Advances in Engineering Software 41 (2010) 471–479

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

Balancing flexibility and performance in three dimensional meshing tools

Felipe Contreras, Nancy Hitschfeld-Kahler *, María Cecilia Bastarrica, Carlos Lillo

Computer Science Department, Facultad de Ciencias Físicas y Matemáticas, Universidad de Chile, Blanco Encalada 2120, Santiago, Chile

ARTICLE INFO

Article history:
Received 20 October 2008
Accepted 2 October 2009
Available online 30 October 2009

Keywords:
Meshing tools
Object-oriented design
Software architecture

ABSTRACT

It is generally thought within the meshing tool community that object-orientation and other decoupling techniques penalize performance when they are used for building concrete meshing tools. In this paper we show that building a meshing tool with good object-oriented design metrics could not only improve maintainability and all other derived attributes such as portability and extensibility, but also its performance is comparable to a standard meshing tool that implements the same algorithms.

1. Introduction

Meshing tools are generally used for modeling a variety of physical phenomena such as the structure of the human brain, fluid mechanics, tree growth, meteorology and geographical information systems, among others. These tools are sophisticated pieces of software due to the complex algorithms and data structures they use, the huge number of elements they involve, and the specific and highly specialized contexts where they are used. There are 2D meshes for modeling planar surfaces, 2.5D meshes for modeling three dimensional surfaces, and 3D meshes for modeling volumes.

The meshing tool community has generally focused on performance for many years, mainly efficient CPU time and storage management, but lately maintainability has also become important. Achieving both attributes, performance and maintainability, simultaneously seems to be contradictory. In fact, there are some popular 3D meshing tools, such as TetGen, that are developed using object-oriented languages, but they do not usually take full advantage of the maintainability that this technology may provide because it is thought that it would strongly worsen performance.

In this paper we show that this seeming contradiction is not always the case, and that a balance between flexibility and maintainability can be achieved. We built a meshing tool by designing its structure according to good object-oriented design patterns so that it had good maintainability metrics. Then, each component was implemented using the same algorithm as another well established meshing tool that did not follow the same structure. In practice this is a refactoring of the existing tool: the new one provides exactly the same functionality but some quality attributes are enhanced. In a previous work, we had already developed a meshing framework for generating quality 3D meshes that had good object-orientation properties [11]. That framework provides Delaunay and Lepp [21] based algorithms. In this work we take that structure as a starting point, we improve it and we provide the possibility of using the algorithm implementations included as part of TetGen [24]; these algorithms include a more robust version of the Delaunay [25] algorithms as well as Delaunay refinement algorithms [23].

We obtained a versatile 3D tool called MeshingToolGenerator. This powerful tool was built in no more than nine months by a masters student [4]. The structural metrics are now better than those in the previous work and much better than those in TetGen, so we expect that it is more maintainable and flexible. Also performance was tested using a set of varying complexity examples, and the results were similar to those in TetGen, both with respect to memory usage and response time.

The paper is structured as follows: In Section 2 we present a series of related work. The MeshingToolGenerator analysis, design and implementation is described in Section 3. The evaluation of the structure and the performance of the developed framework is presented in Section 4. Finally some conclusions and future work are discussed in Section 5.

2. Related work

2.1. 3D tetrahedral meshing tools

A mesh is a discretization of a domain geometry that represents the object to be modeled or simulated. Meshing tools generate and manage meshes that are useful for solving partial differential equations numerically or for visualizing objects. In 3D, different

* Corresponding author. Tel.: +56 2 6784365; fax: +56 2 6895531.
E-mail addresses: fcontrer@dcc.uchile.cl (F. Contreras), nancy@dcc.uchile.cl (N. Hitschfeld-Kahler), cecilia@dcc.uchile.cl (M.C. Bastarrica), clillo@dcc.uchile.cl (C. Lillo).

meshing tools vary in the type of the elements they manage; the most widely used are tetrahedral and hexahedral meshes.

