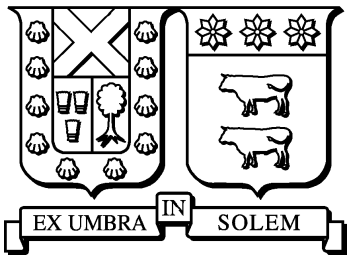
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA

DEPARTAMENTO DE INFORMÁTICA

SANTIAGO – CHILE




# "MEJORAMIENTO DE LA MANTENIBILIDAD Y EXTENSIBILIDAD DE UNA HERRAMIENTA DE GENERACIÓN DE MALLAS VOLUMÉTRICAS"

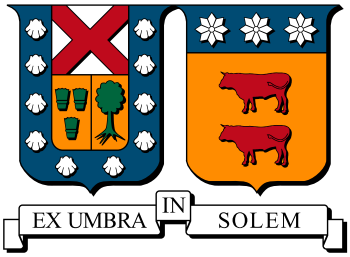
SEBASTIÁN JESÚS TOBAR RIVAS

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL INFORMÁTICO

PROFESOR GUÍA: CLAUDIO LOBOS

ABRIL 2016

**UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA**
**DEPARTAMENTO DE INFORMÁTICA**
**SANTIAGO – CHILE**




# "MEJORAMIENTO DE LA MANTENIBILIDAD Y EXTENSIBILIDAD DE UNA HERRAMIENTA DE GENERACIÓN DE MALLAS VOLUMÉTRICAS"

**SEBASTIÁN JESÚS TOBAR RIVAS**

**MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE INGENIERO CIVIL INFORMÁTICO**

**PROFESOR GUÍA: CLAUDIO LOBOS**
**PROFESOR CORREFERENTE: JOCELYN SIMMONDS**

**ABRIL 2016**

# Agradecimientos

Antes que nada, quisiera extender un especial agradecimiento a CoderGears por facilitarme el software CppDepend, que ha sido de una gran ayuda para el desarrollo de esta memoria. El apoyo de este software ciertamente me ahorró muchos meses de trabajo, y por ello les estoy profundamente agradecido.

En este espacio quisiera agradecer a todas las personas que me han ayudado a terminar este trabajo:

A los profesores Claudio Lobos y Jocelyn Simmonds, que aparte de darme este tema me ayudaron mucho con la redacción y edición de la memoria, corrigiendo errores y dándome sugerencias sobre el desarrollo de este trabajo;

A mi familia, papá, mamá y abuelos, que me han dado apoyo todo este tiempo, y me han proporcionado el espacio para escribir este trabajo y realizar todas las pruebas que necesitase;

A mis amigos con los que hemos jugado y liberado tensiones; y a los Primos del Lab por estar siempre ahí para cualquier cosa, prestarme equipamiento y dar jugo (¡y por los guatones!);

Y finalmente, a Celeste: si no fuera por tí, no creo que hubiese podido mantener la motivación para terminar esto, me hubiese fundido antes. ¡Te amo!

Además, quiero agradecer a todas las personas que me ayudaron a terminar la carrera: profes y amigos que sin ellos no hubiese podido pasar mis ramos, y que son demasiados para mencionarlos todos. ¡Muchas gracias!

# Resumen

Por lo general, las aplicaciones que implementan algoritmos de alta complejidad son diseñadas sin la suficiente consideración a características relacionadas con la calidad del software, para obtener el mejor desempeño posible en términos de tiempo de ejecución y uso de memoria. En este trabajo, se realiza el rediseño de una aplicación de generación de mallas volumétricas, buscando mejorar su mantenibilidad y extensibilidad. Para esto, se hizo un análisis global de la aplicación, determinando que un componente en particular realizaba una gran cantidad de labores críticas del proceso. Se integró el patrón de diseño "visitante" de Gamma et al. para repartir y organizar las labores de este componente en distintos elementos visitantes. Las pruebas de desempeño realizadas demuestran que las diferencias en tiempo de ejecución son aceptables y las diferencias de uso de memoria son despreciables. La aplicación de métricas de calidad de software a las implementaciones realizadas demuestran una mejora en los atributos de calidad de extensibilidad y flexibilidad, a costa de la capacidad de comprensión del diseño.

# Abstract

Applications that implement high complexity algorithms are generally designed to obtain the best possible performance in terms of execution time and memory usage, without enough consideration to other requirements regarding software quality. In this work, an application for generating volumetric meshes is redesigned to improve its maintainability and extensibility. This is achieved by refactoring a highly accessed, oversized component of the application, integrating elements of the "visitor" design pattern proposed by Gamma et al. in order to organize and distribute the functions of this component in smaller visitor elements. Performance tests show that time execution differences are acceptable, while memory usage differences are insignificant. Quality metrics applied to the new implementations show that the extensibility and flexibility quality requirements of the design are improved, at expenses of its understandability.

# Índice de Contenidos

# Índice de cuadros

# Índice de figuras

# Glosario

- Malla volumétrica: Representación de un volumen, compuesta de distintos elementos geométricos tridimensionales.
- Modelo de superficie: Representación de una superficie tridimensional que encierra un volumen, compuesta de triángulos planos.
- Octante: Conjunto de ocho elementos, utilizados para subdividir un dominio geométrico. Pueden representar distintos elementos geométricos, o dividirse en otros octantes.
- Nivel de refinamiento: Indicador de la cantidad de veces que se han subdividido los octantes que representan una determinada región del espacio.
- Patrón de transición: Conjunto de elementos mediante los cuales se mantiene la integridad de una malla volumétrica, entre regiones con niveles de refinamiento diferentes.
- Patrón de diseño: Sugerencia de solución para un determinado problema de diseño, implementada de forma reutilizable mediante orientación a objetos.
- Firma (*signature*) de un método: invocación de un método, que incluye su nombre y los parámetros que utiliza.

# Introducción

Dentro del ámbito de la investigación informática existen varios problemas que requieren algoritmos muy complejos para obtener soluciones. Los programas que se diseñan e implementan para estos efectos suelen ser escritos con el expreso objetivo de ejecutar el desarrollo del algoritmo de la forma más eficiente posible.

La generación de mallas geométricas es uno de estos problemas: consiste en la creación y organización de un conjunto de figuras geométricas, para representar un área o un volumen. Estos conjuntos de figuras, denominados mallas, pueden representar el comportamiento de objetos en simulaciones computacionales, y son muy utilizados en ingeniería y medicina.

Existen varios algoritmos que, en base a un modelo de superficie, generan una malla volumétrica; estos algoritmos se diferencian en las figuras base que utilizan para realizar la malla. En particular, se tienen algoritmos que combinan hexaedros regulares con otros elementos tridimensionales, para generar mallas con niveles variables de refinamiento, de forma de poder disminuir el tiempo de cómputo de una simulación.

El desarrollo de estas herramientas está enfocado principalmente en optimizar la velocidad y/o el uso de memoria de los procesos, generalmente dejando de lado atributos como mantenibilidad y extensibilidad. Estos últimos atributos, relacionados con la calidad del software, han cobrado importancia en el último tiempo, como un modo de hacer más sencilla la implementación de nuevos algoritmos, evitar la duplicación innecesaria de código y facilitar el mantenimiento del mismo por desarrolladores nuevos.

Por otra parte, aunque es natural suponer que dar más importancia a atributos de calidad puede tener un impacto negativo en el desempeño de una aplicación, esto no necesariamente

es así, existiendo ejemplos de aplicaciones [10] que han pasado por un proceso de rediseño y no han disminuido su velocidad, o han aumentado su uso de memoria. Estos rediseños pueden seguir las pautas propuestas por los patrones de diseño del Gamma et al. [15], Martin Fowler [14], y otros, para mejorar la calidad del software.

En este trabajo, se realiza un análisis y rediseño de una aplicación que implementa un algoritmo de alta complejidad, específicamente, la herramienta de generación de mallas volumétricas desarrollada por el profesor Claudio Lobos [24]. Este rediseño tiene como enfoque mejorar la mantenibilidad y extensibilidad de la aplicación, de forma de que futuras modificaciones a la misma se puedan realizar de forma más sencilla, y se mejore en general la calidad de la aplicación.

La estructura de la presente memoria es la siguiente:

- El capítulo 1 presenta la aplicación a rediseñar, las razones para hacerlo y los objetivos que se quieren cumplir con ello.
- El capítulo 2 muestra la investigación realizada para definir "mantenibilidad" y "extensibilidad", el efecto del uso de patrones de diseño en mejorar la calidad del código y los cambios en el rendimiento de las aplicaciones rediseñadas.
- El capítulo 3 describe las herramientas utilizadas para comprender el diseño de la aplicación, crear un diseño nuevo y comprobar las diferencias entre ellos, en términos de rendimiento y métricas de código fuente.
- El capítulo 4 presenta el diseño inicial, el patrón de diseño que se integrará el diseño nuevo, y cómo se comparan ambos diseños.
- El capítulo 5 detalla la forma en que se llevó a cabo la implementación del diseño, y cómo cambia esto la forma de ejecutar los métodos de la aplicación.
- El capítulo 6 presenta las diferencias en rendimiento y métricas del rediseño de la aplicación, y analiza estos resultados.
- Finalmente, en la conclusión, se discutirá el cumplimiento de los objetivos de la presente memoria.

# Capítulo 1

# Definición del Problema

## 1.1. Descripción

Se define una malla (o *mesh*, en inglés) como "una discretización de un dominio geométrico" [17], representaciones de objetos tridimensionales compuestas de elementos que pueden ser bidimensionales (triángulos o cuadriláteros) o tridimensionales (hexaedros u tetraedros); los primeros son utilizados para representar superficies, mientras que los otros se utilizan para representar volúmenes. Este tipo de mallas son ampliamente utilizadas para realizar simulaciones físicas, mediante la aplicación del método de elementos finitos.

Para generar mallas volumétricas existen distintos algoritmos. En particular, la técnica de *octree* [25] divide el dominio en ocho elementos, denominados "octantes", los cuales pueden representar algún elemento geométrico (por lo general un hexaedro regular, como un cubo), o bien subdividirse en nuevos octantes; la cantidad de veces que deban dividirse los octantes para representar adecuadamente el modelo define el "nivel de refinamiento" de la malla. Estos octantes se organizan en una estructura de árbol, de ahí el nombre de "octree". Luego, se puede entender el octante como un conjunto de ocho puntos en el espacio, los cuales representan una o más figuras geométricas, mientras que el octree es el conjunto de octantes que conforman la malla volumétrica.

Los puntos de un octante pueden tomar distintas formas dependiendo del nivel de refinamiento requerido de la malla volumétrica, la cual es generada en base a un modelo de superficie, y cuyo nivel de refinamiento no necesariamente es constante en toda la malla. Puesto que entre áreas con distintos niveles de refinamiento pueden aparecer problemas de continuidad geométrica, se han desarrollado patrones de elementos mixtos [24] que transforman el hexaedro en un conjunto de elementos geométricos, como tetraedros, pirámides, prismas, etc. El uso de estos elementos permite transiciones consistentes entre zonas de refinamiento diferente, y también puede aprovecharse para representar de mejor manera una superficie.

Debido a que estos algoritmos de generación de mallas son de alta complejidad y utilizan grandes cantidades de memoria (dependiendo del nivel de refinamiento y de la complejidad del modelo de superficie, la generación de una una malla geométrica puede requerir más de 16 GB de memoria RAM), la implementación de los mismos se realiza de forma que ocupe la menor cantidad de memoria y tiempo de procesador posible. Este enfoque tiende a no considerar atributos relacionados con la calidad de software en el diseño de la aplicación, tales como efectividad, flexibilidad, capacidad de comprensión, entre otros [3].

Sin embargo, recientemente se han realizado rediseños de herramientas de generación de mallas volumétricas, basadas en otras técnicas distintas a *octree*, con resultados prometedores [10], los cuales convendría comprobar. Para este efecto, se cuenta con la aplicación de generación de mallas volumétricas del profesor Claudio Lobos, que implementa los patrones de elementos mixtos descritos en [24]. Esta aplicación, escrita en C++, consta de 36 clases y 432 métodos, definidos en 4789 líneas de código; estas definiciones se reparten en 85 archivos de código fuente.

La aplicación será sometida a un proceso de rediseño, utilizando como base patrones de diseño que se ajusten a las funciones que realiza el programa: ya que el uso de patrones de diseño tiende a resultar en una mejor calidad de software, rediseñar la aplicación del profesor Lobos en base a algunos de ellos podría mejorar atributos de mantenibilidad y extensibilidad, los cuales se pueden considerar como importantes, ya que este software es revisado y extendido ([16], [2], [11]) en distintas memorias de pregrado.

## 1.2. Objetivos

### 1.2.1. Objetivo principal

Mejorar la mantenibilidad y extensibilidad de una herramienta de generación de mallas volumétricas mediante el rediseño de la misma en base a patrones de diseño.

### 1.2.2. Objetivos secundarios

- Proponer e implementar un diseño nuevo, basado en patrones de diseño, para la aplicación de generación de mallas volumétricas.
- Medir el efecto en el rendimiento que pueda tener el rediseñar esta herramienta.
- Evaluar la mejora del software con respecto a la implementación original, en cuanto a la facilidad de mantener y extender el mismo.

# Capítulo 2

# Estado del Arte

La idea de realizar este trabajo parte de las investigaciones realizadas por las profesoras Nancy Hitschfield y María Cecilia Bastarrica [4], sobre el efecto de los diseños orientados a objetos en el desempeño de herramientas de generación de mallas volumétricas. Puesto que en cierta medida se desea repetir estas pruebas con otros tipos de herramientas, resulta natural revisar la metodología de estos trabajos y de otros relacionados.

Tomando como base estos trabajos, el presente capítulo está estructurado de la siguiente forma: en la sección 2.1 se examinan trabajos que definen calidad de software, mediante métricas aplicables en distintas etapas del proceso de diseño e implementación. En la sección 2.2 se revisan trabajos que validan la efectividad de la aplicación de patrones de diseño en procesos de refactorización. Luego, en la sección 2.3 se describen distintas estrategias de refactorización; en la sección 2.4 se discuten los trabajos mencionados anteriormente, entre otros; para finalmente describir una metodología de refactorización, en la sección 2.5.

## 2.1. Evaluación de la calidad del software

Dentro de los trabajos investigados, la fuente más citada fue el trabajo de Chidamber y Kemerer [7] sobre métricas enfocadas en orientación a objetos. El trabajo en [7] es del año 1994,

y complementa el trabajo de Li y Henry [22], enfocado a investigar por primera vez métricas de mantenibilidad de un sistema orientado a objetos. En estos dos trabajos se definen métricas que son posteriormente utilizadas en muchas otras investigaciones.

Basado en estas métricas de Chidamber y Kemerer, Bansiya y Davis [3] propusieron el año 2001 un modelo de calidad que puede aplicarse en fases tempranas del diseño, y que tiene la particularidad de definir ponderaciones de las métricas para realizar comparaciones de atributos de calidad.

