNecessaryActivity"/>
6   <bean p:order="6000" id="blAddWorkflowVerifyFulfillmentGroupItemsActivity" class="org.
     broadleafcommerce...VerifyFulfillmentGroupItemsActivity"/>
7
8   <bean id="blAddItemWorkflow" class="org.broadleafcommerce.core.workflow.
     SequenceProcessor">
9     <property name="processContextFactory">
10       <bean class="org.broadleafcommerce.core.order.service.workflow.
     CartOperationProcessContextFactory"/>
11    </property>
12    <property name="activities">
13       <list>
14          <ref bean="blValidateAddRequestActivity" />
15          <ref bean="blCheckAddAvailabilityActivity" />
16          <ref bean="blAddOrderItemActivity" />
17          <ref bean="blAddFulfillmentGroupItemActivity" />
18          <ref bean="blAddWorkflowPriceOrderIfNecessaryActivity" />
19          <ref bean="blAddWorkflowVerifyFulfillmentGroupItemsActivity" />
20       </list>
21    </property>
22    <property name="defaultErrorHandler">
23       <bean class="org.broadleafcommerce.core.workflow.DefaultErrorHandler">
24          <property name="unloggedExceptionClasses">
25             <list>
26                <value>org.broadleafcommerce.core.inventory.service.
     InventoryUnavailableException</value>
27             </list>
28          </property>
29       </bean>
30    </property>
31  </bean>
```

FIGURA 8.2: Configuración del workflow que agrega un item al carrito de compras.

Capítulo 9

Evaluación del método de obtención de dependencias

Volviendo sobre la pregunta **RQ(1)** planteada anteriormente, indagaremos en este capítulo las ramificaciones de la misma estudiando la precisión y el recall del método propuesto en el capítulo 9.

Dividiremos este capítulo en tres evaluaciones que creemos darán una idea bastante certera de las posibilidades y precisión del método. Inicialmente analizaremos cuales son los algoritmos de grafo de llamadas factibles de ser ejecutados sobre la aplicación descripta en el capítulo 8 ya que por las razones mencionadas en los antecedentes del trabajo (sección 3.3), el desempeño de los algoritmos y su terminación suele depender de diversos factores como tamaño de la aplicación, librerías y sensibilidad a contexto.

Luego validaremos los resultados de cada grafo de llamadas disponible mostrando su precisión y recall estimado, esta validación representa un problema abierto común en la literatura [42] ya que no es posible para el caso general calcular un grafo de llamadas de referencia, sin embargo haremos uso de una estrategia común en este tipo de situaciones que consiste en la comparación del grafo de llamadas contra la información de flujo obtenida a partir de ejecuciones reales.

Por último evaluaremos el impacto de la precisión del análisis del framework en la construcción del grafo y las limitaciones encontradas a partir de la técnica asociada a la generación del *dummy main*.

9.1. Desempeño de los algoritmos para construcción de grafo de llamadas

La primer evaluación gira en torno a la aplicación de las herramientas presentadas en los antecedentes (sección 3.2), como oportunamente se vio durante los mismos la relación entre precisión y desempeño en la construcción del grafo de llamadas normalmente presenta un desafío dependiendo del tamaño y la naturaleza del código sujeto al análisis. Evaluaremos en esta sección algunas de las variantes de algoritmos provistos y sus variables tecnológicas de interés cómo la exclusión de librerías.

En cuanto al tratamiento de las librerías, como se comentó en la sección 5.2 ambas herramientas exponen la funcionalidad de excluirlas mediante máscaras de paquetes. Tomaremos tres casos que consideramos representativos a los fines de conducir una primera exploración sobre los algoritmos ofrecidos, un análisis que solamente tenga en cuenta las clases propias de la aplicación y excluya todas las librerías (**full-**exclusions), un análisis que tenga en cuenta aproximadamente la mitad del total de las clases (**half-**exclusions) y que en particular contenga aquellas dependencias propias del framework Spring, y por último un análisis con todas las dependencias del proyecto (**no-**exclusions).

Como fue mencionado en la sección 5.2 es importante recordar que estas exclusiones tienen un impacto ya que además de perder *soundness* en el análisis (posibles callbacks de librerías), si existen clases cuya definición depende de alguna librería (ie. una clase extiende o implementa una interfaz contenida en la misma) dicha clase no será considerada en cuenta por nuestro análisis comprometiendo la precisión del grafo de llamadas.

Estudiaremos entonces las ejecuciones de los algoritmos en cada herramienta midiendo desempeño junto a la dimensión del resultado (cantidad de llamados), además dividiremos los resultados bajo los mencionados niveles de exclusión de librerías. Todos los experimentos fueron ejecutados en una computadora con procesador Intel i5 de cuarta generación, un máximo de 8gb de RAM y un tiempo máximo de una hora.

	Soot CHA			Soot Spark		
	t(s)	ram(mb)	nodos	t(s)	ram(mb)	nodos
full-exclusions	15.6	506.5	8995	32	481.0	6564
half-exclusions	OOM			51.36	870.0	5303
no-exclusions	OOM			1504.61	3555.5	5890

Tabla 9.1: Resultados de desempeño para los algoritmos ofrecidos por Soot.

	Wala RTA			Wala 0-CFA			Wala 0-1-CFA			Wala 2-CFA		
	t(s)	ram(mb)	nodos	t(s)	ram(mb)	nodos	t(s)	ram(mb)	nodos	t(s)	ram(mb)	nodos
full-exclusions	16.5	1284.5	7842	22.2	1317.0	4060	45.0	1329.0	2842	2903.4	3648	3872
half-exclusions	38	1385	8745	24.4	1319.0	4306	12393	2858.0	4204	TIMEOUT		
no-exclusions	179.7	2097.0	24012	TIMEOUT			TIMEOUT			TIMEOUT		

TABLA 9.2: Resultados de desempeño para los algoritmos ofrecidos por Wala.

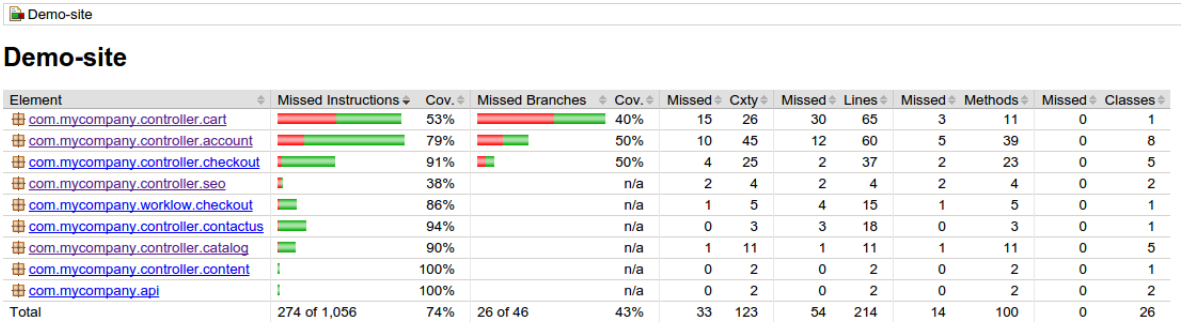
Como se puede ver existen grandes diferencias en el desempeño entre los algoritmos según varía su sensibilidad al contexto y la cantidad de librerías tenidas en cuenta por el análisis. Tomaremos a los resultados de estos algoritmos como sujetos de prueba para el resto de las evaluaciones, consideramos que si bien es de interés un trabajo futuro que brinde más detalles sobre la relación optima entre sensibilidad del algoritmo y librerías, los resultados obtenidos representan una muestra satisfactoria para los objetivos del trabajo.

9.1.1. Precisión y exhaustividad de los algoritmos para construcción de grafo de llamadas

Habiendo construido los diferentes grafos de llamadas el paso siguiente está en preguntarse que precisión y recall poseen los mismos para entender las características de la información sobre la cual se basarán luego las técnicas de reconstrucción propuestas. Como fue mencionado en la sección 3.3 los estudios previos acerca de la precisión de grafos de llamadas suelen comparar sus resultados con datos conseguidos a partir de ejecuciones reales. Esta práctica por lo general se realiza a partir de los grafos de control ejercitados por una suite de tests del sistema, sin embargo consideramos que esta característica no se ajusta del todo a nuestros objetivos debido a que (a menos que los tests sean de integración) probablemente no se logre captar interacciones propias del sistema en conjunto.

Una alternativa que consideramos más afín a las interacciones reales de la aplicación es la comparación de los grafos de llamadas con las ejecuciones de casos de uso de la aplicación. Tomaremos esta dirección a partir de los casos presentados en la sección 8.1.1 y será necesario en principio fundamentar la representatividad de los mismos, una medida común en la literatura para afirmar la completitud de este tipo de información usualmente es alguna medida de cobertura (métodos, líneas, ramas) de las ejecuciones sobre el código analizado. Mostraremos a continuación el análisis sobre la representatividad de las ejecuciones propias de los casos de uso presentados conseguido a partir de la herramienta *jacoco*[1].

[1] http://eclemma.org/jacoco/



Demo-site

Demo-site

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.mycompany.controller.cart		53%		40%	15	26	30	65	3	11	0	1
com.mycompany.controller.account		79%		50%	10	45	12	60	5	39	0	8
com.mycompany.controller.checkout		91%		50%	4	25	2	37	2	23	0	5
com.mycompany.controller.seo		38%		n/a	2	4	2	4	2	4	0	2
com.mycompany.worklow.checkout		86%		n/a	1	5	4	15	1	5	0	1
com.mycompany.controller.contactus		94%		n/a	0	3	3	18	0	3	0	1
com.mycompany.controller.catalog		90%		n/a	1	11	1	11	1	11	0	5
com.mycompany.controller.content		100%		n/a	0	2	0	2	0	2	0	1
com.mycompany.api		100%		n/a	0	2	0	2	0	2	0	2
Total	274 of 1,056	74%	26 of 46	43%	33	123	54	214	14	100	0	26

FIGURA 9.1: Resumen de cobertura para la aplicación de prueba DemoSite.

Podemos ver que los porcentajes de cobertura a nivel de instrucciones y ramificaciones dan una seguridad parcial en la cobertura (74 % y 43 % respectivamente), sin embargo la cantidad de métodos ejercitados versus el total de métodos definidos (86 %) es un indicador que consideramos de confianza acerca de la representatividad de la interacciones sucedidas durante las ejecuciones.

Es importante recordar como se menciona en el capítulo 8.1.1 que la aplicación sobre la que experimentamos (DemoSite) no hace uso de todas las funcionalidades provistas por la librería en la que se apoya (BroadleafCommerce) por lo tanto los resultados de cobertura presentados se enmarcan solamente en las funcionalidades aprovechadas por la aplicación de prueba. Un trabajo posterior podría extender el presente y abarcar también los caminos de ejecución de toda la librería BroadleafCommerce, para tener una referencia, la cobertura total de los casos de uso considerando el código de la librería junto a la aplicación de prueba ronda el 30 %.

Habiendo concluido la etapa de validación respecto a la representatividad de los casos de uso, pasamos a la grabación de las interacciones propias de los mismos, para conseguir esta información de ejecución se desarrollo una herramienta sobre la tecnología de *agentes* concebida para la instrumentación de programas que corren en maquinas virtuales de Java (JVM), esta herramienta permite grabar las interacciones entre invocaciones que se correspondan a clases contenidas en paquetes de interés como también definir exclusiones para métodos o clases de los cuales no se desea tener registro.

Una vez grabado los datos propios de las ejecuciones, nos encontramos con varias interacciones propias de frameworks y comportamientos que caen fuera de la simulación propuesta durante el capítulo 5 y que son el producto de una aplicación compleja que se apoya en numerosos frameworks para lograr su cometido, estas interacciones son las siguientes:

- **Spring Web:** Permite configurar beans para filtrar pedidos o respuestas por los motivos que se consideren necesarios como pueden ser la asignación de algún Header HTTP o agregar un gestor personalizado de dispatch. En el caso estudiado, la librería BroadleafCommerce registra un bean con este propósito llamado

`blURLHandlerFilter`, esto aporta un llamado previo a la intervención de cada uno de los controladores.

- **Hibernate:** Se encarga de la persistencia de manera tal que se encarga de transformar entidades almacenadas en una base de datos en objetos y viceversa. Su aporte en el flujo de control se traduce en la invocación de transformadores de clases u objetos y se producen cuando las mismas son cargadas o re-definidas en la JVM. Es decir, que interrumpen de manera aleatoria el flujo que estamos estudiando.
- **Spring AOP:** Otorga la capacidad de modificar el comportamiento de uno o más métodos mediante instrumentación del bytecode, su principal ventaja es la no intromisión en el modelo de la aplicación. Sin embargo durante la ejecución sus interacciones se vuelven visibles y ocurren según la configuración especificada, en los métodos elegidos antes, durante o luego de su ejecución.

Considerando que estos frameworks y sus funcionalidades caen fuera de los objetivos de simulación (Spring IoC y Spring MVC) construidas a partir del *dummy main* se decidió remover sus efectos de la traza registrada a los fines prácticos de evaluar la precisión y recall del método propuesto. La cantidad de interacciones de la traza conjunta sin estos aportes se redujo en un 38 % pasando de 6000 a 3700 para el **caso-completo**, dicha remoción tendrá un impacto en las reconstrucciones ya que solamente son eliminados los llamados internos que cada framework ejecuta pero no los llamados hacia sus interfaces desde los componentes de negocio, denominaremos a estas interfaces como componentes tecnológicos ya que representan herramientas tales como bases de datos, motores de vistas y herramientas de paginación o caché.

Presentamos entonces los resultados de recall para las configuraciones (algoritmo y exclusión) que no presentaron errores en sus características de desempeño. Dividiremos el resultado según los casos de uso mencionados en la sección 8.1.1 y mostraremos por cada algoritmo que cantidad de las interacciones reales son encontradas por cada uno bajo las diferentes configuraciones de exclusión de dependencias (no-exclusions, half-exclusions y full-exclusions). Haremos hincapié fundamentalmente en los resultados obtenidos a partir de la herramienta Wala debido a que los resultados conseguidos por Soot presentan características contenidas en la variedad de los algoritmos de Wala y por ende no aportan diversidad en sus resultados.

Caso de uso	Interacciones	RTA			0-CFA		0-1-CFA		2-CFA
		no-excl	half-excl	full-excl	half-excl	full-excl	half-excl	full-excl	full-excl
caso-checkout	3005	73 % (2184)	40 % (1198)	39 % (1165)	30 % (889)	29 % (862)	35 % (1038)	27 % (797)	27 % (818)
caso-login	1126	70 % (793)	38 % (427)	36 % (400)	27 % (307)	26 % (294)	43 % (484)	24 % (273)	25 % (280)
caso-review	1359	63 % (862)	33 % (450)	31 % (423)	24 % (329)	23 % (316)	36 % (484)	22 % (296)	22 % (304)
caso-wishlist	2353	74 % (1746)	39 % (917)	38 % (890)	31 % (720)	30 % (707)	38 % (900)	29 % (683)	29 % (693)
caso-completo	3831	72 % (2773)	42 % (1595)	41 % (1553)	32 % (1236)	32 % (1207)	35 % (1352)	29 % (1094)	30 % (1141)

Tabla 9.3: Resultados de recall reportados por los algoritmos ofrecidos por Wala.

Caso de uso	Interacciones	Soot CHA	Soot Spark		
		full-excl	no-excl	half-excl	full-excl
caso-checkout	3005	58 % (1755)	34 % (1033)	31 % (935)	31 % (935)
caso-login	1126	57 % (644)	35 % (395)	31 % (347)	31 % (347)
caso-review	1359	49 % (662)	31 % (418)	27 % (365)	27 % (365)
caso-wishlist	2353	59 % (1386)	36 % (838)	32 % (759)	32 % (759)
caso-completo	3831	58 % (2214)	36 % (1397)	33 % (1281)	33 % (1276)

TABLA 9.4: Resultados de recall reportados por los algoritmos ofrecidos por Soot.

Además revisaremos que proporción representan las interacciones encontradas sobre el total de métodos econtrados por cada algoritmo, es decir la precisión obtenida. Podemos observar en la figura 9.5 la amplia diferencia de precisión entre RTA y las variaciones de CFA en particular cuando son tenidas en cuenta todas las librerías de la aplicación, brindando una perspectiva completa sobre el balance de precisión y performance de los algoritmos para el tipo de aplicacion y el método propuesto, en cuanto a las configuraciones de CFA las mismas presentan menos variación entre sus resultados.

RTA			0-CFA		0-1-CFA		2-CFA
no-excl	half-excl	full-excl	half-excl	full-excl	half-excl	full-excl	full-excl
2773/24012 12 %	1595/8745 18 %	1553/7842 20 %	1236/4306 29 %	1207/4060 30 %	1352/4204 32 %	1094/2842 38 %	1141/3872 29 %

TABLA 9.5: Resultados de precisión reportados por los algoritmos ofrecidos por Wala según las invocaciones producidas en **caso-completo**.

CHA	Spark		
full-excl	no-excl	half-excl	full-excl
2214/8995 25 %	1397/5890 24 %	1281/5303 24 %	1276/5293 24 %

TABLA 9.6: Resultados de precisión reportados por los algoritmos ofrecidos por Soot según las invocaciones producidas en **caso-completo**.

Naturalmente a partir de los datos contenidos en esta tablas surgen nuevos interrogantes, a saber:

(A) ¿Cuales son las interacciones que no son encontradas por RTA y a que se deben?

(B) ¿Porqué disminuyen las interacciones encontradas cuando se aumenta la sensibilidad de los algoritmos?

Para responder la pregunta **A** se recurrió a una inspección manual sobre las interacciones revelando falencias propias de los algoritmos de grafo de llamadas y del método propuesto en el capítulo 5.

Comenzando por las falencias propias de los algoritmos de grafos de llamadas, como se comentó en la sección 3.3 existen problemas a la hora del análisis de reflection. La aplicación de prueba se apalanca en esta tecnología como se puede ver a continuación en el código de la figura 9.2 y su impacto en la resolución de llamados alcanza aproximadamente el 20 %, esto se produce debido a que una vez que el análisis no puede resolver esta situación, las llamadas derivadas de la misma tampoco son tenidas en cuenta.

```
59  public ExtensionManager(Class<T> _clazz) {
60      extensionHandler = (T) Proxy.newProxyInstance(_clazz.getClassLoader(),
61              new Class[] { _clazz },
62              this);
63  }
```

FIGURA 9.2: Código del constructor de la clase ExtensionManager de BroadleafCommerce.

Prosiguiendo, se encontraron tres causantes de perdida de precisión por parte del método propuesto. Dos de ellas pertenecen de las limitaciones conocidas comentadas en 5.1.1 propias de la instanciación de parámetros en los controladores (5 %) y la falta de soporte para la inyección de ciertas configuraciones como beans sin nombre, o beans sin constructores por defecto (15 %). Sin embargo en esta búsqueda se encontró un caso responsable de la mayoría de ejes no simulados (60 %) que cae fuera de las posibilidades de simulación brindadas por nuestro método y que explicaremos a continuación.

Como explicamos en la introducción a las características de Spring (capítulo 4), este realiza la inicialización de beans e inyección de dependencias en tiempo de ejecución por medio de reflection, nosotros en cambio intentamos simular este comportamiento generando código que debiera realizar invocaciones análogas a las realizadas por el mismo. Esto es posible en la medida que el comportamiento de tiempo de ejecución pueda ser descripto bajo el sistema de tipos pues como vimos en la sección (5.2) el mismo debe ser compilado para ser analizado. Casi siempre esta situación se corresponde y no existen problemas, sin embargo existe un caso que cae fuera de esta lógica y corresponde a los Generics de Java, estos definen contratos de tipos en tiempo de compilación que luego serán desechados en tiempo de ejecución. Esto produce que nuestra simulación no pueda representar asignaciones que Spring efectivamente realizará por restricciones de seguridad en tipos.

En cuanto a la pregunta **B**, como explicamos en la introducción a grafos de llamada (sección 3) cuanto más sensible es un análisis menos ejes espurios debería encontrar, sin embargo aquí vemos que además de ejes espurios algunos ejes reales también son desechados, las diferencias más notorias estan ligadas en primer lugar al impacto ilustrado en la sección 5.2 proveniente de la exclusión de librerías (diferencias entre **no**,**full** y **half**) y en segundo lugar a las mencionadas limitaciones conocidas en la simulación como

dependencias de Spring no resueltas (inicializaciones no generadas en el código) y soporte reducido de instanciación de parámetros, estas limitaciones son en alguna medida salvadas por la sobre-aproximación de RTA pero a medida que se aumenta la sensibilidad del algoritmo empleado se impacta negativamente en la cantidad de invocaciones encontradas que efectivamente serán posibles de ejecutarse (recall).

En resumen, las evaluaciones presentadas en este capítulo ofrecen un panorama amplio de las variantes y calidad de los resultados obtenidos a partir del método propuesto en el capítulo 5 para el análisis de Spring y la construcción de grafos de llamadas mediante las herramientas actuales, respondiendo así los primero interrogantes surgidos en **RQ(1)**.

Capítulo 10

Evaluación de técnicas de reconstrucción arquitectónica

Una vez completada la evaluación sobre la precisión de los métodos propuestos para la obtención de dependencias, procederemos al refinamiento de la pregunta **RQ(2)** acerca del valor que pueden brindar las técnicas para retratar comportamientos de ejecución. Propondremos como primera aproximación a la definición de valor encontrar vistas que descubran agrupamientos cohesivos entre entidades vinculadas (en tiempo de ejecución) a un objetivo de negocio denominados en la introducción de este trabajo como componentes y a su vez que la comunicación entre ellos sea lo más clara y coherente posible según la ejecución del sistema.

Como primer paso en esta búsqueda comenzaremos con una validación clásica, en la cual definiremos una separación de entidades que a nuestro entender deberían corresponder a componentes diferentes y mediremos la distancia de los resultados conseguidos por las técnicas propuestas a la misma. Es importante destacar que esta descomposición de referencia no representa una vista acabada del sistema sino que busca encontrar divisiones básicas que según nuestra visión del sistema deberían ser respetadas por cualquier reconstrucción que evidencie interacciones propias de la ejecución.

Aprovechándonos del hecho de que nuestro objetivo es evaluar las particiones como componentes de ejecución del sistema introduciremos una medida novedosa con el fin de aproximar la calidad en la visualización a partir de propiedades de comunicación entre e intra componentes durante ejecuciones concretas, esta medida contará con la ventaja de ser independientemente de la participación de un experto en su realización. Será clave encontrar correlatos y diferencias de la misma frente a la validación clásica, y estudiar los mismos en búsqueda de falencias de ambos enfoques. Por último este proceso contribuirá con nuevas perspectivas para la construcción de técnicas futuras.

Para terminar realizaremos una inspección manual sobre las reconstrucciones más prometedoras o cuyas características despierten particular interés con el objetivo de indagar en profundidad sobre las representaciones obtenidas y dar una visión acabada de sus ventajas y desventajas.

10.1. Validación clásica

La primer validación que realizaremos se basa en aquellas mencionadas en los antecedentes del trabajo (sección 2.3) donde las descomposiciones son contrastadas con una reconstrucción de referencia construida por un arquitecto o experto en el sistema.

Como mencionamos en la sección anterior nuestra aproximación de referencia consistira en seleccionar manualmente algunas entidades que a nuestro mejor saber y entender deberían por sus características de negocio e interacción encontrarse en el mismo componente de ejecución. A partir de ellas calcularemos las distancias de los agrupamientos conocidos y los propuestos a las mismas, esta será nuestra primer medida de la calidad de los resultados para mantener estas estructuras que si bien no representan completamente la arquitectura, son una primera intuición de división entre componentes. Más adelante en la sección 10.3.1 presentaremos una representación visual de la misma.

Como métrica de comparación, tomaremos las distancias mencionadas en los antecedentes del trabajo MoJo y MoJoFM. Mojo consiste en contar la cantidad mínima de movimientos o *joins* necesarios para transformar una descomposición en otra y MoJoFM plantea una relación relativa sobre todas las posibles descomposiciones, expresando con un 100 % una correspondencia óptima con la arquitectura de referencia.

Presentaremos entonces las distancias obtenidas entre los resultados de las técnicas conocidas y propuestas con la reconstrucción de referencia. La nomenclatura para los agrupamientos de la técnica estructural será la siguiente: `algoritmo-excl-pasos`. Siendo `excl` las exclusiones de librerías: no (N), half (H) y full (F), y `pasos` la cantidad de pasos considerada por Walktrap. Por otro lado la nomenclatura para los agrupamientos conseguidos a partir de la técnica basada en información léxica será `T-topics-weight`, donde `topics` representa la cantidad de topics y `weight` el peso de las dependencias obtenidas por el framework. Ordenaremos los resultados de menor a mayor distancia con la arquitectura de referencia.

Agrupamiento	MoJo	MojoFM
0-CFA-F-s15	65	63.07 %
0-CFA-H-s25	70	60.23 %
0-CFA-F-s25	70	60.23 %
0-CFA-F-s10	73	58.52 %
2-CFA-F-s25	75	57.39 %
0-CFA-H-s8	75	57.39 %
0-CFA-H-s15	75	57.39 %
0-1-CFA-F-s25	75	57.39 %
0-CFA-F-s8	77	56.25 %
0-CFA-H-s10	78	55.68 %
2-CFA-F-s4	81	53.98 %
0-1-CFA-F-s4	81	53.98 %
2-CFA-F-s15	82	53.41 %
0-1-CFA-F-s15	82	53.41 %
2-CFA-F-s8	83	52.84 %
0-1-CFA-F-s8	83	52.84 %
2-CFA-F-s10	85	51.70 %
0-1-CFA-F-s10	85	51.70 %
2-CFA-F-s5	89	49.43 %
0-CFA-F-s5	89	49.43 %
0-1-CFA-F-s5	89	49.43 %
0-CFA-H-s5	90	48.86 %
RTA-N-s15	92	47.73 %
RTA-H-s15	92	47.73 %
RTA-F-s15	95	46.02 %
RTA-F-s10	98	44.32 %
RTA-H-s25	101	42.61 %
0-CFA-H-s4	102	42.05 %
RTA-H-s10	103	41.48 %
RTA-N-s25	104	40.91 %
0-CFA-F-s4	105	40.34 %
RTA-N-s10	109	38.07 %
RTA-N-s5	111	36.93 %
RTA-F-s25	113	35.80 %
RTA-N-s4	114	35.23 %
RTA-F-s8	114	35.23 %
RTA-H-s8	115	34.66 %
RTA-F-s5	116	34.09 %
RTA-H-s5	117	33.52 %
RTA-H-s4	121	31.25 %
RTA-F-s4	124	29.55 %

Tabla 10.1: Distancia MoJo y MoJoFM a la arquitectura de referencia para técnicas basadas en información de grafo de llamadas.

Agrupamiento	MoJo	MojoFM
bunch-nahc-hill	56	68.18 %
bunch-sahc-hill	57	67.61 %
ACDC-bso	106	39.77 %
ACDC-so	107	39.20 %
ACDC-s	110	37.50 %
ACDC-bsou	115	34.66 %
ACDC-su	116	34.09 %
ACDC-sou	116	34.09 %
ACDC-bsu	116	34.09 %
ACDC-bou	131	25.57 %
ACDC-bo	131	25.57 %

Tabla 10.2: Distancia MoJo y MoJoFM a la arquitectura de referencia según las técnicas ACDC y Bunch en sus distintas configuraciones.

Agrupamiento	MoJo	MojoFM
T-25-20	52	64.86 %
T-50-10	57	61.49 %
T-200-10	58	60.81 %
T-50-20	59	60.14 %
T-100-10	60	59.46 %
T-50-40	62	58.11 %
T-25-40	62	58.11 %
T-25-10	63	57.43 %
T-100-20	63	57.43 %
T-10-20	63	57.43 %
T-100-40	64	56.76 %
T-200-20	68	54.05 %
T-200-40	70	52.70 %
T-10-40	70	52.70 %
T-100-0	71	52.03 %
T-50-0	73	50.68 %
T-25-0	74	50.00 %
T-200-0	76	48.65 %
T-10-10	76	48.65 %
T-10-0	79	46.62 %

Tabla 10.3: Distancia MoJo y MoJoFM a la arquitectura de referencia para técnicas basadas en información léxica. Se tomaron 10, 25, 50, 100 y 200 tópicos y un k de 0, 10, 20 y 40.

Analizando los resultados encontramos varias observaciones llamativas, sobre las técnicas estructurales es notable variabilidad obtenida que oscila entre un 63.07 % y 29.55 % brindando el primer indicio sobre el rol determinante de la precisión del grafo de llamadas (**RQ(3)**) ya que los agrupamientos basados en RTA se concentran en la parte baja de la tabla mientras que los más precisos (CFA) ocurren casi en su totalidad en la parte superior.

En cuanto a las técnicas basadas en dependencias de uso se evidencian dos situaciones la primera es el desempeño obtenido por la técnica Bunch mejorando incluso los resultados de las técnicas estructurales, y la segunda es la amplia diferencia entre ACDC y Bunch (67.61 % y 39.77 %) reproduciendo las características conocidas en los estudios de reconstrucción (mencionadas en la sección 2.3).

Llama la atención el buen desempeño de técnicas basadas en información léxica junto a su reducida variabilidad (64.86 %-46 %), además es importante ver que aquellos agrupamientos independientes de la información de Spring (terminados en -0) se ubican en el final de la tabla indicando una mejora incipiente respecto al estado del arte en este tipo de técnicas. Exploraremos en la siguiente secciones propiedades que nos brindarán perspectivas más detalladas del desempeño de los resultados de cada técnica.

10.2. Comparación de modularidad y cantidad de llamados

Continuaremos la evaluación presentando nuestro segundo criterio, esta vez independiente de un experto o descomposición del sistema.

A diferencia de las vistas arquitectónicas estáticas donde las relaciones se componen a partir de dependencias de uso entre las clases, para construir una vista donde se busca evidenciar las interacciones de componentes en ejecución se dispone de una herramienta extra aparte del código fuente de la aplicación, la ejecución concreta del mismo.

Utilizaremos estas ejecuciones para medir como se plasman las mismas según cada reconstrucción, para ello haremos uso de los casos de uso (definidos en 8.1.1) en los cuales se espera que componentes del sistema adquieran diferentes grados de participación. Plantearemos dos propiedades complementarias para evaluar la idoneidad de una representación, la primera consistirá en la modularidad de los agrupamientos sobre las relaciones de ejecución, esta medida propuesta inicialmente para encontrar comunidades en redes [48] de distinto tipo (sociales, biológicas, etc) se presenta como una aproximación a la intuición de cohesión entre componentes mencionada anteriormente y su definición es la siguiente.

$$Q = \frac{1}{2m} * \left[\sum_{ij} (A_{ij} - \frac{ki * kj}{2m}) \delta(ci, cj) \right] \tag{10.1}$$

Donde m representa el numero de ejes, A_{ij} es el elemento ij de la matriz de adyacencia, ki es el grado del nodo i, kj el grado del nodo j, ci y cj las comunidades a las que pertenece cada nodo y $\delta(x, y)$ es 1 cuando $x == y$ y 0 en cualquier otro caso, el valor de modularidad más elevado posible es 1.

La idea detrás de esta definición esta asociada a señalar agrupamientos cuyos elementos se relacionan entre sí por encima de la probabilidad aleatoria de que lo hagan, es por eso que cada interacción se resta con la probabilidad aleatoria de su ocurrencia según los grados de los nodos $\frac{ki*kj}{2m}$. Volviendo a nuestro dominio esta formula nos dará un indicio de como estarán repartidas las interacciones dentro de los componentes. Por otro lado, proponemos medir las interacciones entre los agrupamientos ya que podrían ocultarse de la modularidad siempre y cuando la cohesión de los mismos sea positiva.

Mostraremos a continuación los resultados obtenidos a partir de la medición de modularidad y cantidad de relaciones entre clusters. Utilizaremos la misma nomenclatura y división presentada en la sección previa.

Agrupamiento	**caso-checkout**		**caso-login**		**caso-review**		**caso-wishlist**		**caso-completo**		AVG	
	# llamados	mod	# llamados	mod	# llamados	mod	# llamados	mod	# llamados	mod	# llamados	mod
Arq de referencia	16	0.68	2	0.55	10	0.62	15	0.69	18	0.70	12.2	0.64

FIGURA 10.1: Modularidad y cantidad de llamados entre agrupamientos de la arquitectura de referencia.