Así, estas métricas pueden clasificarse en dos tipos: aquellas diseñadas para aplicarse en la fase de diseño (como las de Bansiya y Davis) y las se aplican mejor habiendo ya una implementación parcial o total (como las de Chidamber y Kemerer).

Fuera de esta clasificación, existen también métricas para mejorar la legibilidad del código fuente, que se basan en el estilo del código, más que en el contenido de este.

### 2.1.1. Primeras métricas orientadas a objetos

El trabajo de Chidamber y Kemerer [7] describe el proceso de desarrollo, evaluación y prueba empírica de seis métricas de diseño orientado a objetos. Este es el primer trabajo que sugiere métricas con una sólida base teórica, independiente de la implementación, y cuya obtención no presenta una gran dificultad.

Las métricas generadas son:

- Suma de métodos ponderados de la clase (*Weighted Methods in Class*, WMC): Se pondera cada método por su complejidad, antes de sumarse.
- Profundidad del árbol de herencia (*Depth of Inheritance Tree*, DIT): Distancia entre una clase hija y la clase raíz.
- Cantidad de hijos (*Number of Children*, NOC): Conteo de sub-clases directas de una clase.

- Acoplamiento entre objetos (*Coupling Between Objects*, CBO): Cantidad de clases sobre la cual una clase realiza operaciones.
- Respuestas para una clase (*Response for a Class*, RFC): Cantidad de métodos a llamar en respuesta a un mensaje recibido por la clase.
- Falta de Cohesión en Métodos (*Lack of Cohesion in Methods*, LCOM): Relación entre variables de la clase y métodos de la clase.

Las seis métricas se sugieren como un método para evaluar cuán complejo sería mantener y entender una clase dada, entendiendo que una clase con una evaluación más alta de estas métricas será más compleja que una clase con evaluación más baja, y que altos valores de las métricas pueden ser útiles para sugerir rediseños de una clase. El estudio aplica las métricas a programas en C++ y Smalltalk; la definición de complejidad de los métodos queda a criterio del evaluador. Los autores, en un trabajo posterior [8], validan estas métricas, aplicándolas con éxito en sistemas comerciales orientados a objetos.

En el informe técnico de Li y Henry [22] se evalúan favorablemente las métricas propuestas por Chidamber y Kemerer, en cuanto a su utilidad para evaluar la complejidad de una clase. Además, el trabajo sugiere otros métodos para calcular acoplamiento entre objetos (por herencia y por paso de mensajes), y presenta una métrica adicional: la cantidad de métodos de una clase, útil para mostrar la complejidad de la misma.

En una evaluación del año 2001, realizada por Dubey y Rana [12], se reitera el apoyo al uso de las métricas de Chidamber y Kemerer para medir mantenibilidad, citando una gran cantidad de trabajos, métricas de distintos autores, y resumiendo la relación de cada métrica con aspectos de la mantenibilidad, testeabilidad, productividad y esfuerzo de diseño de un sistema.

### 2.1.2. Métricas de diseño: Bansiya y Davis

Este es un modelo jerárquico para la evaluación de la calidad de los diseños orientados a objetos.

El modelo define cuatro niveles de abstracción:

- Atributos de calidad de diseño
- Propiedades del diseño orientado a objetos
- Métricas del diseño orientado a objetos
- Componentes del diseño orientado a objetos

El modelo define relaciones entre estos cuatro niveles, de forma de poder realizar una evaluación de los atributos de calidad (que se definen desde las propiedades del diseño) en base a las métricas que se toman desde las características de los componentes utilizados en el diseño.

Se definen 11 métricas con directa relación a 11 propiedades del diseño. Estas métricas están basadas en las de Chidamber y Kemerer (sección 2.1.1), a las cuales se le agregan métricas para otras características del diseño, como la cantidad de clases del sistema, encapsulamiento de los datos, relaciones de composición, etc.

Las métricas, una vez normalizadas, se evalúan con ponderaciones específicas, caracterizando seis atributos de calidad: reusabilidad, flexibilidad, capacidad de comprensión, funcionalidad, extensibilidad y efectividad. Se utiliza una herramienta llamada QMOOD++[1] para realizar la medición de las métricas en forma automática, extrayendo el diseño desde código fuente, para luego realizar la ponderación de los atributos de calidad.

En la sección 3.3 se describen en profundidad los atributos y métricas planteados por Bansiya y Davis que son más relevantes a la presente memoria.

### 2.1.3. Métricas de legibilidad de código

Trabajos recientes han procurado establecer la legibilidad del código fuente como una característica importante de la mantenibilidad de un sistema, anteriormente ignorada debido a que

[1] Using QMOOD++ for object-oriented metrics. `http://www.drdobbs.com/web-development/automated-metrics-and-object-oriented-de/184410338` (Consultado el 22 de Junio de 2015)

se enfoca en la forma del código fuente, no en su contenido.

Weiting Lee, en su tesis de pregrado [21], realizó un estudio de diferentes métricas que miden la inteligibilidad del código fuente, es decir, su capacidad de ser comprendido rápidamente, sin conocimiento previo de lo que trata. Estas métricas se basan en el concepto de complejidad espacial planteado por Chhabra y Gupta [6], basadas en la distancia en líneas de código entre la declaración de una función y sus usos.

Por otra parte, el trabajo de Buse y Weimer [5] sugiere el uso de métricas relacionadas con la sintaxis del código (longitud de nombres de variables, longitud de líneas, indentación, palabras clave, identificadores, etc.) para predecir la legibilidad de secciones independientes de código fuente. Utilizando algoritmos de aprendizaje, crearon un modelo en base a las evaluaciones de 120 voluntarios, con distintos niveles de experiencia, sobre 100 secciones de código fuente. El modelo resultante sirvió además para encontrar una correlación con métricas de calidad de software convencionales (estabilidad, cambios en el código, defectos encontrados), sacadas de distintos proyectos de código abierto.

## 2.2. Uso de patrones de diseño en refactorizaciones

### 2.2.1. Efectividad de los patrones de diseño

Ya que es asumido que el uso de patrones de diseño viene asociado a una mejora en atributos de calidad de software, una parte de la investigación se centró en comprobar la efectividad de los patrones de diseño propuestos por Gamma *et al.* [15] en mejorar estos atributos.

Sfetsos *et al.* [28] compararon la efectividad de los patrones de diseño, en relación a su uso en librerías de software y aplicaciones independientes. Se hace la distinción, para comprobar el supuesto de que una librería de software tendrá un mejor diseño que una aplicación aislada, dado que una librería debe proveer una interfaz inmutable, sin importar cómo evolucione la misma.

Se obtuvieron mediante QMOOD [3] datos de la calidad de 26 proyectos de código abierto,

para luego analizarlos estadísticamente; se comprobó que "la funcionalidad, extensibilidad y efectividad de las librerías es significativamente mayor" que las aplicaciones aisladas [28], pero no pudo generalizarse la noción de que una librería use más patrones que una aplicación. Sin embargo, se pudo determinar que hay patrones más efectivos para mejorar la calidad de librerías, y otros patrones más efectivos para mejorar la calidad de aplicaciones; en particular, patrones como Adapter son más efectivos en aplicaciones, por lo que sería prudente examinar el efecto que tendría en la herramienta a refactorizar.

Por otra parte, en una investigación realizada por Ng *et al.* [26], consistente en un experimento controlado para comprobar si el esfuerzo de realizar una refactorización compensa el ahorro de trabajo de quienes modificarán posteriormente el software, en contraste con no modificar el software y procurar que las modificaciones sean hechas por programadores expertos. Para el experimento, se pidió a estudiantes de pregrado y de posgrado realizar mantenimiento perfectivo en un software con y sin patrones de diseño; de forma consistente, los programadores trabajando en la versión refactorizada pudieron realizar sus cambios más rápido, sin importar la experiencia de los encargados. Esto sugiere que una refactorización de software podría acelerar la metodología de investigación que implique la evolución del software del profesor.

### 2.2.2. Aplicación de patrones al modelado

Una distinta interpretación de la refactorización mediante patrones de diseño es la propuesta por Shahir *et al.* [29] que aplica los patrones de diseño al modelado basado en mundo real, es decir, modelos que se hacen en base a objetos y situaciones tangibles. Esta aplicación resulta especialmente efectiva para descartar elementos irrelevantes o clases con demasiados atributos, permitiendo que otras partes interesadas del proyecto puedan entender el modelo sin mucha dificultad; adicionalmente, los modelos refactorizados se prestan mejor a la aplicación de patrones en la fase de diseño o implementación. Así, se refleja en el paper la conveniencia de usar patrones de diseño, incluso en casos poco convencionales.

### 2.2.3. Validación empírica de la efectividad de los patrones

Contrastando con los resultados positivos mostrados por los papers anteriores, Khomh y Guéhéneuc [19] realizaron un estudio basado en las evaluaciones de distintos ingenieros sobre el impacto de los patrones de diseño, para comprobar de forma empírica la hipótesis de que "los patrones de diseño afectan positivamente la calidad del software", en relación a varios atributos. Entre los 23 patrones analizados, se destacaron 3 de ellos por su impacto universalmente positivo, neutro y negativo:

- **Composite** tuvo un impacto positivo en casi todos los atributos, salvo en escalabilidad: hace que un grupo de objetos sea tratado igual que un objeto particular, al coste de que hace más instanciaciones de objetos.
- **Abstract Factory** afectó a la mitad de los atributos positivamente y a la otra mitad negativamente: el patrón permite cambiar la forma en que se instancian objetos en tiempo de ejecución; pero, según las respuestas de los ingenieros encuestados, es muy difícil de implementar efectivamente.
- **Flyweight** tuvo un impacto negativo en todos los atributos, salvo escalabilidad: este patrón resuelve un problema muy particular (crear objeto con características internas y externas, para ser referenciado muchas veces con muy alta escalabilidad) pero es muy complicado y no puede usarse para nada más.

En resumen, ciertos patrones son demasiado específicos a un problema particular, o son complejos de implementar, o tienen inconvenientes; se concluye que los patrones de diseño deben usarse con cuidado, pues, al contrario de la creencia general, no siempre mejoran el diseño, y pueden agregar complejidad. Puesto que se pretende mejorar la extensibilidad y mantenibilidad de un código, es necesario tener esto en cuenta para no complejizar demasiado el diseño.

## 2.3. Estrategias de refactorización

### 2.3.1. Corrección de "olores" en el código

Una manera de detectar qué partes de un código necesitan ser refactorizadas es mediante la detección de "olores del código" o *code smells*, definidos por primera vez por Martin Fowler en 1999 [14]. Un mal olor en el código se define como una mala práctica, una estructura específica que "sugiere (y a veces pide a gritos) la posibilidad de una refactorización". Para corregir estos olores, Fowler sugiere varias estrategias de refactorización; estas son muchas, y no necesariamente ingresan patrones de diseño en el código, sólo se preocupan de corregir estas malas prácticas.

Una investigación realizada por Fontana y Spinelli [13] evaluó el impacto de la aplicación de estas refactorizaciones en la calidad del código: se obtuvieron métricas antes y después de la aplicación de refactorizaciones automáticas; si bien la refactorización funciona para eliminar los defectos específicos buscados, el trabajo no asegura que las herramientas no agreguen otros defectos; sin embargo, se dan varias herramientas que se pueden utilizar para evaluar el código.

### 2.3.2. Validación de la efectividad de las refactorizaciones

Otro trabajo que busca evaluar la efectividad de cada tipo de refactorización para la mejora de las métricas de código (no necesariamente mediante patrones de diseño) fue realizado por Shatnawi y Li [30] en el cual se utiliza un modelo abreviado de Bansiya y Davis (de forma semejante a lo establecido en la sección 2.1.2) para determinar la calidad del software antes y después de aplicar las refactorizaciones de Fowler. De forma adicional, se establecen heurísticas para la elección de refactorizaciones, dependiendo de cuáles sean los atributos de calidad que se desean mejorar.

Los factores de calidad que se revisan en el paper son reusabilidad, flexibilidad, extensibilidad y efectividad. No se incluye la capacidad de comprensión, porque "es un factor subjetivo y usar un modelo de medidas estáticas para evaluar comprensión es inadecuado" [30]. Con

todo, se decidió mantener aquel atributo de calidad en la evaluación que se hará en este trabajo: es el factor que Bansiya y Davis nombran como reemplazo del atributo de calidad de "mantenibilidad" de la ISO 9126 [3], y se considera importante para identificar partes del código que necesiten ser comprobadas para hacerlas más comprensibles. Por otra parte, este trabajo no considera el atributo de reusabilidad, debido a que es poco probable que el código a revisar sea utilizado en otros proyectos de software.

El paper finalmente llega a una relación entre varios métodos de refactorización de Fowler [14] y su impacto en las métricas establecidas por Bansiya y Davis [3] relacionadas con los atributos de calidad escogidos, las cuales fueron validadas en distintos proyectos de código abierto. Se observa que las métricas de mensajería (no citadas anteriormente) y acoplamiento son las más afectadas cuando se realizan refactorizaciones, mientras que las medidas de cohesión y herencia son las menos afectadas.

### 2.3.3. Inclusión de patrones de diseño

Aparte del concepto de defectos y refactorizaciones establecido por Fowler, se encontró [18] una aproximación a la generación automática de estrategias de refactorización, mediante programación genética. El trabajo de Jensen y Cheng usa una representación basada en un árbol de refactorizaciones atómicas (no relacionadas con el trabajo de Fowler), las cuales se aplican de forma secuencial; el algoritmo genético intercambia estas refactorizaciones para buscar una mejor solución. La representación está desarrollada de tal forma que al ejecutarse consecutivamente los cambios, se introducen patrones en el diseño de software. Nuevamente aparece el trabajo de Bansiya y Davis [3], al utilizarse QMOOD como función de evaluación del algoritmo genético.

Las pruebas realizadas a un software ya implementado sugieren estrategias de refactorización en 4 pasos en promedio, con un promedio de 12 patrones de diseño nuevos a ser aplicados. Por tanto, de existir este trabajo como un software utilizable, sería un gran aporte al desarrollo de la memoria; sin embargo, de no haber una implementación, no resulta conveniente aplicar las técnicas estudiadas, debido a que los operadores genéticos requieren validación semántica potencialmente engorrosa.

Adicionalmente se revisó un trabajo, de Palma *et al.* [27], donde se hace un sistema experto para sugerir patrones de diseño en un contexto específico. El paper revisado muestra cómo se obtiene la información para el sistema experto, pero no es capaz, aún, de refactorizar un diseño ya creado.

## 2.4. Efecto de la refactorización en el rendimiento de las aplicaciones

Dado que la introducción de elementos de orientación a objetos viene asociado a un uso de memoria mayor, existe la creencia generalizada de que un proyecto de software de alto desempeño, como una herramienta de generación de mallas volumétricas, debiera optimizarse para ser ejecutado con el menor uso de recursos posible - sin embargo esto por lo general viene asociado a un diseño muy poco mantenible [17].