Agrupamiento	**caso-checkout**		**caso-login**		**caso-review**		**caso-wishlist**		**caso-completo**		AVG	
	# llamados	mod	# llamados	mod	# llamados	mod	# llamados	mod	# llamados	mod	# llamados	mod
0-1-CFA-F-s10	73	0.46	52	0.50	63	0.48	65	0.49	79	0.44	66.4	0.47
0-1-CFA-F-s15	85	0.52	60	0.53	71	0.51	76	0.53	90	0.50	76.4	0.52
0-1-CFA-F-s25	61	0.44	43	0.48	47	0.50	53	0.49	65	0.42	53.8	0.46
0-1-CFA-F-s4	134	0.42	91	0.51	108	0.51	115	0.43	142	0.38	118.0	0.45
0-1-CFA-F-s5	94	0.43	65	0.50	77	0.46	83	0.47	103	0.40	84.4	0.45
0-1-CFA-F-s8	92	0.48	63	0.55	73	0.49	80	0.50	99	0.46	81.4	0.49
0-CFA-F-s10	229	0.42	144	0.45	165	0.45	196	0.42	252	0.41	197.2	0.43
0-CFA-F-s15	114	0.45	80	0.48	92	0.45	98	0.46	121	0.45	101.0	0.46
0-CFA-F-s25	74	0.46	49	0.47	54	0.46	62	0.47	79	0.45	63.6	0.46
0-CFA-F-s4	440	0.35	236	0.40	268	0.38	354	0.37	496	0.36	358.8	0.37
0-CFA-F-s5	353	0.36	210	0.42	240	0.40	298	0.38	404	0.37	301.0	0.39
0-CFA-F-s8	188	0.43	133	0.50	155	0.48	172	0.44	208	0.43	171.2	0.45
0-CFA-H-s10	211	0.41	122	0.46	145	0.45	181	0.41	229	0.41	177.6	0.43
0-CFA-H-s15	224	0.43	131	0.47	154	0.47	192	0.43	247	0.43	189.6	0.45
0-CFA-H-s25	90	0.44	57	0.48	66	0.47	78	0.45	97	0.44	77.6	0.45
0-CFA-H-s4	458	0.36	245	0.42	285	0.40	387	0.39	516	0.37	378.2	0.39
0-CFA-H-s5	343	0.36	200	0.42	232	0.40	289	0.39	393	0.37	291.4	0.39
0-CFA-H-s8	236	0.42	147	0.47	178	0.45	205	0.42	262	0.42	205.6	0.44
2-CFA-F-s10	73	0.46	52	0.50	63	0.48	65	0.49	79	0.44	66.4	0.47
2-CFA-F-s15	85	0.52	60	0.53	71	0.51	76	0.53	90	0.50	76.4	0.52
2-CFA-F-s25	61	0.44	43	0.48	47	0.50	53	0.49	65	0.42	53.8	0.46
2-CFA-F-s4	134	0.42	91	0.51	108	0.51	115	0.43	142	0.38	118.0	0.45
2-CFA-F-s5	94	0.43	65	0.50	77	0.46	83	0.47	103	0.40	84.4	0.45
2-CFA-F-s8	92	0.48	63	0.55	73	0.49	80	0.50	99	0.46	81.4	0.49
RTA-F-s10	434	0.35	252	0.40	307	0.39	390	0.34	495	0.35	375.6	0.37
RTA-F-s15	430	0.34	256	0.40	298	0.38	376	0.34	492	0.35	370.4	0.36
RTA-F-s25	509	0.34	299	0.37	331	0.37	435	0.33	574	0.33	429.6	0.35
RTA-F-s4	523	0.34	300	0.37	355	0.35	445	0.34	600	0.35	444.6	0.35
RTA-F-s5	546	0.34	295	0.38	358	0.35	459	0.33	623	0.33	456.2	0.35
RTA-F-s8	501	0.36	272	0.38	315	0.37	419	0.35	573	0.36	416.0	0.36
RTA-H-s10	420	0.35	227	0.40	282	0.39	359	0.34	479	0.35	353.4	0.37
RTA-H-s15	405	0.36	231	0.40	264	0.40	346	0.35	453	0.36	339.8	0.37
RTA-H-s25	462	0.33	264	0.38	295	0.38	393	0.33	516	0.33	386.0	0.35
RTA-H-s4	543	0.33	290	0.38	349	0.36	450	0.33	618	0.34	450.0	0.35
RTA-H-s5	520	0.34	277	0.38	333	0.36	436	0.33	588	0.34	430.8	0.35
RTA-H-s8	500	0.34	278	0.38	322	0.37	421	0.33	574	0.34	419.0	0.35
RTA-N-s10	352	0.37	243	0.39	260	0.39	310	0.38	398	0.38	312.6	0.38
RTA-N-s15	221	0.38	140	0.49	177	0.47	196	0.44	239	0.38	194.6	0.43
RTA-N-s25	166	0.39	110	0.43	135	0.41	151	0.42	172	0.37	146.8	0.40
RTA-N-s4	348	0.34	210	0.41	255	0.40	294	0.38	390	0.35	299.4	0.38
RTA-N-s5	541	0.32	324	0.33	362	0.35	448	0.34	621	0.32	459.2	0.33
AVG(mod)	0.40		0.45		0.43		0.41		0.39			

Tabla 10.4: Modularidad y cantidad de llamados entre agrupamientos de las técnicas basadas en información de grafos de llamada.

Agrupamiento	**caso-checkout**		**caso-login**		**caso-review**		**caso-wishlist**		**caso-completo**		AVG	
	# llamados	mod	# llamados	mod	# llamados	mod	# llamados	mod	# llamados	mod	# llamados	mod
ACDC-bo	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0.0	0.00
ACDC-bou	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0.0	0.00
ACDC-bso	191	0.27	90	0.34	97	0.35	144	0.31	207	0.27	145.8	0.31
ACDC-bsou	137	0.27	61	0.31	71	0.33	107	0.31	145	0.26	104.2	0.29
ACDC-bsu	120	0.23	62	0.29	72	0.30	98	0.28	128	0.21	96.0	0.26
ACDC-s	162	0.24	88	0.30	96	0.30	130	0.27	182	0.22	131.6	0.27
ACDC-so	190	0.27	87	0.34	94	0.35	143	0.31	207	0.27	144.2	0.31
ACDC-sou	144	0.27	61	0.32	71	0.34	111	0.31	152	0.27	107.8	0.30
ACDC-su	120	0.23	62	0.29	72	0.30	98	0.28	128	0.21	96.0	0.26
bunch-nahc-hill	178	0.43	92	0.57	105	0.57	147	0.49	191	0.44	142.6	0.50
bunch-sahc-hill	323	0.41	130	0.54	156	0.53	259	0.45	372	0.39	248.0	0.46
AVG(mod)	0.25		0.32		0.32		0.28		0.24			

TABLA 10.5: Modularidad y cantidad de llamados entre agrupamientos de la arquitectura de referencia por ACDC y Bunch en sus diferentes configuraciones.

Agrupamiento	**caso-checkout**		**caso-login**		**caso-review**		**caso-wishlist**		**caso-completo**		AVG	
T-100-0	371	0.32	127	0.44	155	0.42	256	0.35	413	0.32	264.4	0.37
T-100-10	417	0.24	128	0.41	166	0.38	300	0.27	467	0.24	295.6	0.31
T-100-20	370	0.25	130	0.40	164	0.38	271	0.29	419	0.25	270.8	0.31
T-100-40	368	0.25	139	0.41	181	0.36	278	0.29	416	0.24	276.4	0.31
T-10-0	38	0.34	25	0.43	27	0.43	33	0.37	38	0.32	32.2	0.38
T-10-10	43	0.28	26	0.39	29	0.40	39	0.33	43	0.27	36.0	0.33
T-10-20	35	0.29	18	0.35	19	0.36	27	0.32	35	0.29	26.8	0.32
T-10-40	32	0.27	17	0.31	24	0.29	26	0.30	32	0.27	26.2	0.29
T-200-0	457	0.29	157	0.38	199	0.40	324	0.32	512	0.30	329.8	0.34
T-200-10	479	0.25	156	0.41	208	0.39	350	0.28	539	0.24	346.4	0.31
T-200-20	476	0.23	153	0.39	200	0.37	339	0.27	541	0.23	341.8	0.30
T-200-40	472	0.24	161	0.38	207	0.36	356	0.27	548	0.24	348.8	0.30
T-25-0	150	0.36	72	0.49	89	0.46	126	0.39	162	0.38	119.8	0.42
T-25-10	145	0.31	58	0.43	69	0.42	118	0.33	156	0.29	109.2	0.36
T-25-20	145	0.31	64	0.40	77	0.41	115	0.34	149	0.31	110.0	0.36
T-25-40	128	0.25	53	0.33	65	0.35	109	0.27	133	0.24	97.6	0.29
T-50-0	246	0.35	97	0.40	128	0.38	183	0.37	264	0.36	183.6	0.37
T-50-10	250	0.28	82	0.43	101	0.43	178	0.31	274	0.28	177.0	0.35
T-50-20	235	0.26	96	0.41	118	0.41	182	0.30	255	0.26	177.2	0.33
T-50-40	221	0.25	80	0.37	105	0.36	177	0.27	250	0.25	166.6	0.30
AVG	0.28		0.40		0.39		0.31		0.28			

TABLA 10.6: Modularidad y cantidad de llamados entre agrupamientos de las técnicas basadas en información léxica.

Para facilitar la comprensión de los datos, mostraremos dos gráficos comparando los resultados de algunas reconstrucciones de interés (figura 10.2) y su relación con MoJoFM (figura 10.3). Los mismos representan en los ejes de coordenadas la modularidad y la cantidad de llamados entre clusters. Según estas propiedades en el gráfico podemos pensar a los agrupamientos que se encuentran en el extremo inferior derecho como aquellos que presentan mejores características (modularidad alta y cantidad de llamados entre agrupamientos baja), mientras que los que se encuentran en el extremo superior izquierdo aquellos que presentan peores características (modularidad baja y cantidad de llamados entre clusters alta).

Además marcaremos con distintos colores las técnicas, de la siguiente manera:

- Azul: Resultados de la Técnica estructural presentada en el capítulo 6 a partir de grafos de llamada precisos (k-CFA).
- Verde: Resultados de la Técnica estructural a partir de grafos de llamada imprecisos (RTA).
- Rojo: Resultados de la técnica Bunch y ACDC.
- Violeta: Resultados de la técnica basada en información léxica presentada en el capítulo 6.

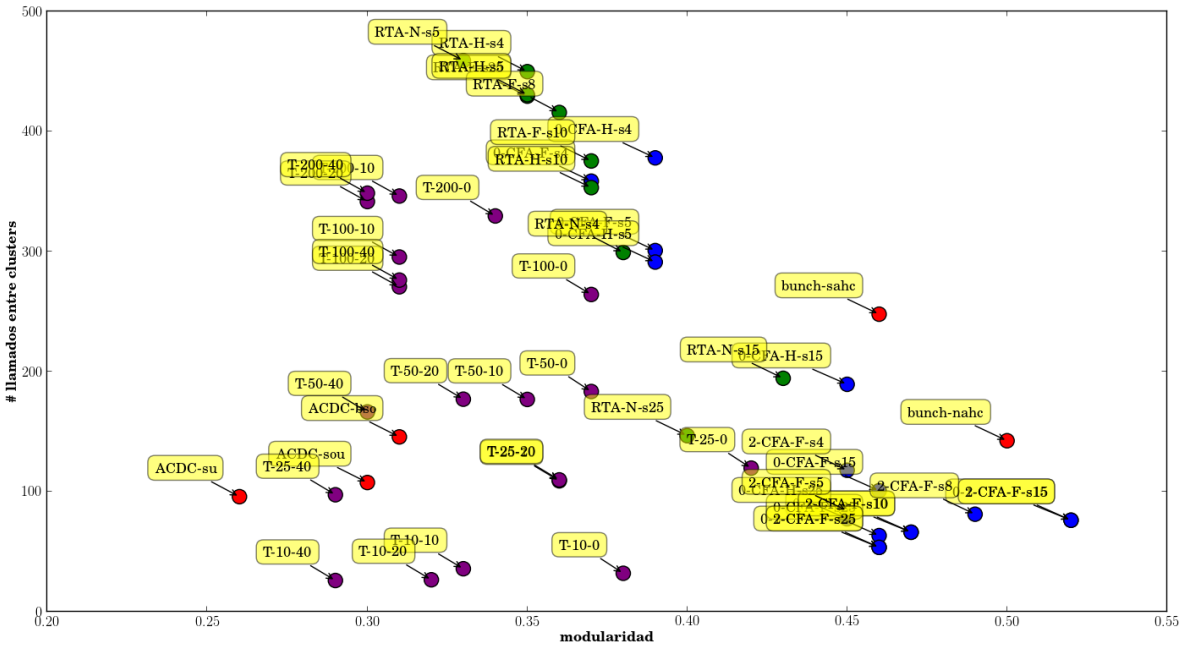


FIGURA 10.2: Gráfico de comparación entre modularidad y cantidad de ejes. Los colores diferencian las técnicas de reconstrucción.

Por otro lado, presentamos un gráfico complementario basado en el anterior donde los nodos adquieren su tamaño según el valor de distancia MoJoFM a la arquitectura de referencia, esto es útil para la comparación entre las propiedades y sus resultados para la comprensión de la calidad en una vista.

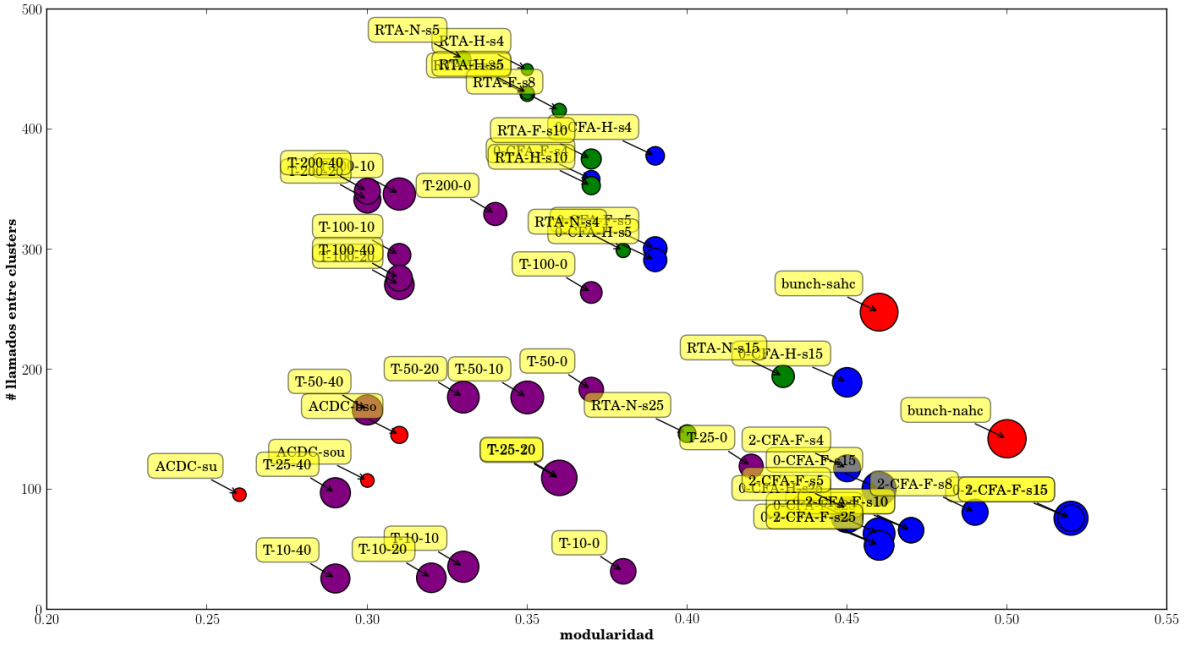


FIGURA 10.3: Gráfico de comparación entre modularidad y cantidad de ejes. Los colores diferencian las técnicas de reconstrucción.

A partir de estas representaciones podemos observar similitudes y diferencias con respecto a las distancias reportadas en la sección anterior, entre los puntos en común podemos observar la pronunciada diferencia tanto en modularidad como cantidad de llamados entre las técnicas que utilizan algoritmos de grafos de llamada con más precisión (CFA) y menos precisos (RTA) encontrando así otro indicio para responder **RQ(3)**. Por otro lado también encuentra un correlato el desempeño de las técnicas estructurales, presentando las variantes de ACDC una modularidad consistentemente menor en comparación con las de Bunch.

Entre las diferencias más notables respecto a los resultados de MoJoFM se encuentran las descomposiciones conseguidas a partir de tópicos, estas se encuentran en un nivel modular más bajo en comparación de las técnicas estructurales basadas en CFA y Bunch que la distancia MoJoFM presentada en la sección anterior no refleja. A su vez según la cantidad de los mismos (que determinan la cantidad de componentes) se distingue una marcada diferencia en cantidad de llamados, lo que podría requerir un ajuste relativo entre la cantidad de clusters y los llamados entre si para evitar sobre-valorar descomposiciones con pocos agrupamientos.

En cuanto a los agrupamientos obtenidos a partir de Bunch, que resultaban según MoJoFM como los más prometedores hasta ahora, observando las características propuestas en esta sección podemos apreciar que si bien su modularidad es elevada presentan una cantidad de interacciones entre clusters peor que las técnicas estructurales

propuestas (CFA) y de orden semejante a ACDC.

Estas observaciones nos motivan a tomar el siguiente paso y evaluar en detalle la calidad de los agrupamientos obtenidos en los casos mencionados donde las medidas presentan discrepancias o cuyas características son de interés.

10.3. Inspección manual de las reconstrucciones

Avanzando sobre las experimentaciones previas, con la exploración manual de las vistas buscamos observar características que caen fuera de los análisis en cuanto a las virtudes o falencias que aportan las mismas a la hora de describir un comportamiento de ejecución. Concretamente mostraremos vistas de como se plasman las reconstrucciones sobre las interacciones propias de los casos de uso seleccionados en 8.1.1 considerados como elementos representativos del sistema. Estas vistas permitirán evaluar con más detalle las cualidades de cada una y contrastarán el impacto en su calidad según las diferencias reportadas en las secciones anteriores. Mostraremos visualizaciones de las mismas en las cuales representaremos a los agrupamientos con cuadrados, cuyo tamaño se determinará según la cantidad de clases involucradas en ejecución y a la comunicación entre agrupamientos con flechas dirigidas de color verde, cuyo tamaño dependerá de la cantidad de interacciones que se dan entre sus respectivas clases.

10.3.1. Arquitectura de referencia

Previamente a la exploración visual de las arquitecturas mostraremos una vista de la arquitectura de referencia utilizada en la sección 10.1, es importante recordar que la misma representa una aproximación reducida de la misma e intenta encontrar aquellas representaciones que mantienen su estructura lo mejor posible. Mostramos esta vista con el objetivo de que el lector pueda apreciar la naturaleza de los componentes que intentamos encontrar en los resultados de las técnicas conocidas y propuestas.