Se han desarrollado recientemente aproximaciones a proyectos que tradicionalmente dejan de lado buenas prácticas de diseño de software, aplicando diseños enfocados en flexibilidad y mantenibilidad. Ampatzoglou y Chatzigeorgiou [1] examinaron dos juegos de código abierto, los cuales fueron refactorizados siguiendo patrones de diseño, para luego evaluar su rendimiento; se encontró que la refactorización aumentó la cohesión y disminuyó la complejidad del proyecto, a costa del aumento del tamaño del mismo.

Por otra parte, Bastarrica y Hitschfield [4] desarrollaron una arquitectura base para facilitar el desarrollo de una familia de herramientas de generación de mallas; el diseño propuesto permite añadir nuevos algoritmos de forma sencilla, aprovechándose del código que comparten las implementaciones de estos últimos. Pruebas de desempeño realizadas entre los algoritmos refactorizados y los originales encuentran que no se producen grandes cambios en el rendimiento, mejorando igualmente la reusabilidad y mantenibilidad del código.

Finalmente, Contreras *et al.* [10] realizó una refactorización parcial de un algoritmo conocido de generación de mallas volumétricas[2] y comprobó empíricamente que el rendimiento del

[2]TetGen, A Quality Tetrahedral Mesh Generator. `http://wias-berlin.de/software/tetgen/`

software refactorizado es el mismo que el software original; este resultado es el que lleva a realizar la refactorización del software del profesor Lobos. Cabe destacar que, aún siendo una refactorización parcial, ésta fue una tesis de master de Contreras [9] que fue desarrollada en el transcurso de nueve meses, factor a tener en cuenta al planear el rediseño de una aplicación de similares características.

Se debe notar que los trabajos previamente citados ([1][10]) evalúan la calidad del diseño mediante las métricas de Chidamber y Kemerer [7]; la ponderación de estas métricas en cada trabajo tiende a ser ligeramente distinta que el modelo de Bansiya y Davis [3].

## 2.5. Ejemplo de refactorización en un componente real

Respecto a la metodología necesaria para aplicar una refactorización, se revisó un trabajo de refactorización de un componente de software, realizado por Kolb *et al.* [20], que mejora el diseño e implementación de un componente de manejo de memoria en máquinas copiadoras. Se decidió revisar este paper debido a que trata de un componente que se presume requiere un rendimiento muy alto, y porque se aplica la refactorización a un software heredado (legacy) que presenta un diseño complejo y difícil de mantener.

El proceso parte con una fase de documentación, en donde se obtiene un modelo del diseño del componente, mediante un análisis del código fuente asistido por herramientas. Luego, se realiza un análisis de métricas, variabilidad de código y búsqueda de código repetido.

La refactorización en sí se divide en cambios a nivel de diseño, como divisiones de sub-componentes e introducción de componentes nuevos, y cambios a nivel de implementación: renombrado de funciones, división de código, cambio de tipos de datos, etc.

Del trabajo se puede derivar una metodología concreta para realizar la refactorización del trabajo a revisar (documentación, análisis, rediseño, reimplementación). Una refactorización hecha con cuidado permite corregir el diseño sin introducir bugs, sin embargo, el proceso es muy largo (sólo para documentar se requirieron 2 meses) y asume que los recursos no alcanzarán para realizar todas las mejoras.

# Capítulo 3

# Metodología

Para lograr los objetivos detallados en la sección 1.2, se creó un plan de trabajo que puede resumirse en tres tareas separadas:

- Analizar la aplicación para identificar las clases más utilizadas.
- Proponer e implementar un nuevo diseño de estas clases, en base a algún patrón de diseño afín.
- Evaluar los cambios en tiempo de ejecución, uso de memoria y métricas de calidad producto de este rediseño.

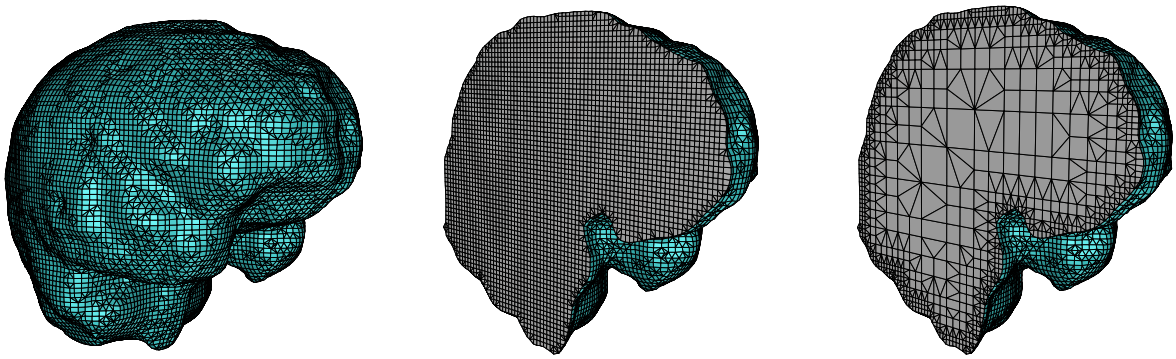
En esta sección, se explicará cómo fueron realizados los puntos anteriores, describiendo las herramientas utilizadas y definiendo los conceptos que sean necesarios.

## 3.1. Análisis de la aplicación

La aplicación desarrollada por el profesor Claudio Lobos, en adelante denominada `mesher`, recibe como entrada, en su forma más sencilla, un modelo de superficie (representado en la figura 3.1a) y el nivel de refinamiento deseado para la malla volumétrica. Opcionalmente se

puede especificar una región geométrica con un nivel de refinamiento más alto. El programa retorna un archivo con la malla geométrica terminada, la cual puede visualizarse mediante el programa `geomview`[1].

El nivel de refinamiento especificado en la entrada tiene dos formas. La forma `-a` (all) divide todos los octantes de la malla, tantas veces como especifique el nivel de refinamiento, como muestra la figura 3.1b; mientras que la forma `-s` (surface) sólo divide los octantes cerca de la superficie del modelo de entrada la cantidad de veces que especifique el nivel de refinamiento, dividiendo los octantes internos sólo las veces que sea estrictamente necesario, y aplicando patrones de transición entre los octantes de refinamiento diferente, como se ve en la figura 3.1c.



(a) (b) (c)

Figura 3.1: Ejemplo de modelo de superficie de entrada, y mallas resultantes con parámetros de refinamiento `-a` y `-s`. (Fuente: Lobos, González [23])

Existen tres aspectos que pueden revisarse mediante la ejecución de la aplicación a través de distintas herramientas: el tiempo de ejecución, el uso de memoria y las funciones más ejecutadas. Estos tres aspectos cobran importancia en distintas partes de este trabajo.

[1]Geomview (1.9.4) `http://www.geomview.org/`

### 3.1.1. Tiempo de ejecución

Registrar el tiempo que demora el programa en realizar su ejecución no requiere herramientas externas, pudiendo hacerse restando la hora de término con la hora de inicio de la ejecución. Esto lo hace el mismo programa de forma interna, y es el único dato que retorna por consola.

### 3.1.2. Uso de memoria

Para medir el uso de memoria de la aplicación se usa la herramienta `Massif`, disponible en la suite Valgrind[2]. `Massif` mide cuánta memoria es asignada en *heap*, identificando el uso en porcentajes de cada estructura de datos, y cómo cambia el uso de memoria a través de la ejecución del programa. También puede medir la memoria *stack* del programa, para obtener una medición más parecida a la reportada por el sistema operativo. Usar `Massif` ralentiza mucho la ejecución del programa analizado, por lo que no es recomendable realizar esta medición en conjunto con la de tiempo de ejecución.

### 3.1.3. Llamadas a funciones

Las llamadas a funciones se pueden analizar mediante la herramienta `Callgrind`, la cual también pertenece a la suite Valgrind[3]. Esta herramienta registra el historial de llamadas a funciones, y mediante el uso de herramientas gráficas como `KCachegrind`[4] se puede analizar qué porcentaje del tiempo de ejecución de la aplicación es usado en cada función, la jerarquía de llamadas entre las funciones de la aplicación, y el número de llamadas a cada función.

[2]Massif: a heap profiler (valgrind-3.10.1) `http://valgrind.org/docs/manual/ms-manual.html`

[3]Callgrind: cache and branch prediction profiler (valgrind-3.10.1) `http://valgrind.org/docs/manual/cl-manual.html`

[4]KCachegrind: Call Graph Viewer (0.7.4kde) `https://kcachegrind.github.io/html/Home.html`

## 3.2. Revisión del diseño de la aplicación

Según lo expuesto en la sección 2.4, el rediseño de una aplicación similar a `mesher` toma alrededor de nueve meses, por lo que se consideró necesario acotar el ámbito de este trabajo, limitando el rediseño a las clases más referenciadas por la aplicación. Es por esto que resulta muy útil disponer de información sobre la relación entre las clases del programa, así como de las llamadas a funciones, con las herramientas de la sección 3.1.3.

En primera instancia se utilizó `Doxygen`[5], una herramienta gratuita para generar documentación automáticamente en base al código fuente de un proyecto: genera listas de funciones, gráficos de relaciones entre métodos, diagramas de inclusiones, etc. Mediante esta documentación se pueden revisar rápidamente las partes de un programa y cómo están relacionadas.

La herramienta que finalmente fue utilizada con más frecuencia, y para muchas tareas relacionadas con el diseño de la aplicación, fue `CppDepend`[6]. Esta es una herramienta enfocada en proveer evaluaciones de calidad de código fuente, en base a análisis de código estático. Proporciona una gran cantidad de métricas a nivel de proyectos, clases y métodos, relacionadas con el tamaño del código, calidad del diseño, llamadas a funciones, relaciones entre objetos, complejidad, etc.

`CppDepend` permite complementar los datos de llamadas a funciones de `KCacheGrind`, mostrando en grafos o matrices cuáles son exactamente las dependencias entre las clases y entre los métodos de las clases. Además, proporciona alertas sobre potenciales problemas de diseño, como clases demasiado grandes, métodos demasiado complejos, entre otros.

## 3.3. Definición de calidad de software

En el capítulo 2, sección 2.1, se mencionan distintos trabajos que definen maneras de evaluar la mantenibilidad de un software orientado a objetos, o la calidad del software en general:

[5] Doxygen: generate documentation from source code (1.8.10) `http://www.stack.nl/~dimitri/doxygen/`

[6] CppDepend: static analysis and code quality tool (6.0.0.9010) `http://cppdepend.com/`

por una parte, el modelo de calidad de Bansiya y Davis [3], aunque está más centrado en obtener métricas antes de la implementación de un diseño, puede utilizarse para medir ciertos atributos de calidad relacionados con los objetivos de la presente memoria; por otra parte, la interpretación original de Chidamber y Kemerer de estas métricas [7], validado en el trabajo de Dubey y Rana [12], permite argumentar posibles mejoras de un rediseño que apunte a disminuir los valores de estas métricas.

La herramienta `CppDepend`, mencionada en la sección 3.2, es capaz de derivar una gran variedad de métricas, tanto a nivel de aplicación como a nivel de clase. Estas métricas pueden relacionarse con las métricas que proponen los autores antes mencionados, para realizar comparaciones entre distintas versiones de una misma aplicación.

A continuación se detalla de qué forma pueden obtenerse las métricas necesarias para evaluar el rediseño de la aplicación, y de cómo se relacionan para calcular otros atributos de calidad.

### 3.3.1. Atributos de Bansiya y Davis

Bansiya y Davis definen seis atributos de calidad en total; para limitar la cantidad de métricas necesarias para realizar la evaluación de la aplicación, se decidió tomar sólo un subconjunto de estos atributos. Los atributos de calidad que se evaluarán fueron escogidos por su relación con el objetivo de la presente memoria, que se refiere a la mantenibilidad y extensibilidad del software a evaluar.

Estos atributos son los siguientes:

- Comprensión (*understandability*): Propiedad del diseño de poder entenderse fácilmente.
- extensibilidad (*extendibility*): Propiedad del diseño que permite agregar nuevos requerimientos al mismo.
- Flexibilidad (*flexibility*): Características del diseño que permiten cambiar o adaptar el mismo.

Los atributos de calidad se ven afectados, a su vez, por las siguientes propiedades del diseño:

- Tamaño del diseño: Cantidad de clases de las que se compone una aplicación.
- Abstracción: Relación entre clases que proporciona una interfaz común a distintas maneras de realizar una acción.
- Encapsulamiento: Propiedad que protege la integridad del estado de una clase, evitando su modificación externa.
- Acoplamiento: Interdependencia entre las clases que componen un diseño.
- Cohesión: Medida de la relación entre los elementos de una clase y los métodos que implementa.
- Composición: Propiedad del diseño donde varias clases sencillas prestan en conjunto una funcionalidad compleja.
- Herencia: Relación entre clases que establece una jerarquía entre realizaciones de una interfaz.
- Polimorfismo: Capacidad de una interfaz de servir a objetos de diferentes tipos.
- Complejidad: Dificultad en entender la estructura de clases y cómo se relacionan entre sí.

Se excluyen de las propiedades de diseño consideradas en este trabajo las jerarquías (clases que son ancestros de otras) y la mensajería (servicios que provee una clase), debido a que están relacionadas con los atributos de reusabilidad, funcionalidad y efectividad, no considerados para la evaluación del diseño.

Cada propiedad mencionada se relaciona con una sola métrica, que, a su vez, se puede obtener directamente de los componentes del diseño orientado a objetos, como se muestra en el cuadro 3.1. Esto proporciona a las propiedades del diseño un valor numérico.

La comparación entre distintas versiones de un diseño se realiza definiendo un estado base, dividiendo los valores de las métricas de las versiones subsiguientes con las de este estado.

| **Propiedad de diseño** | **Métrica** | **Descripción** |
|---|---|---|
| Tamaño del diseño | Clases totales (DSC) | Cantidad total de clases del diseño. |
| Abstracción | Promedio de ancestros (ANA) | Cantidad promedio de clases desde las cuales una clase hereda información. |
| Encapsulamiento | Acceso de datos (DAM) | Relación entre la cantidad de atributos privados y el total de atributos. |
| Acoplamiento | Acoplamiento directo (DCC) | Cantidad de clases diferentes con las que una clase en particular se relaciona de forma directa. |
| Cohesión | Parámetros comunes de métodos (CAM) | Relación entre los tipos de parámetros de los métodos y el total de tipos de parámetro en la clase; sumado para todos los métodos. |
| Composición | Medida de agregación (MOA) | Cantidad de declaraciones de datos de tipos definidos por el usuario. |
| Herencia | Herencia funcional (MFA) | Relación entre los métodos heredados por una clase y el total de métodos de la misma. |
| Polimorfismo | Métodos polimórficos (NOP) | Cantidad de métodos que pueden ser polimórficos. |
| Complejidad | Cantidad de métodos (NOM) | Cantidad total de métodos en el diseño. |

Cuadro 3.1: Relación entre propiedades del diseño y métricas, con descripción. Fuente: Bansiya, Davis [3]

| Atributo de calidad | Relación entre propiedades |
|---|---|
| Comprensión | 0.33 * Cohesión + 0.33 * Encapsulamiento - 0.33 * Tamaño de diseño - 0.33 * Abstracción - 0.33* Acoplamiento - 0.33 * Polimorfismo - 0.33 * Complejidad |
| Extensibilidad | 0.5 * Abstracción + 0.5 * Herencia + 0.5 * Polimorfismo - 0.5 * Acoplamiento |
| Flexibilidad | 0.5 * Composición + 0.25 * Encapsulamiento + 0.5 * Polimorfismo - 0.25 * Acoplamiento |

Cuadro 3.2: Relación entre atributos de calidad y propiedades del diseño. Fuente: Bansiya, Davis [3]

El objetivo es definir el estado base con el número 1, caracterizando el aumento en cada métrica como un número mayor a 1, y la disminución como un número menor a 1.