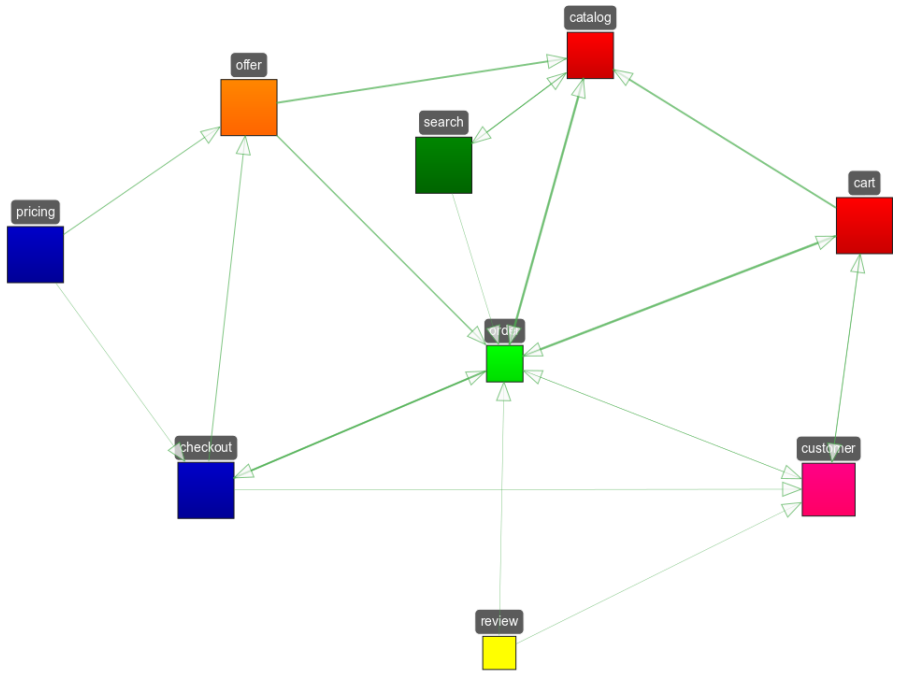


FIGURA 10.4: Vista de la arquitectura de referencia presentada en 10.1.

10.3.2. Exploración visual

Comenzaremos la exploración a partir de algunas vistas seleccionadas como casos de interés en las secciones de evaluación previas, estas se muestran en la figura 10.5 y son ACDC-bso, Bunch-NAHC, 0-CFA-F-s25, RTA-F-s10,T-10-0 y T-25-20. Utilizaremos para este acercamiento tres de los casos de uso presentados en la sección 8.1.1, estos son: **caso-login**, **caso-checkout** y **caso-review** debido a que ejercitan áreas diferentes del sistema. Mostraremos entonces la representación de las técnicas para dichos casos de uso, contemplando las entidades y llamados efectivamente realizados en su consecución. Principalmente veremos en las siguientes vistas relaciones entre agrupamientos (cantidad y diferencias) y los tamaños asignados según cada técnica que entran en juego según el caso de uso retratado.

Luego en la sección siguiente iremos más a fondo desglosando algunas de estas vistas, y allí se podrá hacer mayor hincapié en el rol de la modularidad y su impacto en la calidad de la vista.

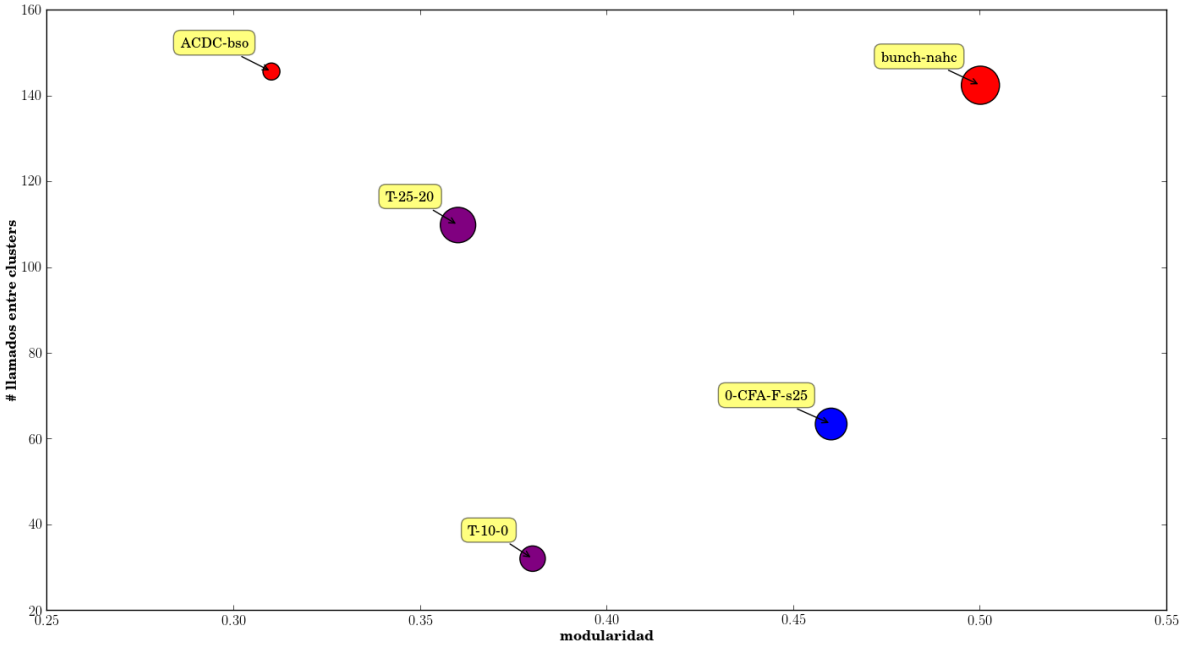


FIGURA 10.5: Agrupamientos de interés elegidos para la exploración manual.

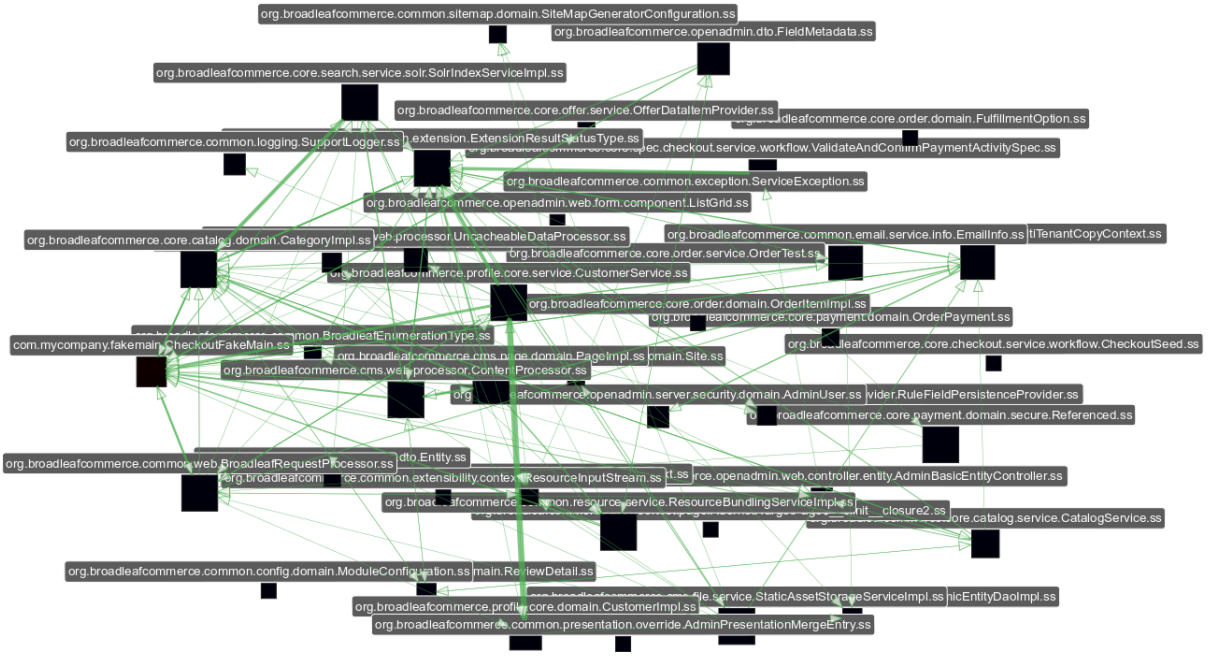


FIGURA 10.6: Vista de Bunch-NAHC sobre **caso-login**.

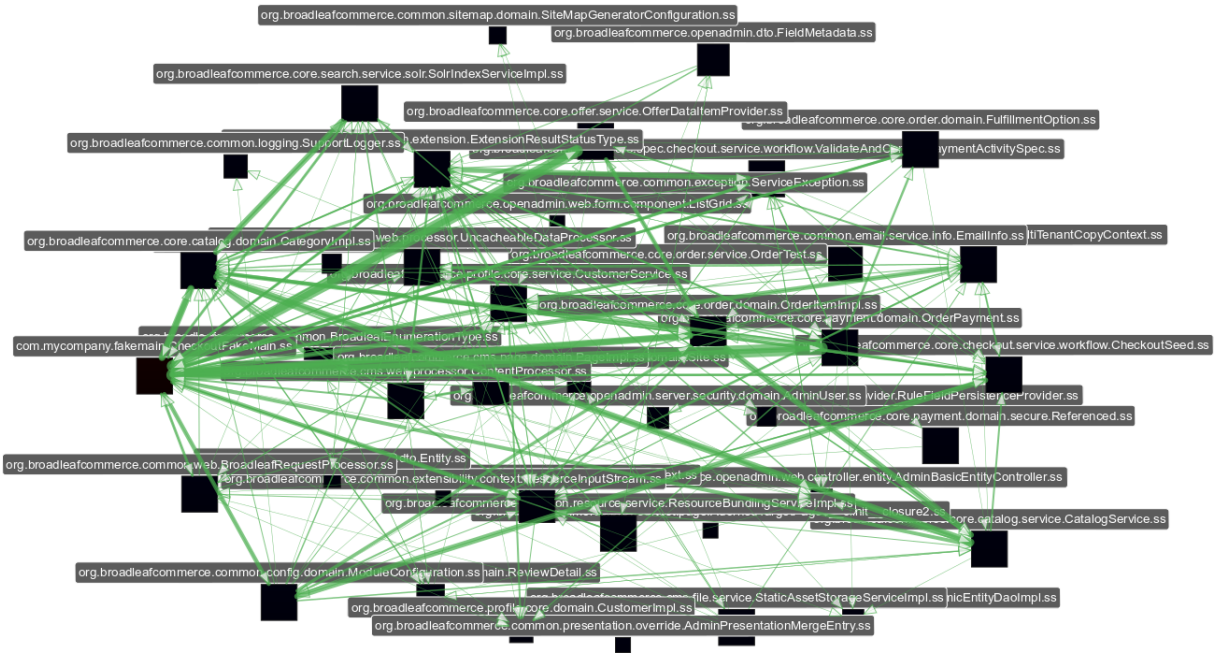


FIGURA 10.7: Vista de Bunch-NAHC sobre **caso-checkout**.

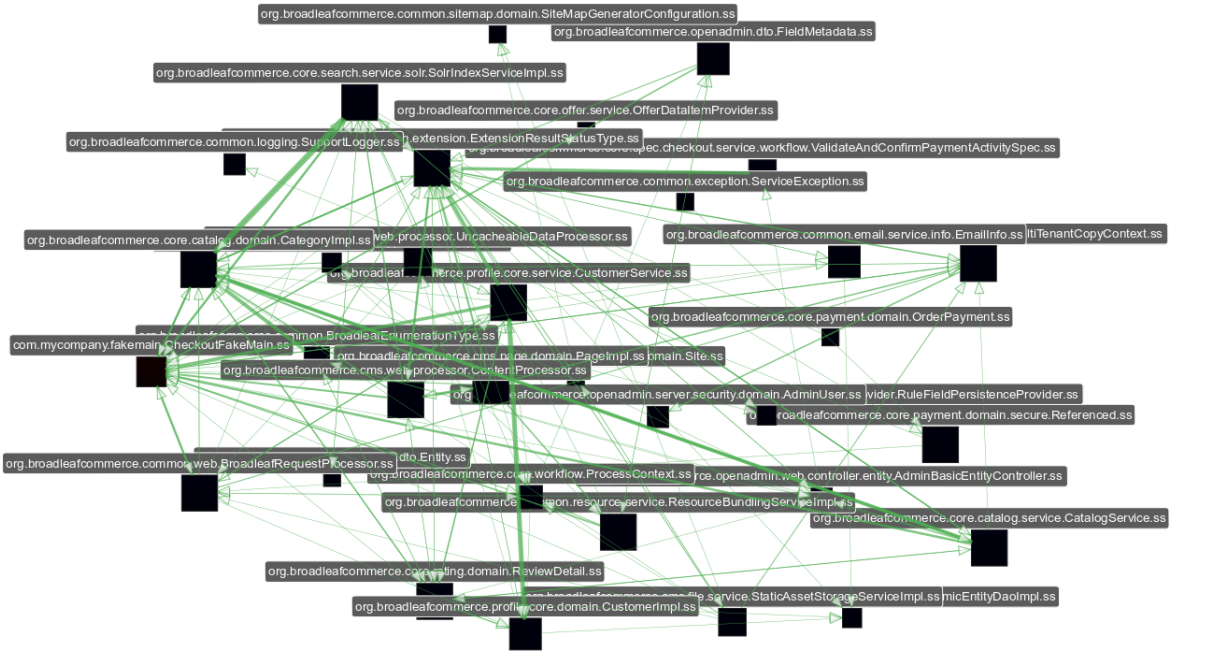


FIGURA 10.8: Vista de Bunch-NAHC sobre **caso-review**.

La primera técnica explorada es Bunch, un punto de interés que se observa a partir de las vistas es acerca de la cantidad de entidades que participan en cada agrupamiento. Si bien la tabla presentada en la sección 10.2 muestra la cantidad de llamados involucrados entre clusters para representar cada caso de uso, no se tiene en cuenta en que proporción los agrupamientos son utilizados, sin embargo las imágenes visualizadas brindan esta información a partir de representar en el tamaño de los cuadrados la cantidad

de entidades que son utilizadas por cada cluster. Podemos observar entonces que Bunch utiliza para describir cada caso de casi todos sus agrupamientos de manera mas o menos uniforme, probablemente indicando una abstracción dispersa.

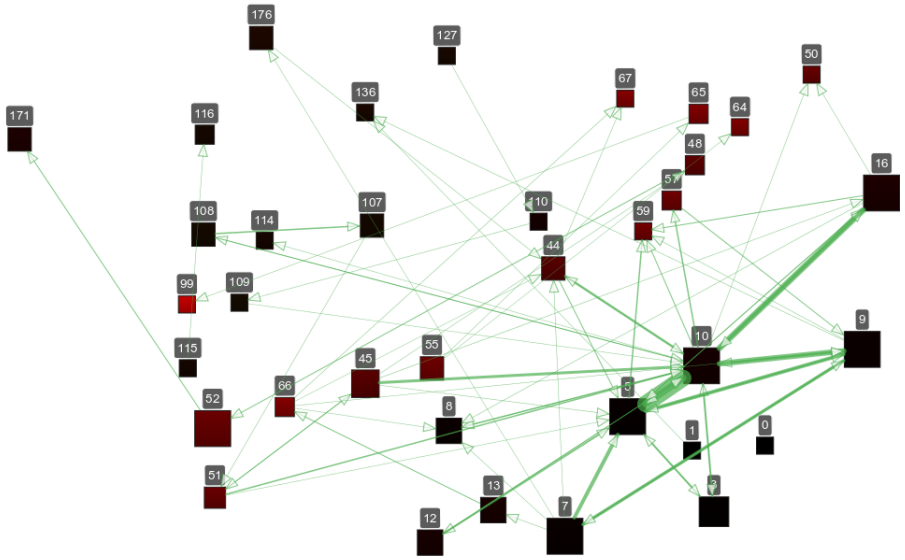


FIGURA 10.9: Vista de 0-CFA-F-25 sobre **caso-login**.

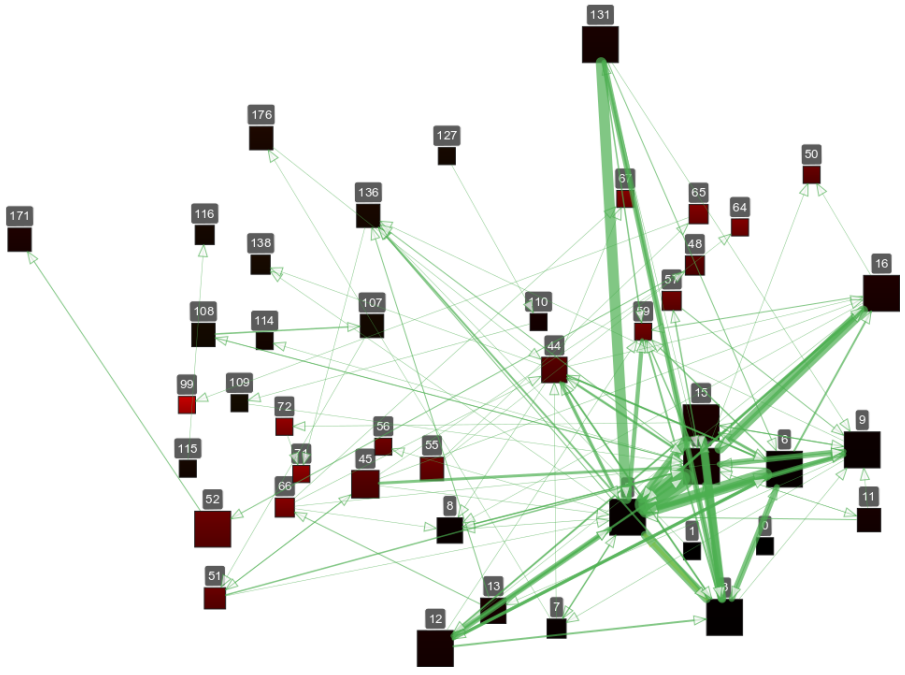


FIGURA 10.10: Vista de 0-CFA-F-25 sobre **caso-checkout**.