Esta variación en las métricas, según las ecuaciones en el cuadro 3.2, permite caracterizar mejoras y deterioros de los atributos de calidad asociados al diseño de la aplicación, a lo largo de diferentes versiones del mismo. Al evaluarse las diferencias ponderadas según estas ecuaciones, se obtendrá un número mayor o menor a 1, representativo de la mejora o deterioro del atributo de calidad en su conjunto.

### 3.3.2. Relación con métricas de Chidamber y Kemerer

Se puede argumentar que las métricas de Chidamber y Kemerer (sección 2.1.1) pueden aplicarse también en el modelo de Bansiya y Davis, en algunos casos, con ciertas modificaciones:

- Cantidad de Métodos (NOM), que Bansiya y Davis usa como indicador de complejidad, fue sugerida por Li y Henry [22] para complementar la métrica de métodos ponderados por complejidad (WMC).
- El promedio de ancestros (ANA) puede derivarse directamente de la profundidad del árbol de herencia (DIT).

- La medida de cohesión de Bansiya y Davis (CAM) puede considerarse como inversamente proporcional a la falta de cohesión de métodos (LCOM) de Chidamber y Kemerer.
- Ambas métricas de acoplamiento (DCC y CBO) miden la relación de una clase con otras clases externas.

Chidamber y Kemerer no definieron métricas relacionadas con el encapsulamiento, la composición y el polimorfismo, y la medida de herencia cuenta sólo el número de hijos, en vez de revisar si estos hijos ocupan las funciones de los ancestros. Por otra parte, la métrica de respuestas para una clase (RFC) está relacionada con la métrica de mensajería del modelo de Bansiya y Davis, excluida en la sección 3.3.1.

### 3.3.3. Derivación de métricas en base a código fuente

CppDepend pone a disposición del usuario las métricas del código fuente que analiza mediante un lenguaje de consulta, similar a SQL, llamado CQLinq[7]. Este lenguaje de consulta define elementos del código con un vocabulario específico:

- Método: Es una función o procedimiento cualquiera.
- Tipo: Puede referirse a una clase, estructura o enumeración.
- Campo: Es una declaración de variable, constante u objeto, realizada en un contexto cualquiera.

Las métricas mencionadas en la sección 3.3, pueden derivarse de las métricas de CppDepend, según el cuadro 3.3.

[7]Code Query Linq. `http://www.cppdepend.com/cqlinq`

| Métrica | Descripción | Consulta de CppDepend |
|---|---|---|
| Tamaño diseño | Número de clases | Obtener conteo total de tipos |
| Abstracción | Promedio de ancestros | Promedio de nivel dentro del árbol de herencia de todos los tipos |
| Encapsulamiento | Atributos privados / atributos totales | Obtener campos con atributo “private”, restar a total de campos, dividir por total de campos |
| Acoplamiento | Uniones entre clases | En cada clase: acoplamiento aferente (quién depende de mi) más acoplamiento eferente (de quién dependo) |
| Cohesión | Cohesión entre métodos de una clase | Complemento de la métrica LCOM (falta de cohesión entre métodos) |
| Composición | Cantidad de datos de tipo definido por el usuario | Obtener todos los campos no primitivos y descartar contenedores (vector, list, set) conteniendo tipos primitivos |
| Herencia | Atributos heredados / atributos totales | Comparar atributos de clases padres y clases hijas |
| Polimorfismo | Número de métodos polimórficos | Obtener métodos con atributo de sobrecarga (incluye constructores de copia) |
| Complejidad | Número de métodos | Obtener conteo total de métodos |

Cuadro 3.3: Consultas de CppDepend que obtienen métricas de Bansiya y Davis.

### 3.3.4. Justificación de la aplicación de métricas

La aplicación de estas métricas dependerá de las circunstancias de la aplicación y de lo que se desea medir, que es la mantenibilidad y flexibilidad de la aplicación, y en menor grado la comprensión de la misma. En este contexto, cada métrica debiera ser aplicada con un objetivo específico, evitando obtener datos ciegamente.

- Número de clases: Estima el tamaño del diseño de la aplicación, lo cual puede identificar la facilidad para comprender la aplicación en su conjunto.

- Promedio de ancestros: Permite determinar cuán extensible es la aplicación, mediante la aplicación de herencia.
- Encapsulamiento: Estima la posibilidad de que la mantención de una clase se extienda a otras clases, por la modificación de métodos públicos.
- Acoplamiento aferente: Indica a cuántos elementos podría potencialmente afectar una modificación en una clase.
- Acoplamiento eferente: Define la dependencia de un elemento con su entorno, y está relacionada con el uso del elemento en aplicaciones distintas a las actuales.
- Falta de cohesión: Permite estimar si los métodos de una clase cumplen con una misma función, limitando el ámbito de aplicación del la clase y facilitando su uso.
- Herencia funcional: Determina si la herencia de la aplicación es bien aprovechada, y podría ser usada para extender la aplicación.
- Polimorfismo: Puede servir para señalar funciones que se pueden usar en distintos ámbitos de la aplicación.
- Número de métodos: Estima el tamaño de una clase, pudiendo con esto determinar la complejidad de la misma y la cantidad de responsabilidades que tiene.

Se omitió la métrica de la composición: como se pretende mover métodos de forma íntegra, sin realizar modificaciones, los tipos de datos nuevos a definir no estarán relacionados con la clase a refactorizar, por lo que la aplicación de esta métrica es objetable.

Por estas razones, en la medición de las métricas de las versiones de `mesher`, se evaluarán 9 métricas, relacionándolas individualmente con los atributos de calidad a medir en la presente memoria.

# Capítulo 4

# Propuesta de Rediseño

A primera vista, `mesher` aparece como una aplicación con una gran cantidad de piezas: como se mencionó en la sección 1.1, esta consta de 36 clases y más de 400 métodos, repartidos en alrededor de 80 archivos de código fuente. Sin embargo, muchas de estas clases definen formas distintas de realizar una función, y el flujo de la aplicación es controlado por unas pocas clases principales.

En el presente capítulo se describe el flujo de la aplicación, el rol de las clases principales, y las formas en que se podrían integrar ciertos patrones de diseño a estas clases, con el objetivo de simplificar las clases más grandes.

## 4.1. Diseño actual

### 4.1.1. Descripción

Las acciones que realiza la aplicación son coordinadas por dos clases principales: *Mesher* y *Octant*. Se distingue la aplicación de la clase del mismo nombre por el uso de las mayúsculas.

Estas dos clases interactúan entre sí para generar una malla geométrica de acuerdo con un modelo inicial, la cual es refinada mediante la acción de otras clases auxiliares.

El flujo de la aplicación puede describirse de la siguiente forma:

1. *Mesher* genera las coordenadas 3D del primer octante en base al modelo inicial, ubicados de forma equidistante en el espacio; estos puntos (*MeshPoint*) se guardan en una lista.

2. Luego, los puntos son agrupados en octantes: se genera una lista de *Octant*, y cada uno guarda ocho referencias a alguno de los puntos de *Mesher*.

3. Para cumplir con el nivel de refinamiento deseado, *Mesher* ordena a cada *Octant* que se divida en caso de ser necesario:

   - *Octant* genera puntos nuevos, que se guardan en la lista de puntos que tiene *Mesher*.
   - Se generan nuevos *Octant* en base a estos puntos nuevos.

4. *Mesher* ordena a cada *Octant* que aplique sobre sus puntos operaciones de transformación: este instanciará un patrón de transición (*SurfaceTemplate* o *TransitionTemplate*) acorde con su ubicación y la cantidad de puntos que tenga dentro del modelo, y entrega sus puntos al patrón para que sean transformados.

5. Entre cada operación se realiza una optimización de la relación entre los puntos de *Mesher* y las referencias que contiene cada *Octant*, eliminando los puntos que ya no son necesarios.

6. Una vez terminadas las operaciones de transformación y refinamiento, *Mesher* ordena la escritura de la malla geométrica en el disco duro.

Así, el diseño de la aplicación puede entenderse en cuatro partes:

- La clase *Mesher*, que coordina las operaciones a realizar y guarda elementos comunes.
- La clase *Octant*, que divide y aplica transformaciones sobre los puntos que definen los octantes.

- Las clases que definen los patrones de transformación, que son las que adaptan los puntos del *Octant* según los algoritmos que implementen.
- Otras clases que definen elementos comunes (*Point3D*, *RefinementRegion*, etc.) y realizan operaciones de lectura/escritura.

### 4.1.2. Análisis de la clase *Octant*

Cabe señalar que las operaciones de división y transformación que se realizan sobre un octante están todas contenidas en una sola clase: *Octant*. Como resultado, esta clase resulta ser de un gran tamaño: se efectuó una revisión a *Octant* mediante CppDepend, la cual muestra que la clase implementa 41 funciones en 781 líneas de código.

Una lectura a la clase muestra que está muy bien documentada, y entender lo que hace cada función no requiere mucho esfuerzo; sin embargo, el largo excesivo de cada función, y el hecho de que estén todas juntas, dificulta mucho la revisión del código y apunta a posibles problemas para encontrar defectos. Agregar nuevas funciones a *Octant* sólo aumentaría estas dificultades.

Por otra parte, una revisión de las dependencias sobre *Octant* muestra que esta clase es utilizada sólo por *Mesher*, con excepción de unas pocas funciones en otros componentes. Esto hace que cualquier modificación que se realice a la estructura de *Octant* quede limitada a realizar cambios sólo en estas dos clases.

Notar que en la jerarquía de llamadas a funciones de `mesher` generados por KCachegrind, en la figura 4.1, se aprecia que más del 70 % del tiempo de ejecución del programa es dedicado a refinar y transformar los octantes. Esto, combinado con la información anterior, permite concluir que los esfuerzos de rediseño de la aplicación debieran enfocarse a intentar ordenar las funciones de la clase *Octant*.

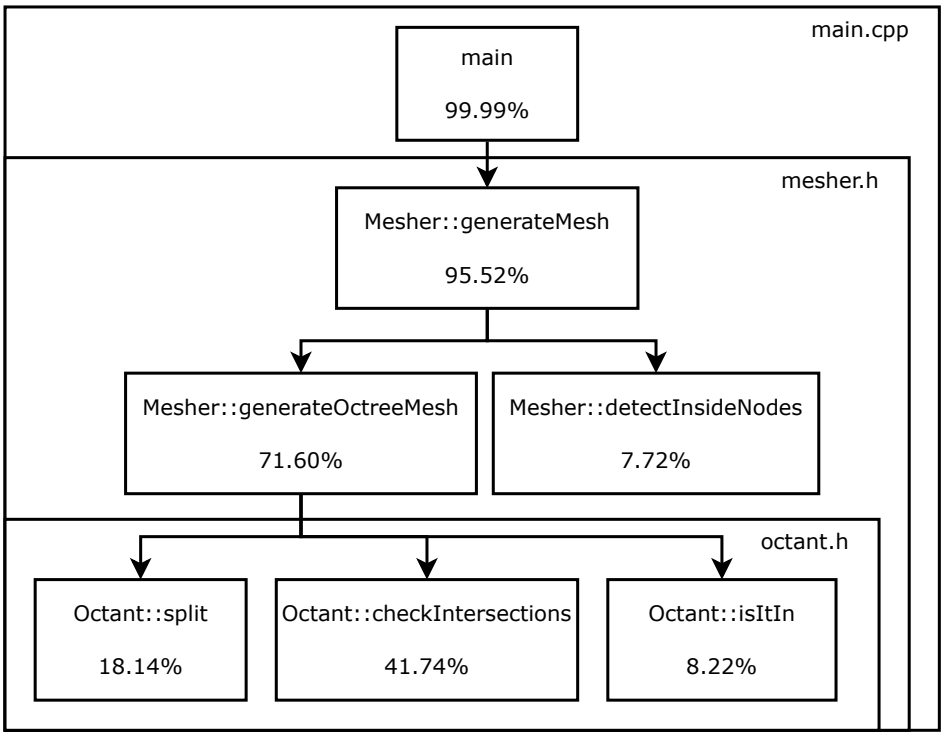



Figura 4.1: Jerarquía de llamadas a funciones de `mesher`, generado por KCachegrind. Figura no incluye todos los llamados a funciones.

## 4.2. Discusión de patrones de diseño

La clase *Octant* tiene el problema de que define demasiadas operaciones, haciéndose difícil de mantener y de comprender en su totalidad. Por esto, un patrón de diseño para esta clase deberá tener en cuenta que es necesario ejecutar muchas operaciones sobre los elementos que componen *Octant*.

Se revisaron los patrones de diseño de Gamma *et al.* [15], en particular los patrones de comportamiento. Asumiendo que la manera más eficiente de hacer de *Octant* una clase más entendible es mediante la categorización de sus funciones, se buscaron patrones que pudieran modelar la organización entre una serie de clases que, entre todas, pudieran replicar lo que hace *Octant*, y su relación con la clase *Mesher*.

En particular, el patrón denominado visitante "representa una operación a ser aplicada a los elementos de una estructura (...), permite definir una nueva operación sin cambiar las clases

de los elementos sobre los que opera". La idea del patrón visitante es diseñar clases sencillas que realicen una sola operación, sobre una o más clases de objetos.

Este patrón es apropiado de aplicar en circunstancias que se tengan muchos tipos de objetos a los que se le tenga que aplicar una misma operación, y cuando se tienen pocos objetos a los cuales haya que aplicar muchas operaciones diferentes. El diseño de estos objetos a los que se les aplicarán las operaciones del visitante tiene que ser estable: se facilita enormemente agregar operaciones nuevas sobre el objeto, pero cualquier cambio al diseño del objeto se tendrá que propagar a todos los visitantes.

Esto se hace mediante la creación de una clase abstracta *Visitor*, que es recibida por el objeto al que se le vayan a aplicar las operaciones en un método `accept` que este implementa; por otra parte, cada operación es realizada como una implementación concreta de la clase *Visitor*, que accede mediante la función `visit` al estado interno del objeto que va a modificar, ya sea por *getters/setters* o por modificación directa. Se determina en tiempo de ejecución qué código es el que se ejecuta sobre el objeto a modificar, mediante la propiedad de doble despacho: la decisión es tomada en base al tipo del visitante recibido por `accept` y al tipo de objeto que recibe `visit`. Las relaciones entre las clases objeto y visitante se ven en la figura 4.2.

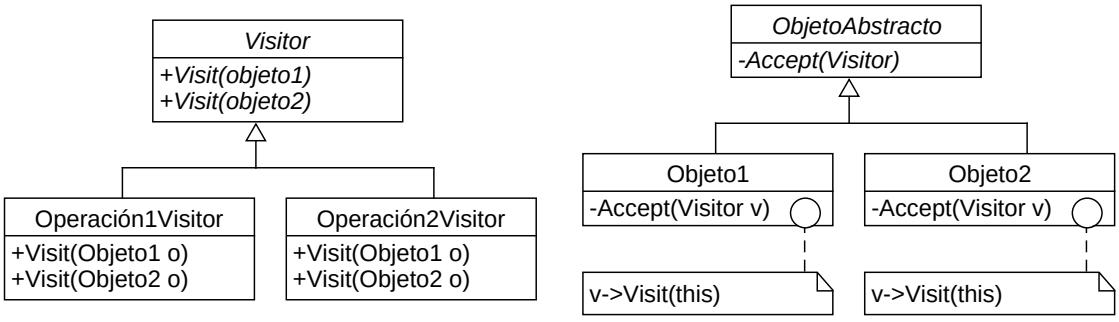



Figura 4.2: Diagrama de clase del patrón visitante. Fuente: Gamma et al. [15]

Por lo tanto, ya que el objetivo es aplicar varias operaciones sobre un mismo objeto (el octante), se decidió rediseñar la clase *Octant*, tomando como objeto a modificar el octante, e implementando en cada visitante las operaciones a reorganizar.

## 4.3. Integración del patrón Visitante

Para poder definir el contenido de cada visitante, se examinaron los métodos de *Octant* que fueran accedidos por otras clases, en particular por *Mesher*, de forma que esta clase instancie al visitante cuando sea necesario y se lo entregue al octante sobre el que realizar la operación.

Se pueden detectar estas funciones fácilmente mediante CppDepend. Al examinar la función `generateOctreeMesh` de la clase *Mesher*, se obtiene un grafo de llamadas de esta función a los métodos de *Octant*, como aparece en la figura 4.3. El grafo generado también muestra las dependencias dentro del mismo *Octant*, ayudando a decidir también cuáles son las funciones extra que deba tener el visitante para llevar a cabo su función.

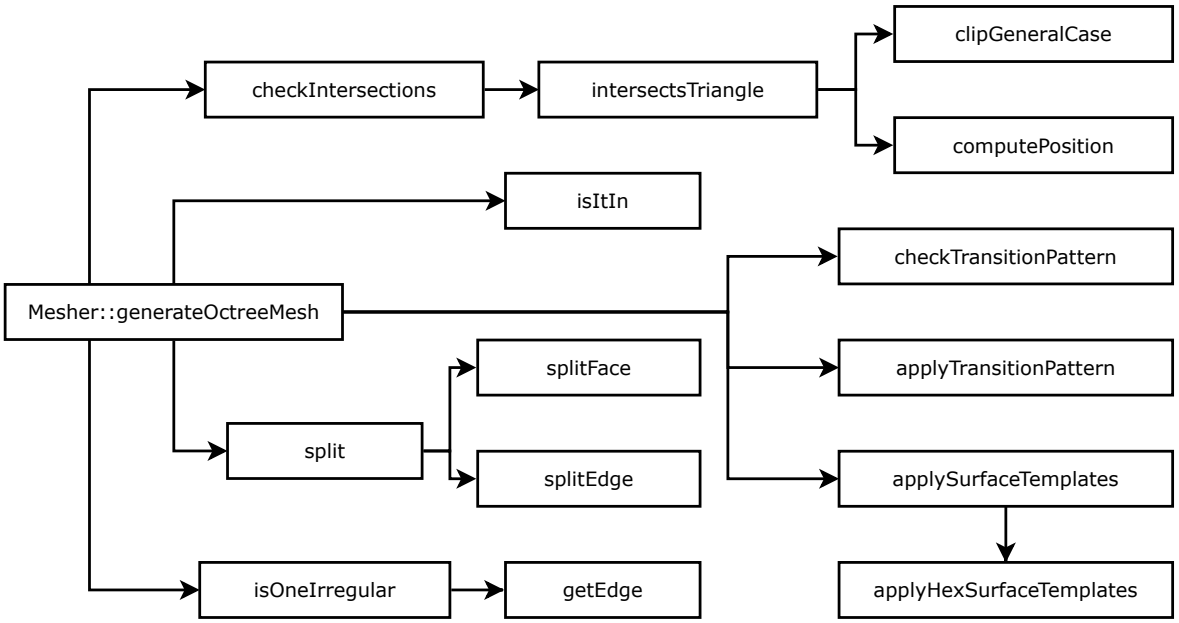



Figura 4.3: Dependencias de `generateOctreeMesh` en *Octant*, generado por CppDepend. Figura no incluye todas las dependencias.

Finalmente se detectaron ocho funciones en *Octant* que son llamadas por *Mesher* en distintas etapas de la ejecución del programa, que resultan ser las más complejas, y que a su vez son las únicas que llaman a otras funciones del mismo *Octant*. El resto de las funciones de la clase, que son invocadas por clases distintas a *Mesher*, fueron consideradas como parte fundamental de la clase, y no fueron movidas. En principio, los métodos de los visitantes

son idénticos a los del octante, sólo siendo necesarias modificaciones para realizar el acceso al octante desde clases externas. Esto hace que la creación de los visitantes sea sencilla.

Para ejecutar las operaciones que implementan los visitantes, se definen métodos de inicialización de variables y un método para ejecutar la operación, llamado `visit`. Este método es llamado por el objeto *Octant*, mediante un método llamado `accept`, el cual recibe un objeto *visitor* genérico.

La ventaja de este diseño es que las operaciones que se realizan sobre *Octant* pueden aplicarse con mínimas modificaciones a la clase, y el objeto `visitor` que se pasa al objeto *Octant* puede ser completamente arbitrario; la operación aplicada se define mediante herencia, sin necesidad de agregar funciones específicas a cada visitante.

## 4.4. Simplificación del patrón Visitante

Puesto que el patrón visitante decide qué operación aplicar mediante herencia, se espera que esto agregue una penalización de tiempo, que no se sabe *a priori* si será significativa.

Por esta razón, se decidió también implementar una forma especial de visitante, donde se realizará la extracción de los mismos métodos que el diseño de la sección 4.3, pero realizando las llamadas a funciones igualmente desde la clase *Octant*, para evitar las penalizaciones de ejecución provocadas por el uso de herencia.

Como en este caso no es necesario tener todos los visitantes por separado, se decidió agrupar los métodos extraídos de la clase *Octant* para categorizarlos. Es posible agrupar los métodos según su objetivo principal, del siguiente modo:

- Operaciones básicas
- Refinamiento del octante
- Revisión de condiciones (intersecciones, movimientos, etc.)
- Operaciones sobre los bordes del octante

- Aplicación de patrones de transformación

Para cada categoría se construyó un visitante, a excepción de las operaciones básicas, que se quedaron en la misma clase, igual que en la integración completa del patrón visitante. La clase padre *visitor* fue simplificada para incluir sólo las características necesarias para modificar *Octant*, eliminando efectivamente el uso de la herencia para decidir qué función ejecutar.

A diferencia del diseño con el patrón visitante completo, donde las funciones son llamadas mediante un método `visit` y los parámetros necesarios son entregados al visitante mediante métodos independientes, en este diseño los métodos serían llamados con una signatura muy parecida a la original. La clase *Mesher* también es modificada lo menos posible: es *Octant* quien hace las llamadas a los métodos en las clases visitantes. En general, se espera que los detalles de esta nueva implementación sean lo más parecidos posible a la implementación original, con esperanzas de bajar posibles tiempos de ejecución extra al mínimo.

## 4.5. Comparación entre diseños

La diferencia principal entre el diseño inicial y los otros dos diseños es, por supuesto, la inclusión de los visitantes, y el tamaño de la clase *Octant*.

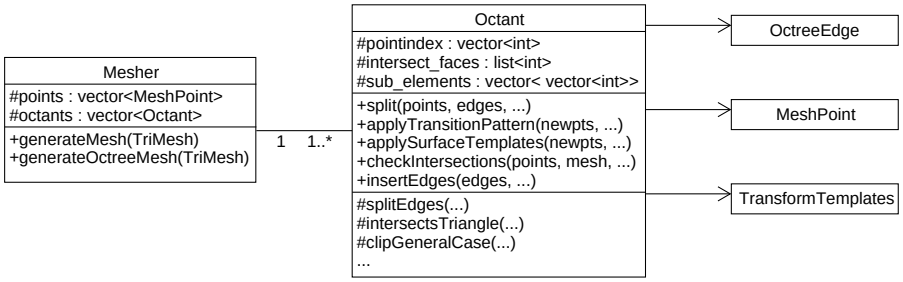
A primera vista, los nuevos diseños resultan más complejos que el diseño anterior, debido a la inclusión de más clases y la interacción entre estas, como se ve al comparar los diagramas en la figura 4.4. Estos diagramas están muy simplificados, para mostrar claramente la relación entre las clases.

La relación entre las distintas clases también sufre cambios. El diseño inicial interactúa directamente con *Octant*, el cual interactúa, también de forma directa, con las otras clases; mientras que la integración del patrón visitante pone una capa de abstracción entre *Mesher* y *Octant*, pues *Mesher* interactúa sólo con los visitantes, y los ocho visitantes interactúan con *Octant* y el resto de las clases. Por otra parte, en la implementación simplificada, *Mesher* interactúa también con *Octant* de forma directa, y adicionalmente esta clase también interactúa

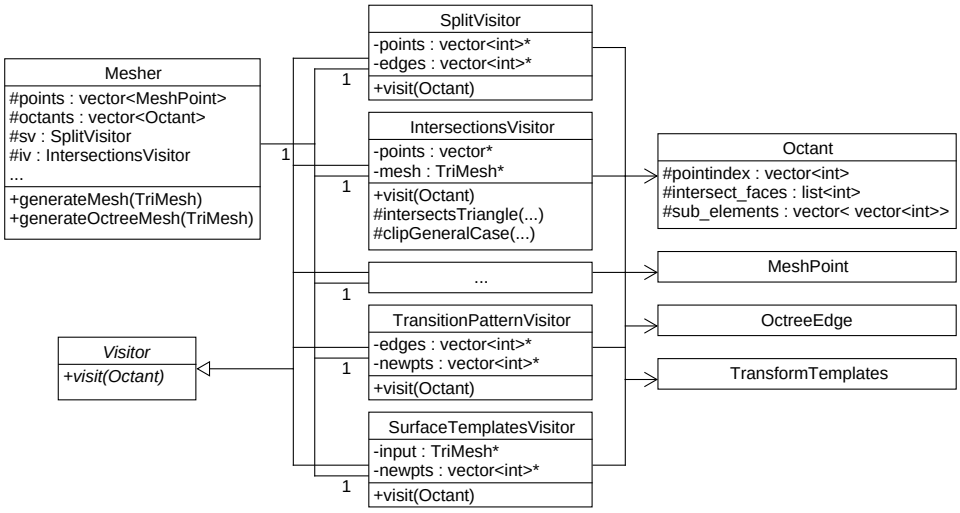
con los visitantes para ejecutar las funciones que sean necesarias.

Otra característica importante de los diagramas de la figura 4.4 es la cantidad de las funciones en *Octant*. Se puede ver que en la figura 4.4a *Octant* tiene muchas funciones, las cuales son repartidas entre los visitantes en los otros diagramas: en 4.4b se reparten todas las funciones, mientras que en la figura 4.4c quedan las funciones públicas, que sólo llaman a los visitantes.

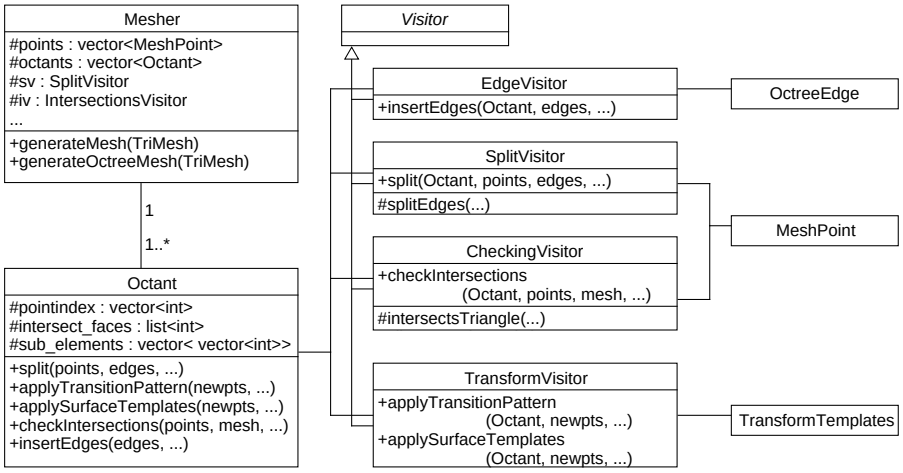
Si bien el tamaño y las conexiones de los diseños que integran el patrón visitante señala una complejidad mayor, el hecho de que las clases sean más pequeñas y que la funcionalidad de la aplicación se reparte en los visitantes ayudará a la mejor comprensión del código fuente, y a una mayor facilidad en la implementación de nuevos métodos.




(a) Diseño de `mesher` antes de la inclusión de visitantes. La clase *Octant* realiza todo.




(b) Diseño de `mesher` después de la inclusión de visitantes: *Octant* no tiene métodos complejos.




(c) Diseño de `mesher` utilizando visitantes simplificados y métodos llamados desde *Octant*.

Figura 4.4: Comparación entre diseños de la aplicación.

# Capítulo 5

# Implementación

Una vez que se define el nuevo diseño, integrando el patrón de diseño visitante, es necesario mover efectivamente los métodos entre las clases, cambiando los llamados a métodos de la forma que sea necesario, y comprobando que este movimiento no cambie el comportamiento de la aplicación. Sin embargo, existen varias maneras de implementar el patrón visitante, en particular su versión simplificada.

En este capítulo, se revisa cómo se movieron los métodos de *Octant* a los visitantes, los cambios en la forma de ejecutar estas operaciones, y de qué forma podrían agregarse nuevas operaciones a esta clase.

## 5.1. Extracción de métodos

Los métodos de *Octant* fueron copiados de forma íntegra en sus respectivos visitantes. Dependiendo del diseño, estos métodos fueron después borrados por completo, en el caso de la implementación completa del patrón visitante (en adelante, patrón "puro"), o reemplazados por llamadas a los visitantes, en el caso de la implementación simplificada (en adelante, patrón "simple")

### 5.1.1. Patrón puro

En la sección 4.3 se menciona que hay ocho métodos de la clase *Octant* que pueden ser encapsulados en su propia clase visitante, junto con cualquier otra función auxiliar que necesiten para realizar su función.