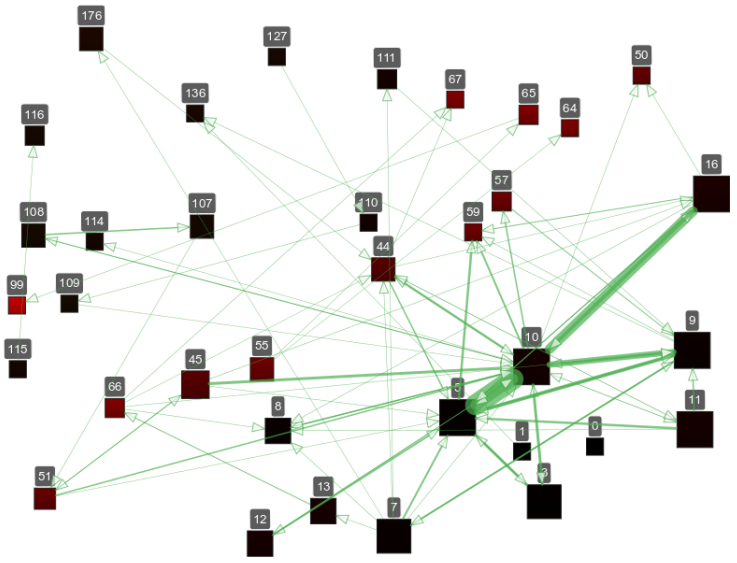


FIGURA 10.11: Vista de 0-CFA-F-25 sobre **caso-review**.

A partir de las visualizaciones obtenidas por 0-CFA-F-25 podemos observar otro extremo en la distribución de entidades, como se puede ver en la tabla presentada en 10.2 la cantidad de clusters involucrados en cada caso de uso es en promedio similar a la utilizada por Bunch, sin embargo el agrupamiento 0-CFA-F-25 notoriamente distribuye de manera no equitativa la participación de algunos clusters. Esto ayuda a evidenciar la participación de algunos agrupamientos según el caso de uso, por ejemplo en **caso-checkout** (figura 10.10) se puede ver como los agrupamientos 131, 15 y 6 entran en juego, así como también en **caso-review** (figura 10.11) el agrupamiento 11 toma mayor relevancia. Consideramos estas características como prometedoras y merecedoras de una vista ampliada.

Por otro lado, recordemos que tanto la modularidad como MoJoFM reportaban valores similares para Bunch y 0-CFA-F-25 de alguna manera indicando una representación similar, no obstante visualmente la diferencia resulta evidente. Es interesante ver que si existió una diferencia en la cantidad de llamados entre agrupamientos, despertando quizá oportunidades para diferenciar casos en los que la modularidad se encuentra en un rango cercano. Otro indicador notorio de diferencias cuyo análisis merece ser evaluado en trabajos futuros es la mencionada distribución de clases partícipes de la ejecución en los agrupamientos.

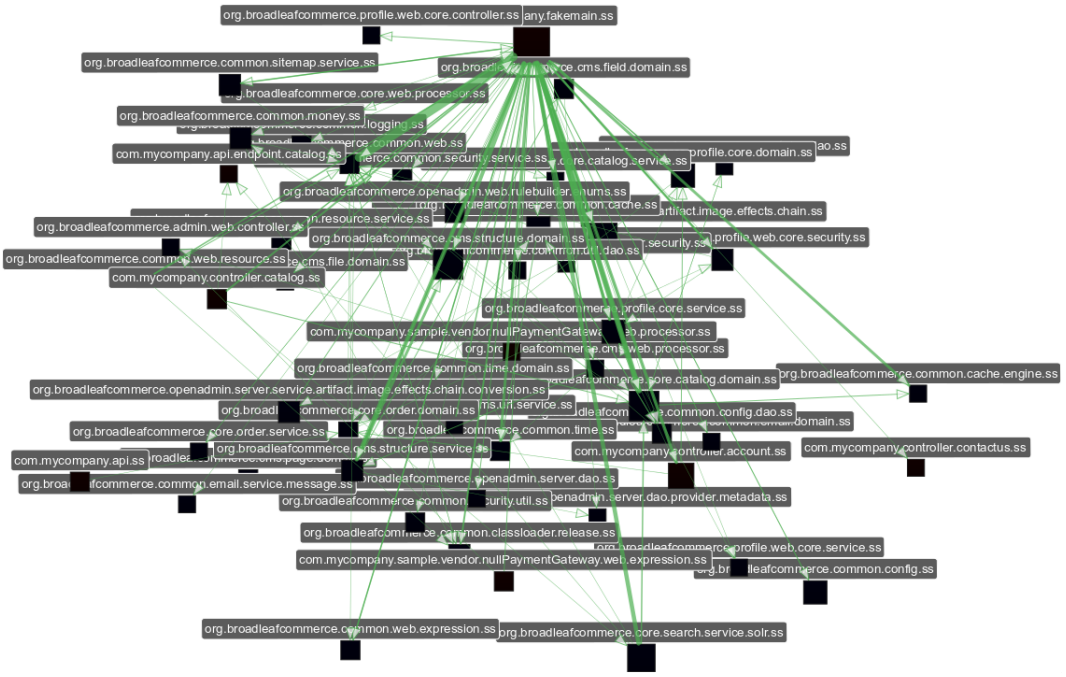


FIGURA 10.12: Vista de ACDC sobre **caso-login**.

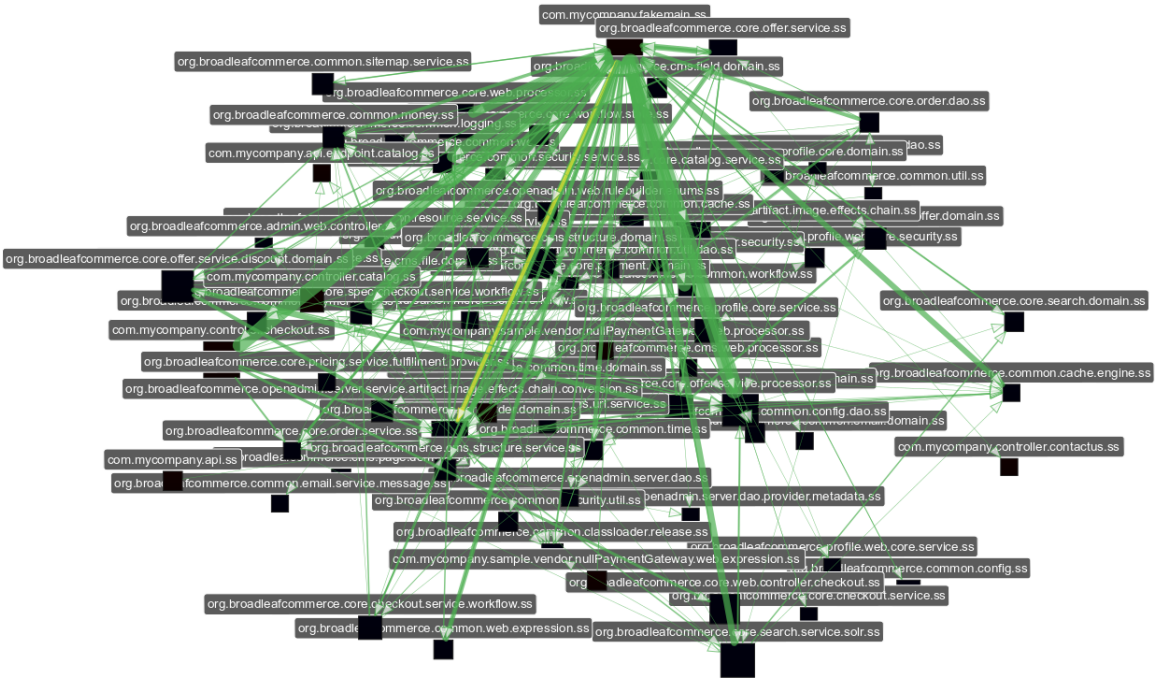


FIGURA 10.13: Vista de ACDC sobre **caso-checkout**.

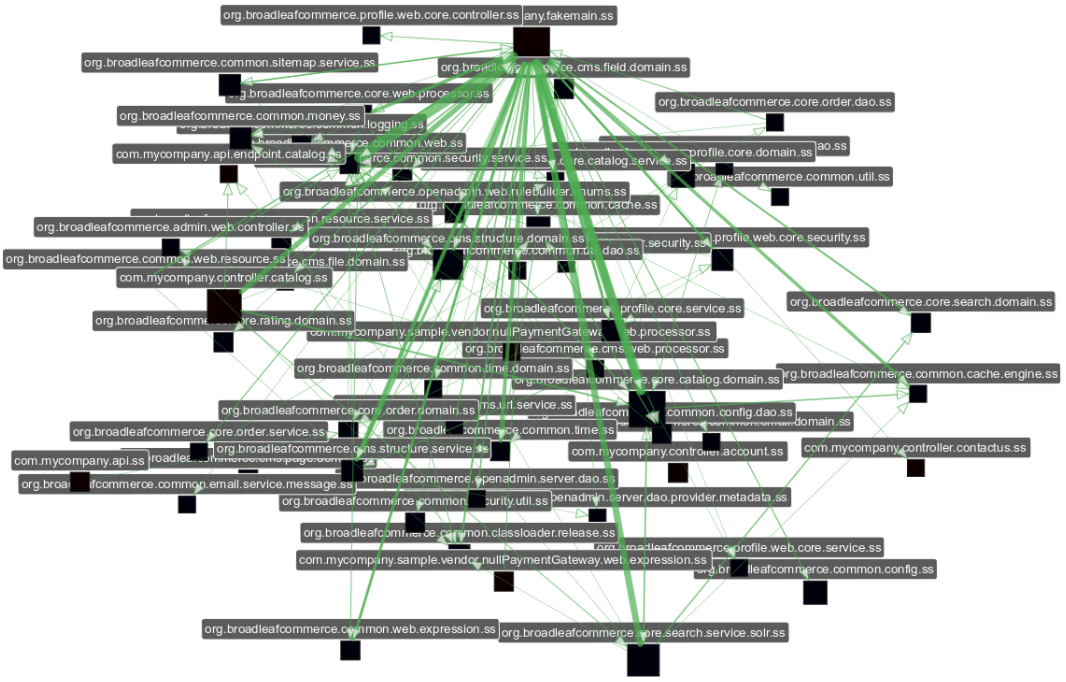


FIGURA 10.14: Vista de ACDC sobre **caso-review**.

Las vistas expuestas a partir de ACDC son un claro ejemplo en el que las tres métricas propuestas tienen un correlato cuando la descomposición evidentemente no retrata la vista buscada, incurriendo en una cantidad excesiva de agrupamientos y de relaciones entre si, dificultando enormemente la extracción de valor de la misma. Aún descartando componentes satélite (utilitarios) la vista provista por ACDC continúa manteniendo demasiadas relaciones entre componentes, y una relación muy baja de interacciones internas alejándose notoriamente de las vistas buscadas en el trabajo.

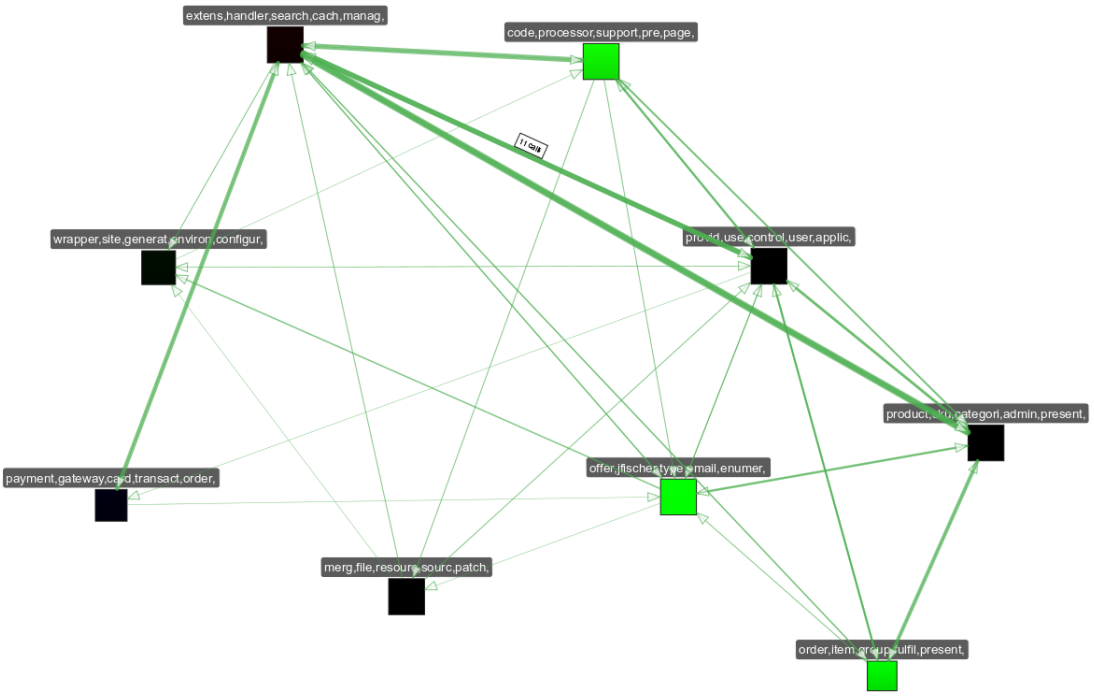


FIGURA 10.15: Vista de T-10-0 sobre **caso-login**.

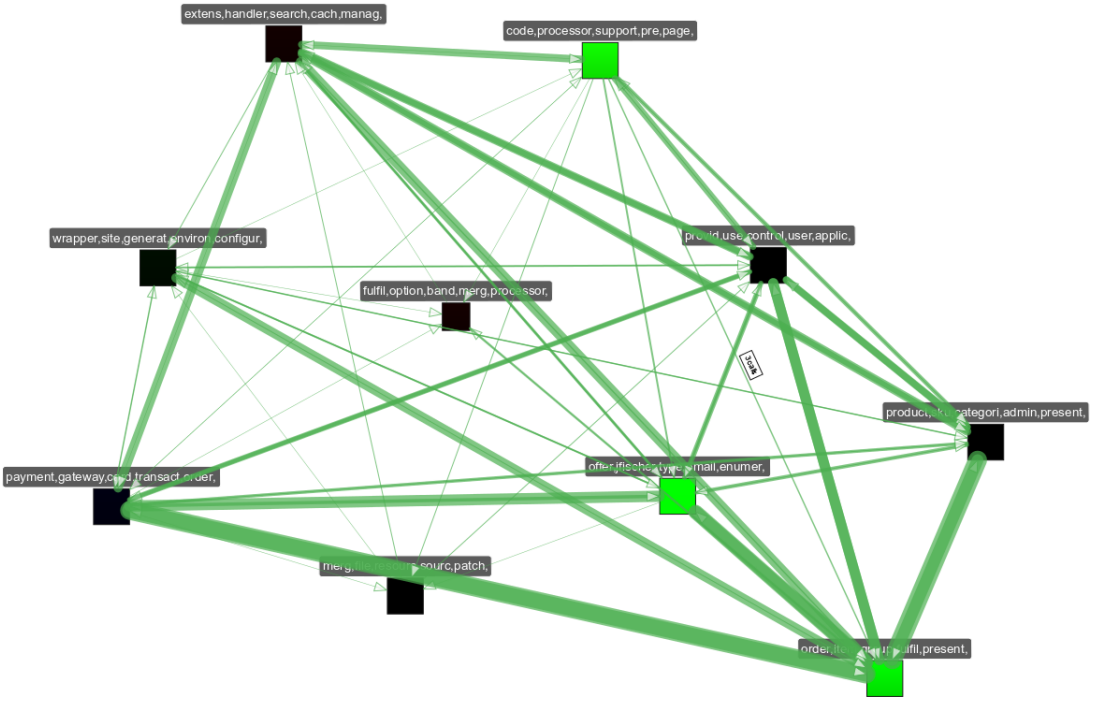


FIGURA 10.16: Vista de T-10-0 sobre **caso-checkout**.

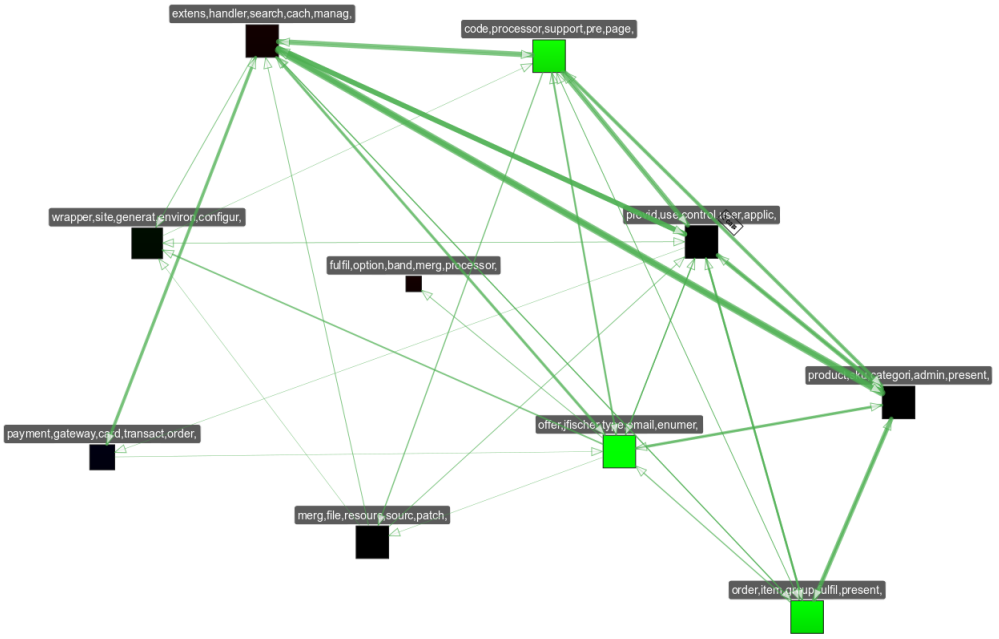


FIGURA 10.17: Vista de T-10-0 sobre **caso-review**.

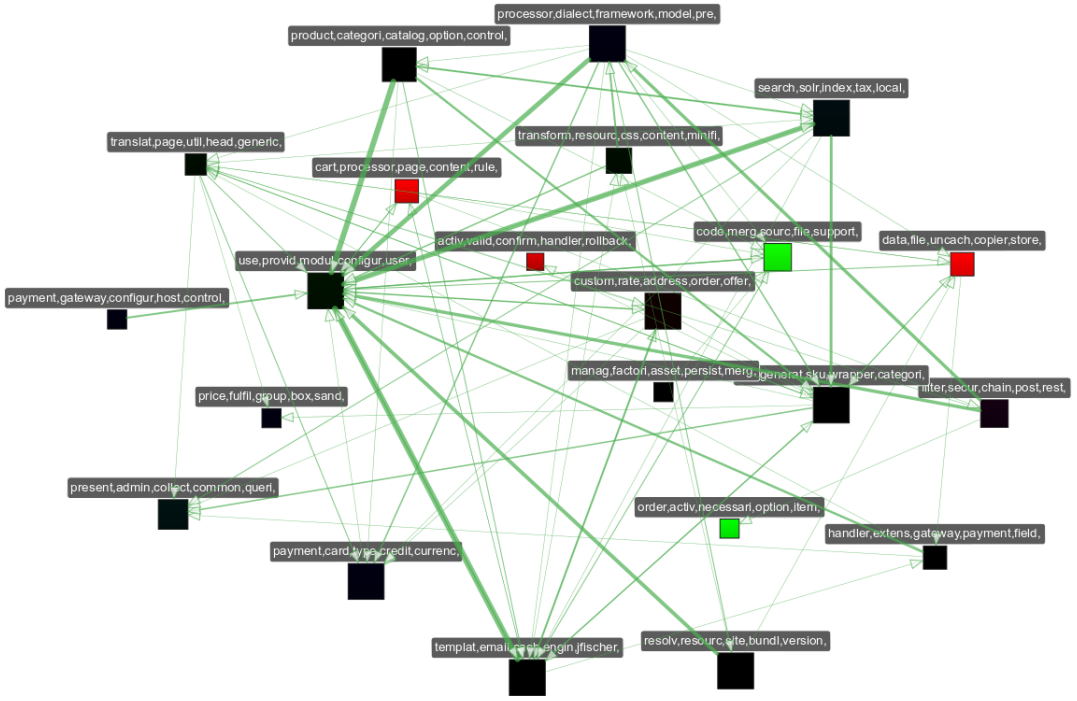


FIGURA 10.18: Vista de T-25-20 sobre **caso-login**.

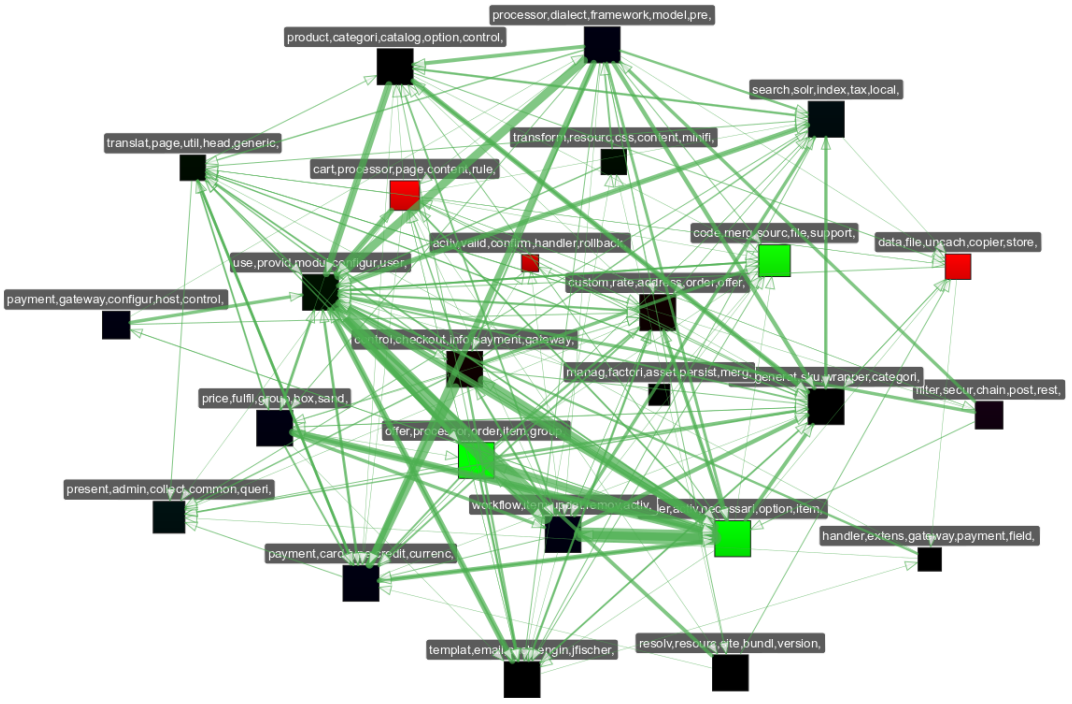


FIGURA 10.19: Vista de T-25-20 sobre **caso-checkout**.

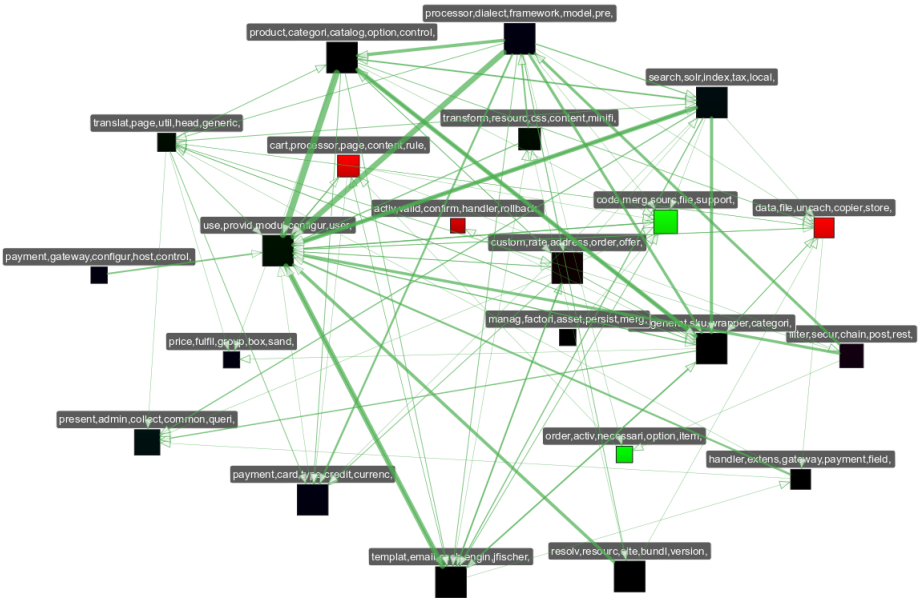


FIGURA 10.20: Vista de T-25-20 sobre **caso-review**.

Dentro de las vistas obtenidas por las técnicas T-10-0 y T-25-20 basadas en información léxica (capítulo 6), se puede apreciar claramente la diferencia de magnitud en la cantidad de relaciones entre componentes. También es posible apreciar como en T-10-0 casi no hay diferencia en los componentes involucrados y su tamaño según el caso de uso, pero si la cantidad de relaciones, brindando una intuición de falta de modularidad. Este caso permite ver como la cantidad de llamados entre agrupamientos no debe ser

tomado a la ligera ya que es posible ser víctima de casos en donde al ser reducida la cantidad de agrupamientos, la cantidad de llamados entre ellos es inherente-mente baja.

A su vez T-25-20 nos muestra la otra cara de la moneda, mostrando u ocultando varios componentes según el caso de uso, en algunos casos en su totalidad. Esta técnica obtuvo el mejor desempeño en general según MoJoFM, sin embargo según el resultado de modularidad su desempeño dista de ser óptimo (menor a 0.35), estas condiciones nos invitan a desglosar esta vista para observar más en detalle sus cualidades de presentación de la arquitectura.

10.3.3. Vistas desplegadas

A raíz de las propiedades observadas en las secciones previas mostraremos algunas vistas desplegadas de Bunch, 0-CFA-F-25 y T-25-20, con la finalidad de indagar en profundidad los atributos de cada reconstrucción y la posibilidad de las mismas para discernir componentes claros de tiempo de ejecución.

Para ello primero desglosaremos un poco los comportamientos del sistema esperados para dos casos de uso **caso-review** y **caso-completo** según los componentes sugeridos en la arquitectura de referencia.

Definición de **caso-review**:

1. Un usuario no registrado ingresa un texto en la barra de busqueda
2. Hace un ordenamiento por nombre
3. Ingresa a los detalles de un item
4. Elige la opción de ingresar para hacer un review
5. Ingresa un rating y el texto del review

Inicialmente nos imaginamos al menos cinco componentes razonables, estos son un componente para la resolución de la búsqueda que interactúe con los servicios provistos por Solr (mencionado en la sección 8.1.1), comunicado con un componente que maneje el catálogo de los productos y que se encargue del filtrado de los mismos, por otro lado un componente encargado de efectuar el login, otro de recuperar la información de un cliente y finalmente otro componente o sub-componente responsable de adquirir y grabar el review. Este caso es de interés ya que hace uso de componentes sin interactuar con las funcionalidades más profundas del sistema como la carga de órdenes.

Definición de **caso-checkout**:

1. Un usuario no registrado hace click en ofertas
2. Aplica filtros relacionados con la marca de la salsa
3. Ordena las publicaciones por precio
4. Entra en el detalle de una publicación

5. La agrega al carrito de compras
6. Entra al carrito
7. Modifica la cantidad del producto y procede al checkout
8. Se ingresa como invitado e Ingresa los datos de facturación y pagos
9. Termina el checkout

El checkout de una orden es sin dudas el caso de uso más complejo abordado en este trabajo, repitiendo el ejercicio mental podríamos separar las responsabilidades en al menos cinco componentes, repetiríamos la interacción con un componente responsable del catálogo de ofertas y el filtrado, luego a patrir de los detalles de una publicación activaríamos el componente de promociones. Los eventos: agregar un producto al carrito, modificar su cantidad, proceso de checkout, pago y envío son parte de los ya mencionados workflows de la aplicación (sección 8.1.2) y componen la parte más compleja del sistema analizado, por lo tanto dar precisiones sobre cuantos componentes podrían integrar es más complejo que para otros comportamientos del sistema. Propondremos al menos una división en dos componentes, uno responsable de la confección de la orden (agregar los ítems, actualizarlos, etc) y otro encargado del checkout (carga de datos y proceso de pago). Por último para realizar una compra es necesario al menos ingresar al sistema como invitado por lo que el componente de login y manejo de usuarios debería registrar alguna actividad.

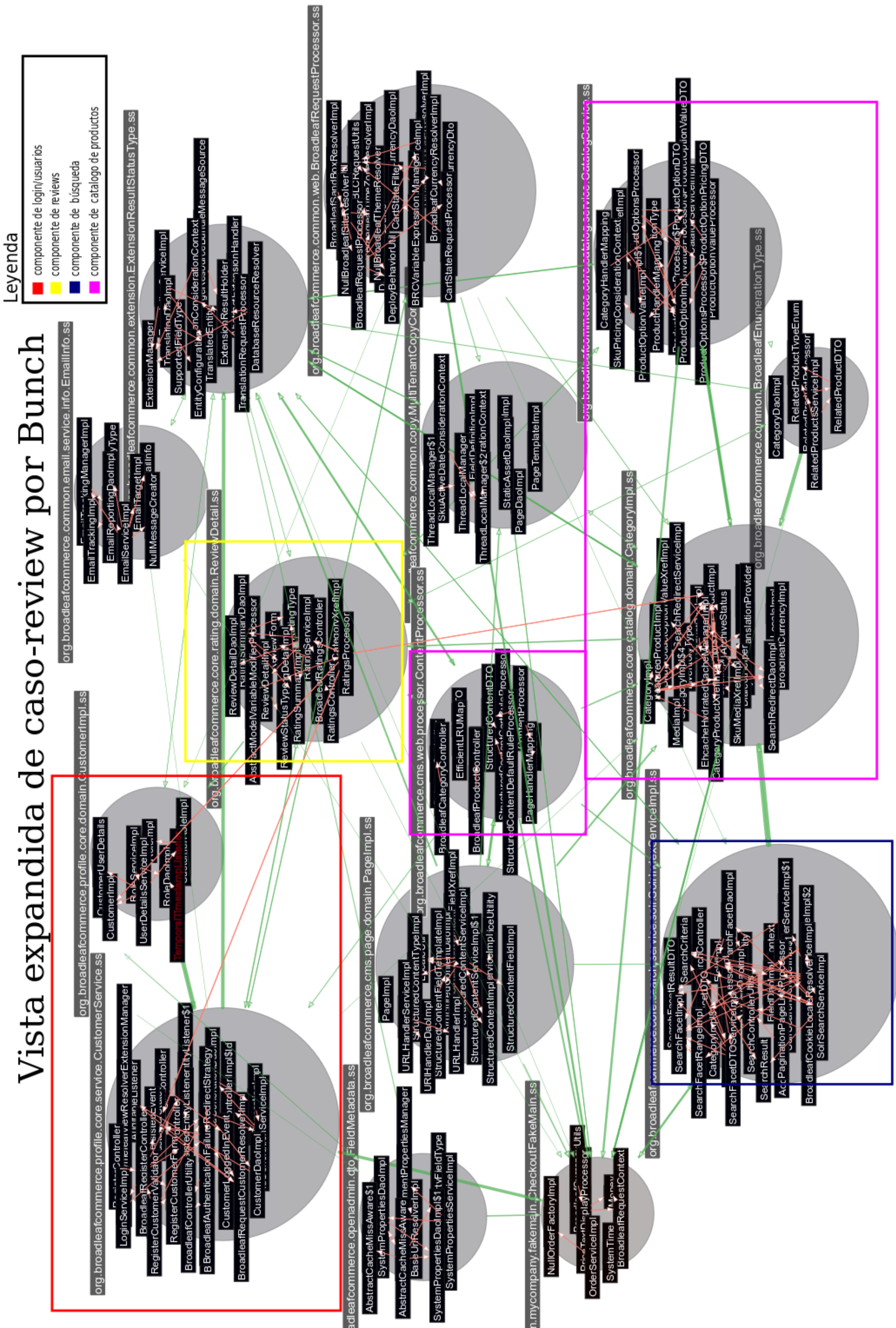
En contraste con el **caso-review**, no se debería incurrir en demasiada actividad alrededor del componente de búsquedas, manejo de usuarios o adquisición de reviews.