Todas las clases visitantes se componen de tres elementos:

- `visit`: Método principal. Este es una copia del método extraído de *Octant*, accesible públicamente.
- Variables: Junto con sus respectivos *setters*, equivalen a los parámetros que se entregaban al método de *Octant*.
- Métodos auxiliares: Son aquellos métodos a los que accede exclusivamente el método extraído de *Octant*, y son privados.

El hecho de que las variables con las que antes se llamaba el método pasen a formar parte del visitante, permite que estas sean pasadas al visitante una sola vez, acortando la firma (en inglés, *signature*) del método, y permitiendo que sea llamado varias veces, cambiando solamente las variables que sean necesarias: a diferencia de la llamada a `octant.split` original (listado 5.1, líneas 2 y 3) que debe tener en cuenta qué variables se usan a lo largo de toda la función, usar el visitante permite configurar las variables usadas en distintos momentos del código. (listado 5.2, líneas 2 a 7)

Notar que, en el diseño nuevo, es el octante el que recibe al visitante, tal como se especificó en la sección 4.3. En este caso, el método `octant.accept` recibe un `SplitVisitor` (listado 5.2, líneas 8 y 14); pero el método podría recibir cualquier tipo de visitante usando exactamente la misma firma.

Para poder modificar el octante, los visitantes deben tener acceso a los datos de este; esto se hace mediante *friend classes*, o "clases amigas", las cuales pueden acceder a datos marcados como protegidos (no privados). Debido a esto, cada visitante nuevo debe ser "registrado" en la clase *Octant*, pero esta es la única modificación que debe hacerse a la clase; todo lo demás queda contenido al visitante.

```
1  //llamada original
2  octant.split(points,new_pts,octreeEdges,
3               split_elements,clipping_coords);
4
5  //llamadas consecutivas requieren llamarlo todo de nuevo
6  octant.split(points,new_pts,octreeEdges,
7               split_elements,clipping_coords);
```

Listado 5.1: En *Mesher.cpp*: forma original de invocar una función de *Octant*.

```
1  //inicializar el visitante
2  SplitVisitor sv;
3  sv.setPoints(points);
4  sv.setEdges(octreeEdges);
5  sv.setNewPts(new_pts);
6  sv.setClipping(clipping_coords);
7  sv.setNewEles(split_elements);
8  octant.accept(&sv);
9
10 //luego se repite la llamada, y se cambia
11 //solo lo que sea necesario
12 sv.setClipping(clipping_coords);
13 sv.setNewEles(split_elements);
14 octant.accept(&sv);
```

Listado 5.2: En *Mesher.cpp*: forma de invocar una función de un visitante.

### 5.1.2. Patrón simple

Para llevar a cabo la implementación del patrón simplificado, se repartieron los métodos de todos los visitantes del patrón puro en cuatro visitantes, según lo mencionado en la sección 4.4.

- *CheckingVisitor*: Comprueba distintas condiciones de la malla geométrica. Consta de cuatro funciones principales y cinco funciones auxiliares.
- *EdgeVisitor*: Realiza operaciones sobre los bordes de la malla geométrica. Consta de dos funciones principales.
- *SplitVisitor*: Realiza la división de un octante. Consta de una función principal y dos funciones auxiliares.
- *TransformVisitor*: Aplica patrones de superficie y patrones de transición al octante. Consta de cuatro funciones principales y una función auxiliar.

A diferencia del patrón puro, estos visitantes se componen de dos elementos: los métodos principales, que son llamados por *Mesher*, y los métodos auxiliares, que son llamados sólo por los métodos principales.

Estos visitantes sirven como una suerte de contenedor para las funciones de *Octant*, y son llamadas por esta misma clase. El código en *Mesher.cpp* no cambia: las firmas son iguales en la versión original y en el rediseño simple, como se puede ver en los listados 5.1 y 5.3. Al igual que en el patrón puro, es necesario agregar los visitantes como clases amigas en *Octant* para que estos puedan realizar modificaciones al objeto.

### Integrar llamadas a los visitantes

Si bien el diseño creado en la sección 4.4 hace que *Octant* sea quien invoque las funciones implementadas en los visitantes, existen dos formas de implementar este comportamiento: instanciando los objetos visitante en un ámbito global, o declarando las funciones como estáticas y llamándolas directamente. Ambos tipos de implementación tienen ventajas y desventajas:

- Instanciar los visitantes como objetos globales tiene la ventaja de que se realiza una sola instanciación, que es utilizada a lo largo de toda la ejecución del programa, lo cual ahorra memoria y ciclos de creación y borrado de los objetos. Sin embargo, llamar

a objetos que fueron declarados en un archivo de código fuente diferente perjudica mucho la legibilidad del código, según el concepto de complejidad espacial (sección 2.1.3).

- Declarar funciones como estáticas corrige los problemas de legibilidad del código: se deja de llamar a un objeto declarado en un archivo distante de código fuente y se llama a la función de una clase en específico, más fácil de ubicar. Sin embargo, para ejecutar la función estática se instancia un objeto en cada ámbito que sea necesario, lo cual podría afectar el rendimiento por creación y borrado de estos objetos estáticos.

Como el patrón simple no considera que los visitantes guarden un estado, recibiendo todos los parámetros por referencia en la misma signatura, ambas implementaciones tienen capacidades equivalentes.

Realizar ambas implementaciones en *Octant* no requiere mucho esfuerzo adicional, por lo que se decidió hacer las dos implementaciones de forma separada, y compararlas según uso de memoria, tiempo de ejecución y métricas.

Dependiendo del tipo de implementación, la forma en que *Octant* llama al visitante cambia. La versión global llama a una función de un objeto `sv`, en la línea 6 del listado 5.4: este objeto es un *SplitVisitor* definido en un ámbito global, y al que tienen acceso todos los octantes.

Por otra parte, en la implementación estática no hay ningún objeto visitante como tal; como se ve en el listado 5.5, basta con incluir el archivo *SplitVisitor.h* y realizar la llamada a la función directamente, en la línea 8.

Notar que en ambas implementaciones, la firma de la función `split` de *Octant* y *SplitVisitor* difiere sólo en que la función del visitante recibe a un objeto *Octant*, aparte del resto de las variables.

```
1  octant.split(points,new_pts,octreeEdges,
2                split_elements,clipping_coords);
```

Listado 5.3: En *Mesher.cpp*: invocación de una función usando patrón simple. La firma no cambia respecto a la implementación original.

```
1  void Octant::split(vector<MeshPoint> &points,
2                     list<Point3D> &new_pts,
3                     set<OctreeEdge> &edges,
4                     vector< vector<unsigned int> > &new_eles,
5                     vector<vector<Point3D> > &clipping){
6      sv.split(this,points,new_pts,edges,new_eles,clipping);
```

Listado 5.4: En *Octant.cpp*: función de *Octant* llama a la función correspondiente en el objeto global visitante.

```
1  #include "Visitors/SplitVisitor.h"
2
3  void Octant::split(vector<MeshPoint> &points,
4                     list<Point3D> &new_pts,
5                     set<OctreeEdge> &edges,
6                     vector< vector<unsigned int> > &new_eles,
7                     vector<vector<Point3D> > &clipping){
8      SplitVisitor::SplitOctant(this,points,new_pts,
9                                edges,new_eles,clipping);
10 }
```

Listado 5.5: En *Octant.cpp*: función de *Octant* llama a la función visitante estática correspondiente.

## 5.2. Pruebas de integridad de la aplicación

Al mover métodos desde *Octant* a los visitantes, se espera que los métodos de *Octant* puedan reemplazarse directamente por llamadas a funciones de los visitantes y no realizar más cambios. Se tomaron dos medidas para asegurar la integridad de la aplicación después de cada reemplazo, es decir, que la aplicación siguiera teniendo la misma salida, dada una entrada estándar.

1. Se les agregó a los visitantes comandos para demostrar, mediante salida por consola, que sus métodos estaban efectivamente siendo llamados.
2. Se comparó el hash SHA256 de la malla volumétrica de salida del programa rediseñado con el del programa original; si estos hash son iguales, se asume que ambas mallas son iguales, y que se mantiene la integridad del programa.

Para probar cada cambio menor, se ejecutó el programa con parámetros definidos con anterioridad, utilizando un modelo de superficie proporcionado por el profesor Claudio Lobos, y generando una malla con un solo nivel de refinamiento.

Una vez terminadas las implementaciones, se comprobó el hash SHA256 de las mallas generadas para los modelos de superficie y niveles de refinamiento que se describen en la sección 6.1, comprobando así que las implementaciones nuevas producen los mismos resultados que la aplicación original.

## 5.3. Pruebas de extensión de la aplicación

El rediseño de la aplicación necesariamente va a cambiar la forma en que se agregan nuevas funciones para que sean ejecutadas sobre *Octant*; esto se pudo apreciar incluso en el proceso de implementación de los visitantes. Por esto, se puede hacer una comparación de cómo se podría agregar una funcionalidad nueva a la aplicación, que realice una operación sobre *Octant*.

Una manera de extender la funcionalidad del programa, según lo hablado con el profesor Claudio Lobos, es que la malla volumétrica sea generada en base a la intersección entre dos modelos de superficie. Esto implicaría cambios en las comunicaciones entre los objetos, para poner a disposición más de un modelo de superficie, y en los algoritmos de división de octantes y de aplicación de patrones de transformación.

Para el caso del diseño original, esto implicaría implementar funciones nuevas dentro de la clase *Octant*, las cuales serían llamadas en el momento que corresponda desde la clase *Mesher*. La complejidad del diseño no debiera aumentar; sin embargo, se agravaría mucho el problema de tamaño de *Octant*, pues no hay otra parte donde agregar estas funciones.

La implementación de patrón simple tiene el mismo problema de la implementación original, pero limitado a la cantidad de métodos de la clase; a *Octant* se agregan métodos que llaman a otros métodos en los visitantes, en los cuales se implementarían los algoritmos cambiados.

Por otra parte, para la implementación de patrón puro, no se realiza ninguna modificación a *Octant*, aparte de la adición de una clase amiga al encabezado de la clase; la inicialización del visitante puede realizarse con variables arbitrarias (objetos modelo, contenedores de estos objetos, etc.) y el llamado al visitante se hace exactamente igual que los demás visitantes.

# Capítulo 6

# Resultados

Una vez terminadas las nuevas implementaciones, se procedió a realizar pruebas de rendimiento y análisis de métricas a todas las versiones generadas del programa:

- `mesher_control`, que es la versión original de la aplicación.
- `mesher_visitor`, que implementa los visitantes de patrón puro.
- `mesher_global`, que implementa los visitantes de patrón simple como objetos globales.
- `mesher_static`, que implementa los visitantes de patrón simple mediante funciones estáticas.

Todos los resultados se caracterizan por el porcentaje de cambio en relación a la versión original `mesher_control`.

## 6.1. Condiciones

De acuerdo con lo mencionado en las secciones 3.1 y 3.2, las cuatro versiones deben ser probadas en términos de tiempo de ejecución, uso de memoria y análisis de métricas de

código fuente.

Se dispone de cuatro modelos de superficie:

- `all_bones`: Representa los huesos de pierna, tobillo y pie humanos. Es el modelo de mayor complejidad, debido a su composición, consistente de muchas partes aisladas. Es también el más pesado, hecho con 40 mil vértices y 80 mil triángulos.
- `bones`: Similar al modelo anterior, representa sólo los huesos del pie humano; también se compone de varias partes aisladas pero es más pequeño. Está hecho con 6900 vértices y casi 14 mil triángulos.
- `cortex`: Representa un cerebro humano; la forma arrugada del mismo hace que sea un modelo también complejo. Consta de 3100 vértices y 6300 triángulos.
- `palate`: Representa un paladar humano. Es el modelo más sencillo, ya que es pequeño y liso. Se compone de 1300 vértices y 2500 triángulos.

Para las pruebas de tiempo de ejecución, se generaron mallas volumétricas con los cuatro modelos, en niveles de refinamiento 3, 4, 5 y 6, tanto para todos los octantes (opción `-a`) como para los de superficie (opción `-s`). Cada malla fue generada cinco veces; se descartaron los casos extremos y entre ellos se obtuvo el promedio y la desviación estándar.

Del mismo modo, las pruebas de uso máximo de memoria se hicieron generando mallas volumétricas con los cuatro modelos, en niveles de refinamiento 5, 6 y 7, opciones `-a` y `-s`. Como se especificó en la sección 3.1, Massif tiene dos formas de medir uso de memoria; la medición se realizó con ambas formas.

Puesto que la obtención de métricas se realiza sólo con el código fuente, se utilizó CppDepend en las cuatro implementaciones, de acuerdo con la metodología especificada en la sección 3.3.3.

Las pruebas de tiempo de ejecución y memoria fueron realizadas en un computador con un procesador Intel Core i5 a 2.6 GHz, 8 GB de memoria RAM y sistema operativo Ubuntu 15.10 64-bit. El análisis de código fuente se realizó con CppDepend 6.0.0 Professional Edition para Windows.

## 6.2. Tiempo de ejecución

Resulta difícil establecer una tendencia clara con respecto al comportamiento de cada una de las versiones; es decir, ninguna de ellas obtiene resultados consistentemente más rápidos o más lentos para todos los modelos probados y todos los niveles de refinamiento, y las diferencias no son proporcionales, ni a la complejidad del modelo, ni al nivel de refinamiento. Los resultados están graficados en la figura 6.1, y los gráficos se encuentran en la misma escala.

Se observa que `mesher_visitor` obtiene tiempos casi siempre mayores que el diseño original, en la mayoría de los casos; esta implementación presenta también las mayores diferencias de tiempo: hasta 8 %, en el caso s-6 del modelo `all_bones`. Dos excepciones son los modelos `cortex` y `palate` en nivel de refinamiento a-6; sin embargo, este comportamiento no es distinto a las otras implementaciones.

Las diferencias de tiempo obtenidas para las otras dos implementaciones son mucho menores, en general; además, se da más seguido que el tiempo de ejecución es menor que la implementación original. Como se ve en los gráficos de la figura 6.1, las barras que más destacan son las de `mesher_visitor`; el resto de las diferencias, salvo contadas excepciones (`all_bones` a-6, `bones` a-6, `palate` s-3) oscilan entre -2 % y 2 %.

Notar que, en todos los casos, `mesher_visitor` obtiene resultados más lentos que `mesher_global` y `mesher_static`. En el caso de estas dos últimas implementaciones, las diferencias entre ellas son bastante cercanas, o al menos ambas versiones tienden a mejorar o empeorar los tiempos en relación a la implementación original. Sin embargo, en los casos más sencillos, resulta notorio su comportamiento diametralmente opuesto, como es el caso de `palate` a-3.

Una consecuencia de que no exista una clara tendencia en las diferencias de tiempo, es que en ninguna de las versiones parece escalar la diferencia de tiempo en proporción a la complejidad del par modelo-refinamiento. Más aún: en varios casos, las diferencias en los niveles de refinamiento a-6, a-7, s-6 y s-7 son menores que las diferencias del nivel inmediatamente inferior.


Figura 6.1: Diferencias en tiempo de ejecución de `mesher_visitor`, `mesher_global` y `mesher_static` respecto a `mesher_control`.

## 6.3. Uso de memoria

En muchos casos, las diferencias en uso de memoria de cada versión, medidas por *stack*, fueron de un valor numérico constante para todos los modelos y todos los niveles de refinamiento. Esto hace que la escala de los gráficos de la figura 6.2 no pueda ser única, pues los cambios en la proporción de uso de memoria varían mucho de modelo en modelo. Numéricamente, la diferencia de la mayoría de los casos fue de 8 KB para `mesher_visitor` y de 4 KB para las otras implementaciones.

Las excepciones son el modelo `all_bones` en nivel de refinamiento s-7, donde el uso de memoria es mayor en comparación con los otros niveles, y el modelo `palate` en todos los niveles de refinamiento, donde el uso de memoria es menor que la implementación original.

## 6.4. Métricas de calidad

Al igual que las diferencias en tiempo de ejecución y uso de memoria, se comparan las métricas de las implementaciones mediante porcentajes de cambio, destacando en rojo los cambios que empeoran el diseño y destacando en verde los que lo mejoran, según lo discutido en la sección 3.3.4. Estos datos se presentan en el cuadro 6.1.

Tal como se esperaba del rediseño, las métricas que describen el tamaño del diseño (cantidad de clases y cantidad de métodos) aumentaron, obviamente en mayor porcentaje en `mesher_visitor`, debido a que son más visitantes, cada uno con métodos específicos; por otra parte, el alto acoplamiento necesario entre *Mesher*, *Octant* y los visitantes hace que el acoplamiento promedio, tanto aferente como eferente, experimente un alza parecida a la de los cambios en la cantidad de clases.

Con respecto a las métricas relacionadas con la herencia, se esperaba que fueran mucho mayores en los diseños nuevos, debido a que la implementación original tiene muy pocas clases que hereden de otra, las cuales no están relacionadas directamente con *Octant*. En particular, la herencia funcional aumentó sólo en la implementación de patrón puro; el uso de herencia en `mesher_global` y `mesher_static` no involucra ninguna función, por eso es

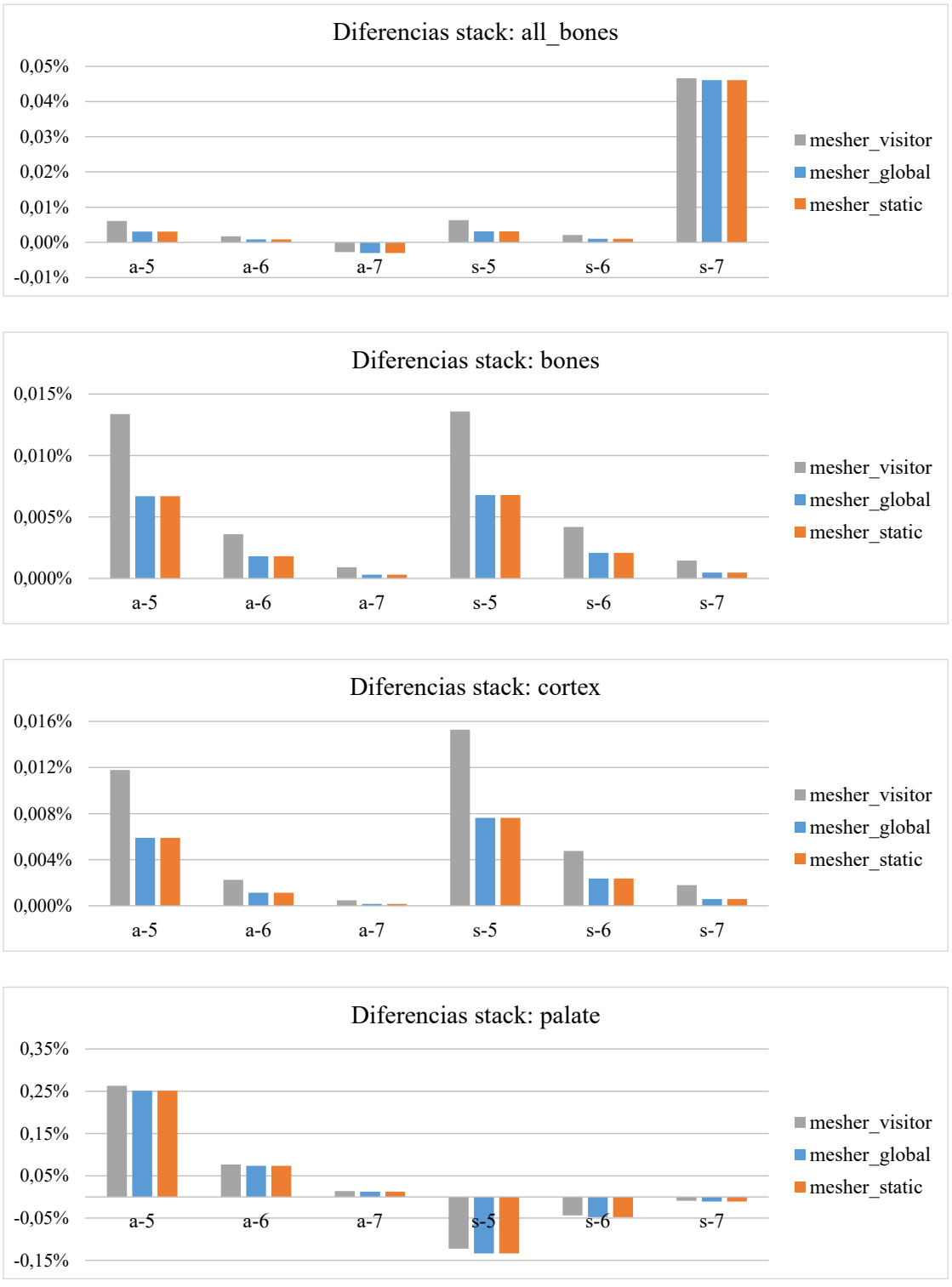



Figura 6.2: Diferencias en uso de memoria de `mesher_visitor`, `mesher_global` y `mesher_static` respecto a `mesher_control`, medido por uso de *stack*.

| Métrica | Abv. | Dif. global | Dif. static | Dif. visitor |
|---|---|---|---|---|
| Cantidad de clases | dsc | 18 % | 18 % | 27 % |
| Cantidad de métodos | nom | 10 % | 9 % | 14 % |
| Promedio de ancestros | ana | 67 % | 67 % | 133 % |
| Herencia funcional | mfa | 0 % | 0 % | 59 % |
| Encapsulamiento | dam | -1 % | 2 % | 6 % |
| Acoplamiento Ca promedio | ca | 26 % | 25 % | 28 % |
| Acoplamiento Ce promedio | ce | 6 % | 5 % | 7 % |
| Falta de cohesión promedio | lcom | 0 % | -6 % | 3 % |
| Cantidad de métodos polimórficos | nop | 9 % | 9 % | 23 % |

Cuadro 6.1: Diferencias en métricas a nivel de proyecto.

que no hay diferencia en esta métrica para aquellas versiones.

Puesto que la adición de los visitantes agrega más métodos al proyecto, y estos son, por lo general, de acceso protegido, esto resulta en una mejora en la métrica de encapsulamiento, con excepción del caso de `mesher_global`, donde el encapsulamiento se reduce, presumiblemente debido al hecho que los visitantes son instanciados globalmente.

La falta de cohesión es calculada en base al uso de las variables de la clase por los métodos de la misma: aunque los visitantes puedan tener una muy baja falta de cohesión, esto no alcanza a verse reflejado en el promedio del proyecto, en el caso de `mesher_visitor`. Cabe mencionar que la métrica asume que, en caso de que la clase no tenga variables locales, es completamente cohesiva; esto hace que el promedio de la falta de cohesión sea exactamente igual en `mesher_global` y baje en `mesher_static`.

Finalmente, en el caso del polimorfismo, se ve un aumento leve en las implementaciones de patrón simple, debido a la duplicación de funciones de *Octant* que ya eran polimórficas, y un aumento mayor en la implementación de patrón puro, que depende de polimorfismo para decidir qué visitante se ejecutará.

## 6.5. Análisis de resultados

### 6.5.1. Tiempo de ejecución

Si bien se esperaba que rediseñar la aplicación aplicando un patrón de diseño tuviera algún efecto en el tiempo de ejecución del programa, se esperaba que estas diferencias invariablemente aumentaran este tiempo. Notablemente, se obtuvieron algunos casos en los que el tiempo de ejecución disminuyó. Estas mejoras en el tiempo de ejecución se produjeron en los niveles de refinamiento más altos, para `cortex` y `palate`; notar que estos modelos son de una sola pieza, a diferencia de `all_bones` y `bones`, que se componen de varias partes separadas.

Para el resto de los casos, el comportamiento de la implementación de patrón puro resulta siempre peor que las implementaciones de patrón simple. Lo más probable es que esto se deba al uso de polimorfismo en el llamado a los visitantes: mientras que `mesher_visitor` decide todas las llamadas mediante herencia y polimorfismo, `mesher_global` y `mesher_static` definen todas las llamadas de forma directa, a costa de necesitar definirlas dentro de *Octant*. Claramente, estas definiciones permiten ganar tiempo en las llamadas a funciones, comportamiento que queda reflejado en los tiempos de ejecución de las implementaciones de patrón simple, siempre cercanos a la implementación original.

De todas formas, el aumento en el tiempo de ejecución que introduce la herencia y polimorfismo en la implementación de patrón puro, aunque no es despreciable, tampoco es incapacitante, siendo en todos los casos menor al 10 %. Y puesto que las aplicaciones de este programa no son en tiempo real, estas diferencias no son graves, especialmente si mejoran la mantenibilidad y extensibilidad.

Por otra parte, las diferencias menores obtenidas mediante el uso del patrón simplificado, sugieren un posible flujo de trabajo, donde se realiza la extensión y depuración de nuevas características en el diseño de patrón puro, para luego generar una versión optimizada del programa, sin la herencia y el polimorfismo.

### 6.5.2. Uso de memoria

Con respecto a los cambios en el uso de memoria, los resultados obtenidos son muy buenos: se obtuvo una diferencia invariable en la mayoría de los casos, y que aún más, no escala con cambios en la complejidad. Es posible afirmar que los cambios en el uso de memoria generados por la adición de nuevos objetos, en comparación con la gran cantidad de memoria que ya ocupa el programa para representar la malla volumétrica, son totalmente despreciables, aún en los casos en que estos cambios son mucho más altos de lo normal.

Ahora bien, no se sabe a ciencia cierta qué provoca estos cambios en el uso de memoria con respecto a otros casos, teniendo en cuenta que el comportamiento de la aplicación debería ser siempre igual. Por otra parte, estas diferencias se producen en las versiones de patrón puro y de patrón simple, sugiriendo que el aumento de memoria está vinculado al diseño de la aplicación, más que a un detalle de implementación.

Relacionado con estas diferencias vinculadas al diseño de la aplicación está el hecho de que, para los niveles de refinamiento `-s` del modelo `palate`, se registró una disminución de uso de memoria, de exactamente el mismo valor numérico, para las tres implementaciones. Es posible que asociada al diseño haya una reducción de memoria, relacionada con las clases *Mesher* y *Octant*, que no se cancele con las instanciaciones de los visitantes cuando se realiza refinamiento de superficie a una malla más sencilla. Sin embargo, es necesario cuestionar la utilidad de investigar las razones para esta reducción, teniendo en cuenta la reducida escala de la misma.

### 6.5.3. Calidad de software

Recordando las ecuaciones presentadas en la sección 3.3.1 (cuadro 3.2), se pueden transformar las variaciones en las métricas en variaciones de la extensibilidad, flexibilidad y capacidad de comprensión del diseño, como se ve en el cuadro 6.2. Se omitió la aplicación de la métrica de composición, y se tomó como acoplamiento el promedio entre acomplamientos aferente y eferente.

| Atributo | Dif. global | Dif. static | Dif. visitor |
|---|---|---|---|
| Comprensión | -40 % | -36 % | -70 % |
| Extensibilidad | 30 % | 30 % | 99 % |
| Flexibilidad | 0 % | 1 % | 8 % |

Cuadro 6.2: Aplicación del modelo de Bansiya y Davis a las diferencias entre versiones.

Esto se resume en los siguientes cambios:

- Se da una baja en la capacidad de comprensión del diseño, debido al aumento en la complejidad, tamaño de diseño, acoplamiento y polimorfismo.
- Los nuevos diseños son más extendibles (en mayor medida en `mesher_visitor`), pues integran herencia y polimorfismo.
- La flexibilidad de los diseños nuevos es mayor que el diseño original, también por el uso de polimorfismo.

Las cifras para cada atributo de calidad son de diferente escala; los cambios en la evaluación de las métricas de este trabajo hacen que los atributos no sean proporcionales.

La mejora de dos atributos de calidad, de entre los tres atributos seleccionados en la sección 3.3.1, reflejan que, efectivamente, las implementaciones que integran el patrón visitante, tanto en su versión simple como pura, suponen una mejora en la calidad de la aplicación rediseñada.

De forma inevitable, la integración del patrón visitante introducirá acoplamiento en el diseño general, debido al alto acoplamiento necesario entre *Octant* y los visitantes; sin embargo, se puede argumentar que el modelo de Bansiya y Davis evalúa las implementaciones de forma incompleta, al no tener en cuenta el tamaño de la clase al evaluar la complejidad del sistema.

El hecho de que las operaciones que se aplican sobre *Octant* estén apropiadamente categorizadas, en clases altamente cohesivas, da la impresión de facilitar el entendimiento de los pasos que sigue la aplicación para realizar su labor, pero es necesario reconocer que esta

conclusión es sesgada, debido al conocimiento de la aplicación que se obtuvo al categorizar e implementar las clases visitantes. Sería deseable, aunque está fuera del alcance de esta memoria, evaluar el entendimiento de las distintas implementaciones con sujetos de prueba, para comprobar si la complejidad agregada al diseño es compensada con una comprensión más rápida de los procesos ejecutados en la aplicación.

Por otra parte, las mejoras en extensibilidad y flexibilidad, derivadas de las mejoras de las métricas de herencia, respaldan las conclusiones de la sección 5.3: el uso de herencia permite agregar nuevas funcionalidades fácilmente, acotando los cambios que deban realizarse a las clases ya programadas.

Al final, será necesario comprobar empíricamente si las mejoras en extensibilidad y flexibilidad obtenidas mediante el rediseño de la aplicación compensan el aumento en complejidad del sistema, en el momento de realizar mantenciones en la aplicación.