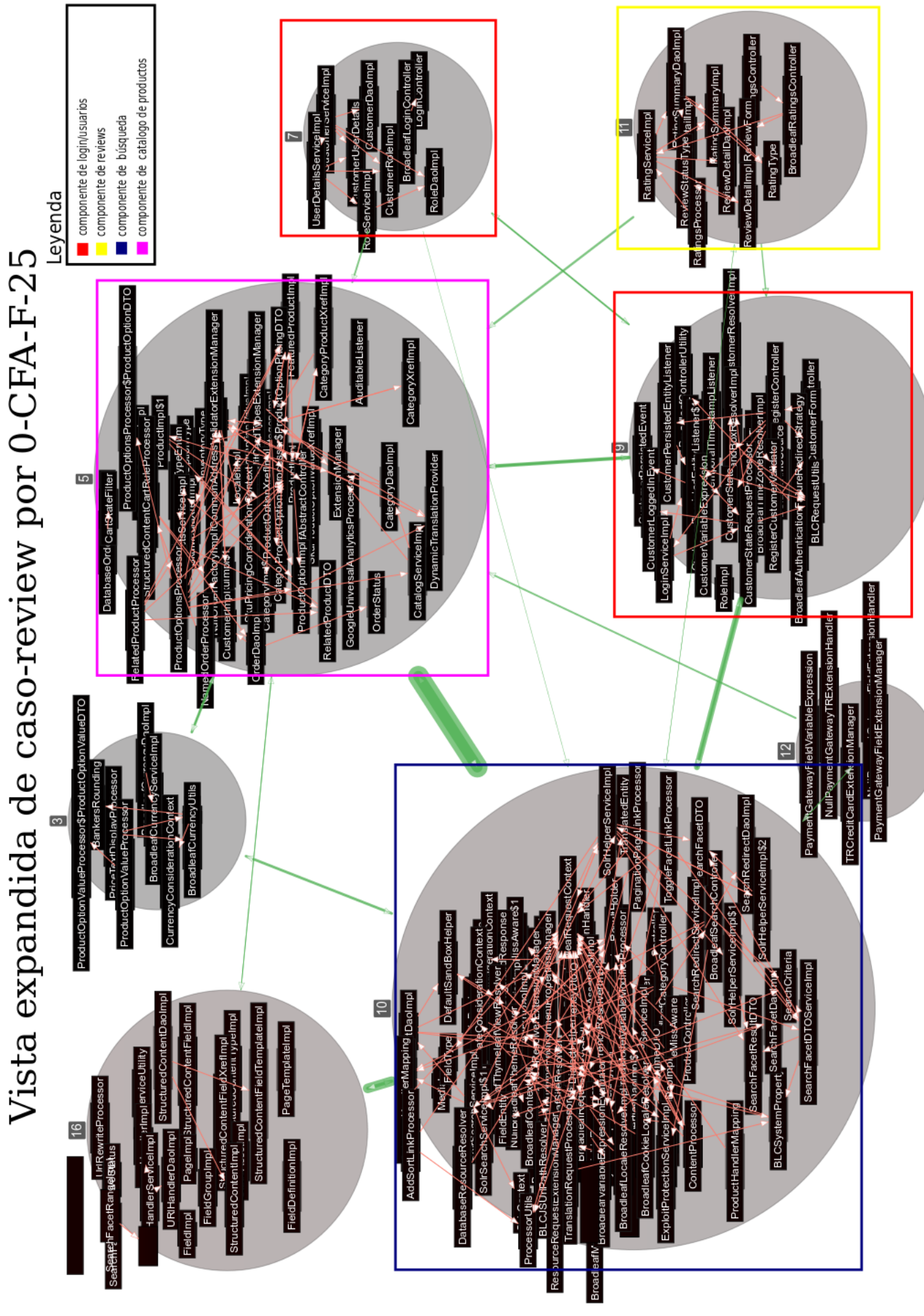
Por otro lado no debemos olvidar que las interacciones filtradas en la sección 9.1.1 sobre las interacciones internas con otros frameworks deberían representarse en estas vistas como interacciones con las interfaces de componentes tecnológicos tales como base de datos, motor de vistas o caché.

Intentaremos contrastar entonces esta aproximación de componentes con el detalle de las reconstrucciones obtenidas por Bunch y 0-CFA-F-25 y reportaremos las características distintivas de los mismos. En el proceso de despliegue nos encontramos en ambos casos con agrupamientos de tamaño pequeño con poca interacciones entre varios clusters, estos fueron removidos de las vistas ya que se debían en su mayor parte a utilitarios que no aportaban información sustancial al comportamiento entre componentes, es importante destacar que su remoción tiene un impacto despreciable sobre la modularidad (+-0.01) y mantiene la relación relativa en la cantidad de llamados entre clusters.

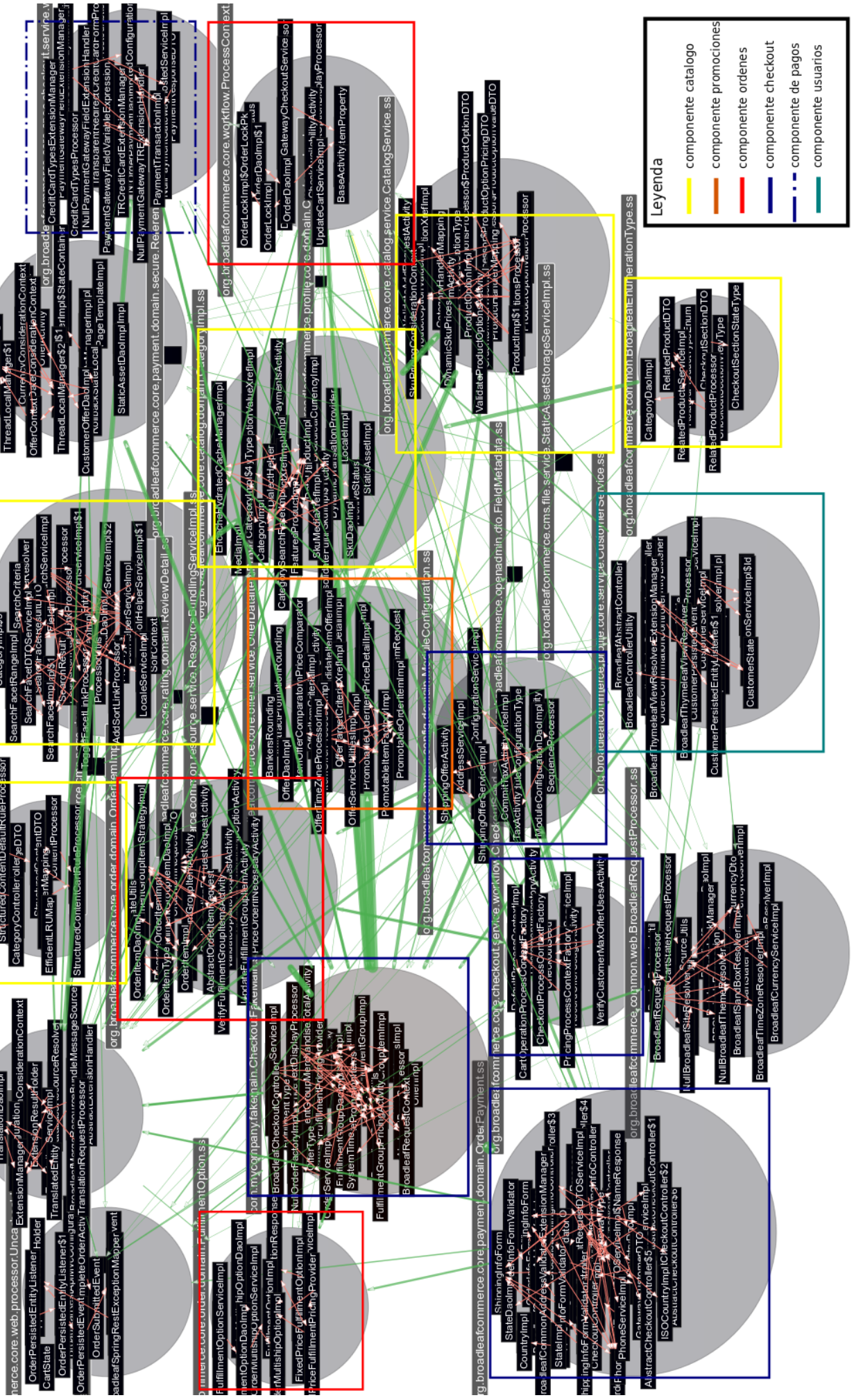
Visualizaremos entonces las reconstrucciones desplegadas para los casos de uso mencionados en esta sección, cada agrupamiento expandido se representa con un círculo y sus interacciones internas con flechas dirigidas de color naranja. Además sus entidades se representan por medio de rectángulos negros en donde se indican sus nombres.

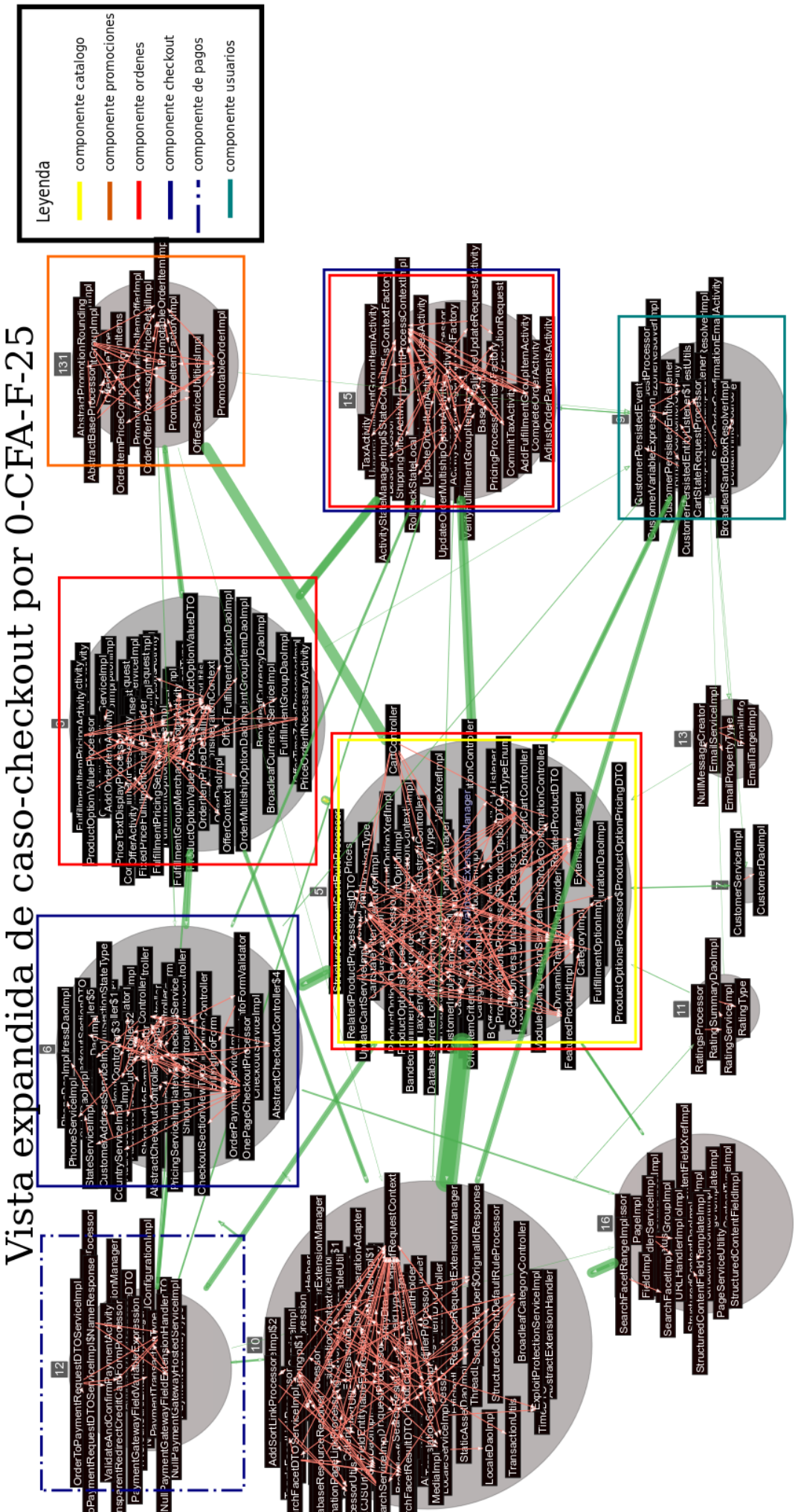
Además indicaremos sobre las vistas la presencia en los agrupamientos de los elementos pertenecientes a los componentes sugeridos mediante cuadrados diferenciados por color, explicados en la leyenda de los gráficos.


Vista expandida de caso-review por Bunch
Leyenda
componente de login/usuarios
componente de reviews
componente de búsqueda
componente de catalogo de productos


Vista expandida de caso-review por 0-CFA-F-25
Leyenda
componente de login/usuarios
componente de reviews
componente de búsqueda
componente de catalogo de productos
7
5
11
9
3
12
10
16

Vista expandida de caso-checkout por Bunch

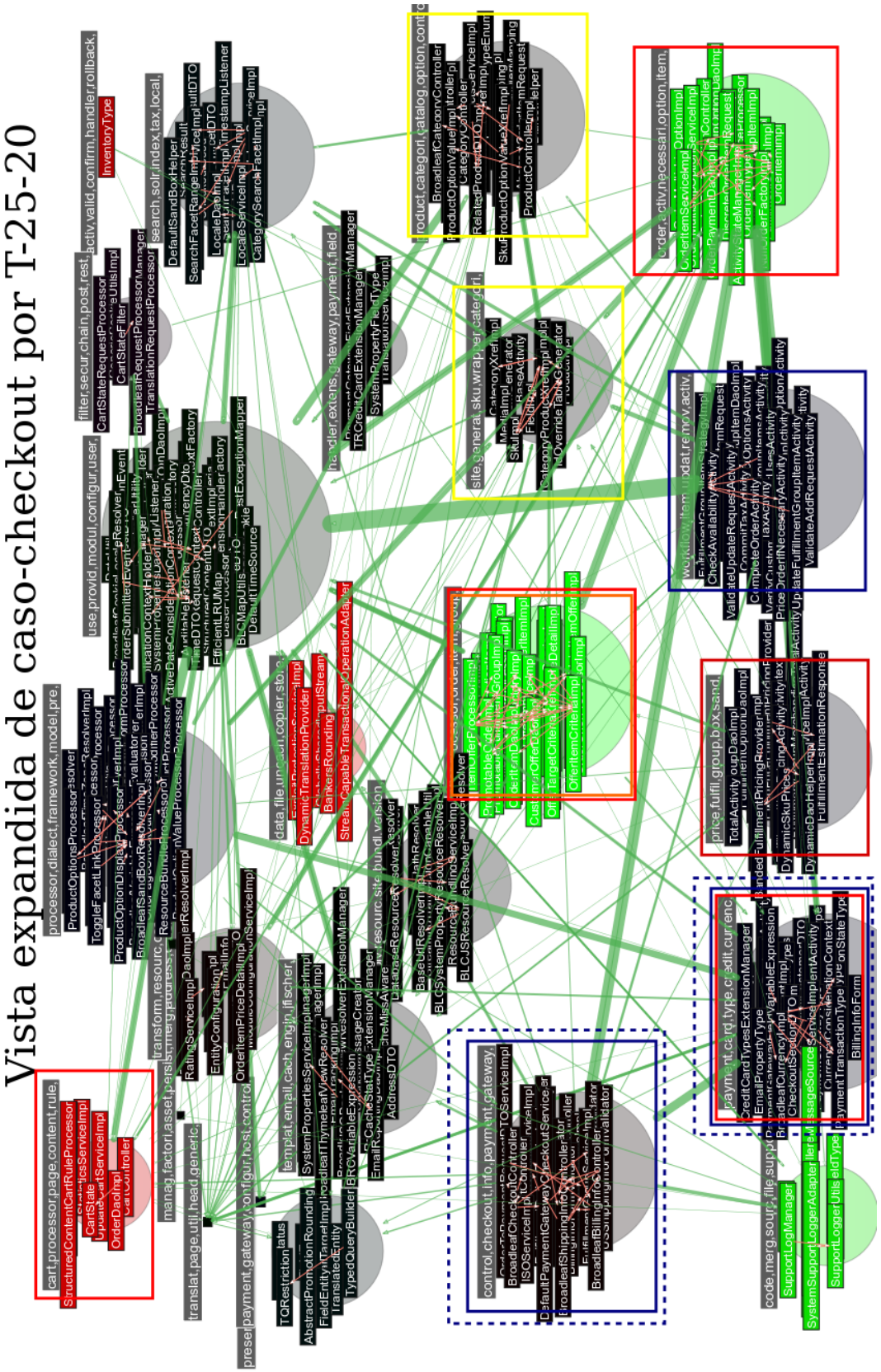



Vista expandida de caso-checkout por 0-CFA-F-25
Leyenda
componente catalogo
componente promociones
componente ordenes
componente checkout
componente de pagos
componente usuarios

Sobre la vista de Bunch desplegada en el **caso-review** y **caso-checkout** se pueden observar varios fenómenos, el primero se evidencia en la fragmentación de componentes de negocio y tecnológicos, es fácil ver como bunch requiere de más de un agrupamiento para concentrar la lógica responsable de resolver cierta parte específica del problema. Por otro lado, estas fragmentaciones provocan una densidad notable de interacciones entre componentes volviendo a los diagramas complejos de entender en términos de componentes, sin embargo es posible en el caso más simple (**caso-review**) discernir algunas interacciones esperadas, entre ellas las relaciones entre el componente de review y el componente de usuarios, el componente de review y el componente de catálogo y el componente de búsqueda y el componente de catálogo.