# Conclusiones

El objetivo del presente trabajo fue claro: mejorar la calidad, en términos de facilidad de mantención y extensión, de una aplicación que implemente un algoritmo de alta complejidad como el de generación de mallas volumétricas. Sin embargo, también se quiso comprobar cuál es la penalización en rendimiento asociada a esta mejora.

Para mejorar la mantenibilidad y extensibilidad de la aplicación, se aplicó el patrón de diseño visitante a la clase *Octant* del programa, encargada de representar un conjunto de puntos en el espacio, mediante los cuales se genera la malla volumétrica. Se hizo esto porque era una de las clases con mayor tiempo de ejecución del programa, y también era una de las clases con mayor cantidad de responsabilidades. Aplicar el patrón visitante a esta clase permitió categorizar apropiadamente todas las operaciones, extrayendo los métodos más complejos a sus propias clases; de este modo, se logra simplificar una clase a la cual se le dedica una gran parte del tiempo de ejecución de la aplicación.

Para comprobar que la aplicación del patrón visitante pudo mejorar estos atributos, se definieron nueve métricas, respaldadas por distintos trabajos, para poder responder esta pregunta. Estas métricas modelan tres atributos, definidos por Bansiya y Davis, sobre los cuales se basó la evaluación de calidad de este trabajo: capacidad de comprensión, extensibilidad y flexibilidad.

Se implementaron dos versiones de un rediseño que integró el patrón visitante a la aplicación del profesor Claudio Lobos, y la comparación entre las nueve métricas definidas permite afirmar que se produjo efectivamente una mejora en la extensibilidad y flexibilidad de ambas

versiones; sin embargo, esto se logró mediante la integración de características como herencia y polimorfismo, y a costa de un aumento en el acoplamiento y en la falta de cohesión de la aplicación en su conjunto, disminuyendo, en las métricas, la capacidad de comprensión del sistema.

Considerando la mantenibilidad como una combinación entre flexibilidad y capacidad de comprensión del diseño, se puede afirmar entonces que el objetivo principal de la presente memoria, que es mejorar la mantenibilidad y extensibilidad de esta herramienta de generación de mallas volumétricas, se cumple; sin duda, la extensibilidad del programa ha mejorado mediante la introducción del patrón de diseño visitante, y en menor grado, también la mantenibilidad.

Resulta discutible el grado en se ve mermada la capacidad de comprensión del sistema en el diseño nuevo. Si bien es cierto que las métricas de Bansiya y Davis determinan que la comprensión del sistema debería ser menor, es necesario tener en cuenta las proporciones de la clase que se modificó. *Octant* es una clase que tiene una enorme cantidad de responsabilidades, y si bien un diseño sencillo, que integre todas las responsabilidades en una sola clase, puede parecer más fácil de entender, la escala de la clase en cuestión hace que este diseño sencillo sea demasiado difícil de aprehender en su totalidad; por esto, una categorización de las funciones de esta clase puede resultar en un diseño más fácil de entender, en contraste a lo que las métricas de diseño puedan sugerir.

Adicionalmente al mejoramiento de la mantenibilidad y extensibilidad de la herramienta, también se pudo comprobar, mediante pruebas de tiempo de ejecución y uso de memoria, de qué forma afectó al rendimiento de la aplicación la introducción de estas características: las estadísticas de ejecución obtenidas demuestran que el uso de memoria adicional es completamente despreciable, mientras que los tiempos de ejecución, si bien pueden sufrir una penalización, es de porcentajes inferiores al 10 %; además, implementaciones más sencillas del patrón visitante muestran incluso bonificaciones de tiempo de ejecución.

El diseño creado en este trabajo podrá ser utilizado por otros memoristas para integrar nuevas operaciones sobre los octantes de forma más rápida, facilitando también su comprensión del mismo, permitiéndoles realizar su trabajo de forma más eficiente.

# Trabajo futuro

## Pruebas empíricas de mantenibilidad y extensibilidad

La mejora en la mantenibilidad y extensibilidad de este diseño se basa principalmente en las métricas obtenidas del programa, por lo que realizar un estudio empírico de cuánto más fácil resulta mantener y extender la aplicación permitiría validar de mejor manera el trabajo realizado.

Mediante la participación de un grupo de estudio, se podría pedir que se integren nuevas características a la aplicación, y medir cuánto se demora el grupo en comprender el diseño y realizar la integración de la característica; también se podrían introducir defectos intencionalmente en la aplicación, y pedir al grupo de estudio que encuentre y corrija estos problemas.

Este grupo de estudio podría componerse de personas con distintos niveles de experiencia, con lo cual podría medirse cuánto influye la experiencia previa de una persona a la hora de entender un programa desconocido, y cómo ayudaría a esto la inclusión de un patrón de diseño, de manera similar a los experimentos mencionados en la sección 2.2.

## Refactorización de otros componentes

Aparte del conjunto de *Octant* y *Mesher* que fue rediseñado en la presente memoria, aún quedan una gran cantidad de clases que no fueron examinadas con el objetivo de refactorizarlas; estas clases pueden ser sujetas a un posible rediseño para que se ajusten a un patrón de diseño.

En particular, resaltan dos tipos de clases cuyo diseño podría ser examinado con más detenimiento: el conjunto de clases que controlan las regiones de refinamiento (característica de la aplicación que en este trabajo no fue considerada) y el conjunto de clases que aplican los patrones de superficie.

Las clases que controlan las regiones de refinamiento son la única estructura que utiliza herencia, aparte del conjunto de visitantes; por otra parte, las clases de patrones de superficie, que hasta el momento son 11, son aplicadas dependiendo de cuántos puntos del octante se encuentren dentro del modelo inicial. Este tipo de estructuras, que parecen tener altas posibilidades de expansión, podrían ser consideradas para ser rediseñadas siguiendo algún patrón de diseño para organizar múltiples objetos, como el patrón Decorador (patrón estructural) o el patrón Estrategia (patrón de comportamiento).

## Simplificación del diseño

Teniendo en cuenta la mejora en el tiempo de ejecución que se produjo al simplificar el patrón de diseño aplicado al programa, valdría la pena realizar una simplificación más ordenada de la implementación pura del patrón visitante.

Esto puede hacerse especificando en las clases *Mesher* y *Octant* exactamente qué visitantes realizan qué acciones, evitando el uso de polimorfismo en la elección del código a ejecutar. Es posible que hacer esto aumente el acoplamiento entre los visitantes y las clases *Mesher* y *Octant*, pero el evitar el uso de polimorfismo puede resultar en mejoras de rendimiento, sin que esto implique necesariamente que disminuya mucho la capacidad de extender el programa.

Más aún, se podrían desarrollar nuevas características para la aplicación utilizando el diseño de `mesher_visitor`, y realizar un paso adicional para especificar las acciones realizadas por los visitantes, antes de utilizar el programa en entornos de producción. Existe también la posibilidad de que este paso adicional pueda realizarse de forma automática; encontrar una manera de realizar esto con mínima intervención humana sería un gran aporte, al poder modificar un programa fácilmente mantenible y extensible, que luego se puede "precompilar" para que tenga un mayor rendimiento.

# Bibliografía


[1] Ampatzoglou, A. y A. Chatzigeorgiou: *Evaluation of Object-oriented Design Patterns in Game Development*. Inf. Softw. Technol., 49(5):445–454, Mayo 2007, ISSN 0950-5849. http://dx.doi.org/10.1016/j.infsof.2006.07.003.

[2] Arenas, Cristopher: *Detección y representación de características finas en mallas de volumen tipo octree*. Memoria para optar al título de Ingeniero Civil en Informática, Universidad Técnica Federico Santa María, Noviembre 2015.

[3] Bansiya, Jagdish y Carl G. Davis: *A Hierarchical Model for Object-Oriented Design Quality Assessment*. IEEE Trans. Softw. Eng., 28(1):4–17, Enero 2002, ISSN 0098-5589. http://dx.doi.org/10.1109/32.979986.

[4] Bastarrica, Maria Cecilia y Nancy Hitschfeld-Kahler: *Designing a Product Family of Meshing Tools*. Adv. Eng. Softw., 37(1):1–10, Enero 2006, ISSN 0965-9978. http://dx.doi.org/10.1016/j.advengsoft.2005.04.001.

[5] Buse, R. P. L. y W. R. Weimer: *Learning a Metric for Code Readability*. IEEE Transactions on Software Engineering, 36(4):546–558, July 2010, ISSN 0098-5589.

[6] Chhabra, Jitender Kumar y Varun Gupta: *Contemporary Computing: Second International Conference, IC3 2009, Noida, India, August 17-19, 2009. Proceedings*, capítulo Evaluation of Code and Data Spatial Complexity Measures, páginas 604–614. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, ISBN 978-3-642-03547-0. http://dx.doi.org/10.1007/978-3-642-03547-0_57.

[7] Chidamber, S. R. y C. F. Kemerer: *A Metrics Suite for Object Oriented Design*. IEEE Trans. Softw. Eng., 20(6):476–493, Junio 1994, ISSN 0098-5589. http://dx.doi.org/10.1109/32.295895.

[8] Chidamber, Shyam R., David P. Darcy y Chris F. Kemerer: *Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis*. IEEE Trans. Softw. Eng., 24(8):629–639, Agosto 1998, ISSN 0098-5589. http://dx.doi.org/10.1109/32.707698.

[9] Contreras, Felipe: *Adaptación de una Herramienta de Generación de Mallas Geométricas 3D a una Nueva Arquitectura*. Memoria para optar al título de Ingeniero Civil en Computación, Universidad de Chile, Octubre 2007. http://repositorio.uchile.cl/handle/2250/104666.

[10] Contreras, Felipe, Nancy Hitschfeld-Kahler, Maria Cecilia Bastarrica y Carlos Lillo: *Balancing Flexibility and Performance in Three Dimensional Meshing Tools*. Adv. Eng. Softw., 41(3):471–479, Marzo 2010, ISSN 0965-9978. http://dx.doi.org/10.1016/j.advengsoft.2009.10.005.

[11] Daines, Esteban: *Mejoramiento de la calidad de elementos mixtos en patrones de transición para la técnica octree*. Memoria para optar al título de Ingeniero Civil en Informática, Universidad Técnica Federico Santa María, Noviembre 2015.

[12] Dubey, Sanjay Kumar y Ajay Rana: *Assessment of Maintainability Metrics for Object-oriented Software System*. SIGSOFT Softw. Eng. Notes, 36(5):1–7, Septiembre 2011, ISSN 0163-5948. http://doi.acm.org/10.1145/2020976.2020983.

[13] Fontana, Francesca Arcelli y Stefano Spinelli: *Impact of Refactoring on Quality Code Evaluation*. En *Proceedings of the 4th Workshop on Refactoring Tools*, WRT '11, páginas 37–40, New York, NY, USA, 2011. ACM, ISBN 978-1-4503-0579-2. http://doi.acm.org/10.1145/1984732.1984741.

[14] Fowler, Martin, Kent Beck, John Brant, William Opdyke y Don Roberts: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[15] Gamma, Erich, Richard Helm, Ralph Johnson y John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[16] Gonzalez, Eugenio: *Diseño de patrones de transición entre zonas refinadas y gruesas de una malla de volumen de elementos mixtos*. Memoria para optar al título de Ingeniero Civil en Informática, Universidad Técnica Federico Santa María, Noviembre 2013.

[17] Hitschfeld, N., C. Lillo, A. Caceres, M.C. Bastarrica y M.C. Rivara: *Building a 3D Meshing Framework Using Good Software Engineering Practices*. En Ochoa, SergioF. y Gruia Catalin Roman (editores): *Advanced Software Engineering: Expanding the Frontiers of Software Technology*, volumen 219 de *IFIP International Federation for Information Processing*, páginas 162–170. Springer US, 2006, ISBN 978-0-387-34828-5. http://dx.doi.org/10.1007/978-0-387-34831-5_13.

[18] Jensen, Adam C. y Betty H.C. Cheng: *On the Use of Genetic Programming for Automated Refactoring and the Introduction of Design Patterns*. En *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, páginas 1341–1348, New York, NY, USA, 2010. ACM, ISBN 978-1-4503-0072-8. http://doi.acm.org/10.1145/1830483.1830731.

[19] Khomh, F. y Y. G. Gueheneuc: *Do Design Patterns Impact Software Quality Positively?* En *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, páginas 274–278, April 2008.

[20] Kolb, R., Dirk Muthig, T. Patzke y K. Yamauchi: *A case study in refactoring a legacy component for reuse in a product line*. En *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, páginas 369–378, Sept 2005.

[21] Lee, Jackson Weiting: *Studies on Software Comprehensibility*. Tesis de Licenciatura, The University of Western Australia, 2009.

[22] Li, Wei y Sallie Henry: *Object Oriented Metrics Which Predict Maintainability*. Informe técnico, Department of Computer Science, Virginia Polytechnic Institute, 1993.

[23] Lobos, C. y E. González: *Mixed-element Octree: a meshing technique toward fast and real-time simulations in biomedical applications*. International Journal for Numerical Methods in Biomedical Engineering, 31(12):1–31, 2015.

[24] Lobos, Claudio: *A Set of Mixed-Elements Patterns for Domain Boundary Approximation in Hexahedral Meshes.* 184:268–272, 2013. http://dblp.uni-trier.de/db/conf/mmvr/mmvr2013.html#Lobos13.

[25] Meagher, Donald: *Geometric modeling using octree encoding*. Computer Graphics and Image Processing, 19(2):129 – 147, 1982, ISSN 0146-664X. http://www.sciencedirect.com/science/article/pii/0146664X82901046.

[26] Ng, T. H., S. C. Cheung, W. K. Chan y Y. T. Yu: *Work Experience Versus Refactoring to Design Patterns: A Controlled Experiment*. En *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, páginas 12–22, New York, NY, USA, 2006. ACM, ISBN 1-59593-468-5. http://doi.acm.org/10.1145/1181775.1181778.

[27] Palma, Francis, Hadi Farzin, Yann Gaël Guéhéneuc y Naouel Moha: *Recommendation System for Design Patterns in Software Development: An DPR Overview*. En *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering*, RSSE '12, páginas 1–5, Piscataway, NJ, USA, 2012. IEEE Press, ISBN 978-1-4673-1759-7. http://dl.acm.org/citation.cfm?id=2666719.2666720.

[28] Sfetsos, P., A. Ampatzoglou, A. Chatzigeorgiou, I. Deligiannis y I. Stamelos: *A Comparative Study on the Effectiveness of Patterns in Software Libraries and Standalone Applications*. En *Quality of Information and Communications Technology (QUATIC), 2014 9th International Conference on the*, páginas 145–150, Sept 2014.

[29] Shahir, H.Y., E. Kouroshfar y R. Ramsin: *Using Design Patterns for Refactoring Real-World Models*. En *Software Engineering and Advanced Applications, 2009. SEAA '09. 35th Euromicro Conference on*, páginas 436–441, Aug 2009.

[30] Shatnawi, Raed y Wei Li: *An Empirical Assessment of Refactoring Impact on Software Quality Using a Hierarchical Quality Model*. International Journal of Software Engineering & Its Applications, 5(4):127, October 2011.