Por otro lado las vista producidas por 0-CFA-F-25 muestran la otra cara de la moneda, aquí en vez de fragmentarse lo componentes tienden a solaparse, mientras que las interacciones entre componentes se contraen considerablemente, esto si bien brinda vistas más compactas produce interacciones más complicadas cuando componentes tratan sobre diferentes propósitos, este es el caso del agrupamiento 10 que en ambos escenarios condensa casi todos los componentes tecnológicos junto al componente de búsquedas. Otro comportamiento interesante surge de la vista **caso-checkout** donde los agrupamientos 3 y 15 conglomeran las actividades participantes de los principales workflows del sistema (mencionadas en la sección 8.1.2), esto evidencia una falencia indirecta de la modularidad provocada por la imprecisión propia de la abstracción de clases (análoga al ejemplo visto en la sección 6.2) ya que en tiempo de ejecución las actividades propias de un workflow deberían relacionarse con instancias particulares de una misma clase, representadas por un único nodo en nuestro diagrama.

Si observamos la vista desglosada de **caso-checkout** producida por T-25-20 apreciaremos una modularidad mucho menor a Bunch y 0-CFA-F-25, es decir una cantidad de llamados intra-componentes mucho más baja en relación a la inter-componentes, dificultando muchísimo la obtención de valor a partir de esta representación. Este es un caso claro en donde MoJoFM no representa correctamente las falencias de un agrupamiento.


Vista expandida de caso-checkout por T-25-20

10.4. Discusión

A partir de las evaluaciones realizadas en las técnicas de reconstrucción hemos ofrecido un pantallazo sobre el problema de descomponer un sistema en componentes de negocio, como ejes de discusión general podemos plantear si hemos encontrado o no vistas cercanas a las deseadas inicialmente (**RQ(2)**) y si las mismas son correctamente evaluadas por las métricas propuestas.

En cuanto al primer interrogante podemos plantear que si bien las dos representaciones que reportaron mejor desempeño en general (Bunch y 0-CFA-F-25) presentan irregularidades tales como fragmentación o solapamiento, en ambas es posible observar al menos un componente diferenciado e interacciones razonables según nuestro entendimiento del sistema, por lo que consideramos que representa un primer paso positivo en el camino de la reconstrucción de vistas arquitectónicas de comportamientos.

Continuando en este plano es merecedor de una mención la clara importancia en la precisión de los algoritmos de construcción de grafos de llamada para la construcción de vistas cercanas al comportamiento en tiempo de ejecución, motivando un trabajo futuro en la mejora en los métodos y algoritmos evaluados en esta tesis.

En cuanto a las propiedades y distancias utilizadas para comparar los agrupamientos encontramos las propuestas a partir de ejecuciones como prometedoras a pesar de ciertas falencias dado que no requieren en principio de la interacción por parte de un experto y que sus resultados ayudaron a discriminar casos donde la validación clásica arrojaba resultados engañosos (caso T-25-20). No obstante la elección de la abstracción y sus características resultan determinantes para tomar las medidas propuestas como un parámetro comparador de vistas, esta dimensión lleva a pensar trabajos futuros alrededor de modificaciones en las medidas combinando información de precisión de las abstracciones.

Capítulo 11

Limitaciones a la validez

La principal limitación en cuanto a la validez de las evaluaciones y resultados expuestos en el presente trabajo se engloba en la utilización de una única aplicación de prueba, ya que si bien la misma poseé características de interés como su tamaño, y la interacción de varios frameworks para su funcionamiento, por si sola resulta insuficiente para afirmar una generalización de los resultados debido a que la misma podría presentar un sesgo según asociado por ejemplo al dominio (aplicaciones e-commerce), diseños y abstracciones elegidos por los programadores o incluso el uso (o no uso) particular de ciertas capacidades de los frameworks.

Por otro lado, el armado de la arquitectura de referencia a partir de la experiencia y entendimiento adquirido en el sistema junto con la exploración manual de las vistas presentan un sesgo propio de la interacción con personas que generalmente se reduce por medio de la participación de arquitectos o programadores con un alto entendimiento del sistema aunque esto a su vez trae aparejadas otras limitaciones como se menciona en la sección 2.3 de los antecedentes del trabajo.

A su vez la remoción de las interacciones propias de otros frameworks no soportados por el análisis presentado en este trabajo (sección 9.1.1) implica una limitación clara en cuanto a los componentes e interacciones posibles de ser representados por las vistas y la posterior evaluación de las mismas. En esta misma línea se encuentra la exclusión de la participación de librerías en la vista arquitectónica, cuyo impacto no es medido y seguramente varíe según la aplicación y su articulación sobre las mismas.

Capítulo 12

Conclusiones y trabajos futuros

Durante este trabajo hemos presentado un recorrido sobre la reconstrucción de vistas arquitectónicas de componentes de negocio en aplicaciones basadas en Spring IoC y Spring MVC, presentando un método de obtención de dependencias en runtime, dos técnicas de reconstrucción, dos propiedades para comparar la calidad de las vistas independientes de un experto en el sistema y una evaluación de todas estas contribuciones sobre un caso de estudio real.

A partir de lo observado, se puede afirmar que tanto cada área en este recorrido como la integración de las mismas fue exitosa, pudiendo obtener vistas con características afines a las buscadas, sin embargo todavía se presentan varios desafíos que pasaremos a detallar a continuación.

Sobre la comparación de vistas (capítulo 10) podemos ver una correlación para ciertos grupos de reconstrucciones según las distancias MoJoFM a la arquitectura de referencia y las propiedades de comparación introducidas en este trabajo (modularidad y cantidad de llamados entre componentes) que brinda una validación interesante sobre el tipo de vista buscada en el trabajo. En cuanto a las diferencias entre las mismas encontramos dos fenómenos para destacar, estos son: la alta premiación de distancias MoJoFM para algunos agrupamientos cuyas características de interacción no son ideales (caso T-25-20), posiblemente causado por la importancia en la cantidad de clusters o la calidad de la arquitectura de referencia y por otro lado, reconstrucciones cuya modularidad es razonable y cantidad de llamados entre agrupamientos mínima (caso 0-CFA-F-25) que desemboca en el solapamiento entre componentes e interacciones que a nuestro entender deberían presentarse como componentes separados. Estos fenómenos de la comparación de vistas dan el puntapié inicial a trabajos futuros para la construcción de medidas y combinaciones más precisas independientes de interacciones con expertos y eventualmente sean adaptables a los tipos de vistas buscados.

Avanzando sobre las técnicas propuestas en el capítulo 6 podemos ver una clara ventaja en modularidad de la técnica estructural sobre la técnica basada en información

léxica, sin embargo como vimos en las evaluaciones existen fenómenos propios del diseño del software y la representación elegida (clases) cuyo impacto puede ser mayor si es atacado por técnicas de información estructural que quizá podrían ser evitados por técnicas basadas en información léxica. Esto a nuestro entender abre la puerta a varios trabajos futuros como por ejemplo un cambio de representación parcial de clases a objetos (permanente o bajo demanda), la confección de agrupamientos con varios niveles jerárquicos en donde creemos que el valor propio de las técnicas basadas en información léxica podría brindar un valor agregado para discernir subcomponentes según el dominio en el que trabajen, y también el estudio de otros diseños o patrones de diseño y el impacto que los mismos podrían tener según la abstracción elegida para la reconstrucción.

Un ejemplo claro en este sentido es el efecto causado por los componentes utilitarios, representados como satélites pequeños con varias conexiones hacia componentes de diferente naturaleza y entorpeciendo una visualización nítidas en algunas de las vistas presentadas en este trabajo.

Si hacemos foco en el método de obtención de dependencias presentado en el capítulo 5, podemos ver en las evaluaciones del capítulo 9 que si bien a los fines del trabajo presentado los resultados de precisión resultan satisfactorios, existen varias aristas disponibles para recorrer producto de complejidades tecnológicas propuestas por los frameworks y los lenguajes de programación involucrados, principalmente el soporte de frameworks cuya interacción con la aplicación es diferente a IoC o MVC tales como AOP o Hibernate (involucrados en la aplicación de prueba) e incluso la búsqueda de una implementación con un soporte más global a los conceptos fundamentales de los frameworks y el soporte de construcciones más avanzadas propias de versiones nuevas tanto del framework como del lenguaje de programación.

A su vez, un punto importante del trabajo estuvo en entender el impacto que tiene la precisión de cada algoritmo de grafo de llamadas y el análisis del framework con respecto a la calidad de reconstrucción arquitectónica obtenida, concluyendo que su precisión resulta clave para la misma y por ende brindando otra línea de trabajo alrededor de lograr la posible ejecución de algoritmos y análisis más precisos.

Dejando de lado las cuestiones de nivel más bajo, una línea muy amplia de investigación que surge alrededor de este tipo de vistas consiste en como entender y sacar valor de las mismas ya sea como documentación (mencionado en antecedentes) o herramienta de análisis, verificación o validación de características globales de la aplicación o bien integradas al entorno de desarrollo. En la actualidad tanto las herramientas de visualización como los estudios de comprensión de software se rigen sobre estudios con usuarios, habitualmente programadores, en los que se plantean tareas y se documentan los procesos de solución, comunicación e interacción con los programas (o vistas) para entender que aspectos de los mismos resultan útiles y en donde todavía hay espacio para mejoras. Creemos que una investigación en este sentido podría brindar claridad para

aprender que características resultan las más atractivas en la construcción de vistas de runtime.

Siguiendo esta línea cabe destacar además que el foco de este trabajo radica particularmente en encontrar reconstrucciones de tiempo de ejecución en las cuales se evidencien componentes de negocio, esta descomposición es una entre varias posibles elecciones de componentes como pueden ser tecnológicos o de despliegue. El estudio de como obtener vistas más afines a una u otra representación, y el impacto que pueden tener entre si según el diseño del sistema representa un área de trabajo inmensa en la cual pueden surgir nuevas herramientas de visualización, análisis y métricas de comparación de naturaleza muy diferente a las que se presentan en este trabajo.

Por último, merece un párrafo aparte la visualización cuyo papel resulto esencial en la última etapa de evaluación del trabajo y cuyas características o posibilidades cayeron fuera del alcance del trabajo pero representaron la condensación final del valor en el recorrido de la reconstrucción arquitectónica. Creemos que esta área de investigación deberá ser explorada en trabajos posteriores donde cambios en las abstracciones podrían significar desafíos en las herramientas de visualización, así como la inserción de librerías, la capacidad de mostrar la interacción de varios frameworks en la aplicación e incluso la vista de una reconstrucción de varios niveles.

Bibliografía

[1] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger, "Comparing software architecture recovery techniques using accurate dependencies,"

[2] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav, "Alias analysis for object-oriented programs," in *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pp. 196–232, Springer, 2013.

[3] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," *Foundations and Trends® in Programming Languages*, vol. 2, no. 1, pp. 1–69, 2015.

[4] R. Minelli, A. Mocci, M. Lanza, and T. Kobayashi, "Quantifying program comprehension with interaction data," in *Quality Software (QSIC), 2014 14th International Conference on*, pp. 276–285, IEEE, 2014.

[5] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, "On the comprehension of program comprehension," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, p. 31, 2014.

[6] T. D. LaToza and B. A. Myers, "Developers ask reachability questions," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 185–194, ACM, 2010.

[7] B. Dagenais, H. Ossher, R. K. Bellamy, M. P. Robillard, and J. P. De Vries, "Moving into a new software project landscape," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 275–284, ACM, 2010.

[8] B. J. Berger, K. Sohr, and R. Koschke, "Extracting and analyzing the implemented security architecture of business applications," in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pp. 285–294, IEEE, 2013.

[9] A. R. Yazdanshenas and L. Moonen, "Tracking and visualizing information flow in component-based systems," in *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pp. 143–152, IEEE, 2012.

[10] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little, *Documenting software architectures: views and beyond*. Pearson Education, 2002.

[11] J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic, "Obtaining ground-truth software architectures," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 901–910, IEEE Press, 2013.

[12] M. Waterman, J. Noble, and G. Allan, "How much up-front? a grounded theory of agile architecture," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1, pp. 347–357, IEEE, 2015.

[13] V. Tzerpos and R. C. Holt, "Acdc: An algorithm for comprehension-driven clustering," in *wcre*, p. 258, IEEE, 2000.

[14] O. Maqbool, H. Babri, *et al.*, "Hierarchical clustering for software architecture recovery," *Software Engineering, IEEE Transactions on*, vol. 33, no. 11, pp. 759–780, 2007.

[15] A. Corazza, S. Di Martino, V. Maggio, and G. Scanniello, "Investigating the use of lexical information for software system clustering," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pp. 35–44, IEEE, 2011.

[16] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *Software Engineering, IEEE Transactions on*, vol. 32, no. 3, pp. 193–208, 2006.

[17] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pp. 552–555, IEEE Computer Society, 2011.

[18] J. Wu, A. E. Hassan, and R. C. Holt, "Comparison of clustering algorithms in the context of software evolution," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pp. 525–535, IEEE, 2005.

[19] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pp. 486–496, IEEE, 2013.

[20] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *ACM SIGPLAN Notices*, vol. 49, pp. 259–269, ACM, 2014.

[21] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg, "F4f: taint analysis of framework-based web applications," *ACM SIGPLAN Notices*, vol. 46, no. 10, pp. 1053–1068, 2011.

[22] H. Tang, X. Wang, L. Zhang, B. Xie, L. Zhang, and H. Mei, "Summary-based context-sensitive data-dependence analysis in presence of callbacks," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 83–95, ACM, 2015.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[24] N. Shi, R. Olsson, *et al.*, "Reverse engineering of design patterns from java source code," in *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pp. 123–134, IEEE, 2006.

[25] A. Corazza, S. Di Martino, and G. Scanniello, "A probabilistic based approach towards software system clustering," in *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pp. 88–96, IEEE, 2010.

[26] V. Tzerpos and R. C. Holt, "The orphan adoption problem in architecture maintenance," in *Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on*, pp. 76–82, IEEE, 1997.

[27] P. Andritsos and V. Tzerpos, "Information-theoretic software clustering," *Software Engineering, IEEE Transactions on*, vol. 31, no. 2, pp. 150–165, 2005.

[28] G. Scanniello, A. D'Amico, C. D'Amico, and T. D'Amico, "Using the kleinberg algorithm and vector space model for software system clustering," in *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pp. 180–189, IEEE, 2010.

[29] R. A. Bittencourt and D. D. S. Guerrero, "Comparison of graph clustering algorithms for recovering software architecture module views," in *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pp. 251–254, IEEE, 2009.

[30] P. Pons and M. Latapy, "Computing communities in large networks using random walks.," *J. Graph Algorithms Appl.*, vol. 10, no. 2, pp. 191–218, 2006.

[31] Z. Wen and V. Tzerpos, "An effectiveness measure for software clustering algorithms," in *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*, pp. 194–203, IEEE, 2004.

[32] M. Shtern and V. Tzerpos, "Refining clustering evaluation using structure indicators," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pp. 297–305, IEEE, 2009.

[33] A. Lienhard, S. Ducasse, T. Gırba, and O. Nierstrasz, "Capturing how objects flow at runtime," in *Proceedings International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006)*, pp. 39–43, 2006.

[34] O. Lhoták, "Spark: A flexible points-to analysis framework for Java," Master's thesis, McGill University, December 2002.

[35] L. O. Andersen, *Program analysis and specialization for the C programming language*. PhD thesis, 1994.

[36] A. Feldthaus, M. Schafer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for javascript ide services," in *Software Engineering (ICSE), 2013 35th International Conference on*, pp. 752–761, IEEE, 2013.

[37] M. Sharir and A. Pnueli, *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences. ComputerScience Department, 1978.

[38] O. Shivers, *Control-flow analysis of higher-order languages*. PhD thesis, Citeseer, 1991.

[39] D. Grove and C. Chambers, "A framework for call graph construction algorithms," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 23, no. 6, pp. 685–746, 2001.

[40] F. Tip and J. Palsberg, *Scalable propagation-based call graph construction algorithms*, vol. 35. ACM, 2000.

[41] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, "Introspective analysis: context-sensitivity, across the board," in *ACM SIGPLAN Notices*, vol. 49, pp. 485–495, ACM, 2014.

[42] A. Feldthaus, M. Schafer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for javascript ide services," in *Software Engineering (ICSE), 2013 35th International Conference on*, pp. 752–761, IEEE, 2013.

[43] Y. Smaragdakis, G. Kastrinis, G. Balatsouras, and M. Bravenboer, "More sound static handling of java reflection," tech. rep., Tech. Rep, 2014.

[44] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 241–250, ACM, 2011.

[45] K. Ali and O. Lhoták, "Application-only call graph construction," in *ECOOP 2012–Object-Oriented Programming*, pp. 688–712, Springer, 2012.

[46] R. Johnson, J. Hoeller, K. Donald, C. Sampaleanu, R. Harrop, T. Risberg, A. Arendsen, D. Davison, D. Kopylenko, M. Pollack, *et al.*, "The spring framework, reference documentation.",," 2004.

[47] R. Johnson, J. Hoeller, A. Arendsen, and R. Thomas, *Professional Java Development with the Spring Framework*. John Wiley & Sons, 2009.

[48] M. E. Newman, "Modularity and community structure in networks," *Proceedings of the national academy of sciences*, vol. 103, no. 23, pp. 8577–8582, 2006.