There are several 3D tetrahedral meshing tools currently available but not all of them provide the same functionality [19]; they vary depending on the application for which they were designed.

Three examples of well known 3D tetrahedral meshing tools are TetMesh, QMG and TetGen.

TetMesh [10] is a widely known commercial product for the generation of quality tetrahedral meshes for finite element methods. It was originally developed in FORTRAN 77 and afterward migrated to C.

QMG [17] is an open source octree based mesh generator for automatic unstructured finite element mesh generation. It was developed in C++ and Tcl/tk using object-orientation concepts. It uses octrees as the main data structure, thus all algorithms conform to this structure, yielding an efficient yet highly coupled tool.

TetGen [24] is a very efficient and robust open source tool for the generation of quality Delaunay meshes for solving partial differential equations using finite element and finite volume methods. TetGen has been developed using C++, but not necessarily incorporating object-oriented concepts, since it is implemented using only 10 classes without using information hiding, encapsulation, and polymorphism.

In general, all mesh generation tools are focused on reaching efficiency and robustness and not necessarily extensibility, flexibility or modifiability.

Each of these tools, and others as well, have particular problems they are intended to solve. Adding new features or improving existing ones is generally difficult due to the high complexity of the tools even though the algorithms to be included may be completely implemented. We intended to tackle the problem of making it easier to extend a 3D meshing tool by using good object-orientation design practices and thus obtaining higher flexibility. We used TetGen as a source of algorithm implementations and also as a benchmark for performance evaluation.

2.2. Software engineering practices in developing meshing tools

It is not new the idea of applying good software engineering practices in the development of meshing tools since they are software too. In [7] the use of formal methods for improving reliability of mesh generation software has been addressed using algebraic specifications, and lately an approach for using these specifications has also been proposed [29] for formal verification.

In [15], an object-oriented approach for developing finite element software is presented. Here, encapsulation and information hiding are used as a means for leveraging the level of the abstractions that form part of the systems under development but not necessarily as building blocks as architectural components. In 1999, the Engineering with Computers journal devoted a complete issue to object technology used in the development of engineering software [2]. However, none of the articles included relate specifically to meshing tools.

Object-orientation has also been applied within the meshing tool community. Some of the interesting published work include the design of object-oriented data structures and procedural classes for mesh generation [18], the computational geometry algorithm library CGAL [8], and the definition of an optimal object-oriented mesh representation that allows the programmer to build efficient algorithms [20].

However, as complexity grows and the need for evolving systems is always present, a new approach using software architectural concepts becomes necessary.

In this direction, the Common Component Architecture (CCA) has been developed [13]; it is a modular stack of technologies that provides a Scientific Interface Definition Language (SIDL) that allows the user to define how components interact. CCA allows to maximize flexibility in new code, to combine legacy code by wrapping it using common interfaces, and to configure required applications using the framework.

SCIRun [12] is a problem solving environment developed using CCA; it is a workbench in which a scientist can design and modify simulations interactively via a component-based visual programming. Our approach follows a similar philosophy but it is applied specifically in the process of building tools for mesh generation.

There have also been some attempts in using software product lines (SPL) concepts for building meshing tools [1,27]. Smith and Chen have applied SPL to the meshing tool domain [28] essentially using FAST [30]. Even though their approach is systematic, they are not taking advantage of the meshing tool domain characteristics probably because a general method for domain analysis for scientific computing software is applied [26].

3. Framework development

We developed a framework that implements the same functionality as TetGen but also provides some other quality attributes such as maintainability and flexibility. We achieved this by redesigning the tool architecture but reusing the algorithm implementations contained as part of TetGen, taking advantage of the fact that it is an open source tool.

3.1. Requirements and analysis

Required functionality and expected quality attributes are both considered as a basis for the framework development.

3.1.1. Functional requirements

Any 3D mesh generation framework should implement each one of the following processes:

- read the input geometry;
- generate an initial volume mesh that fits the domain geometry;
- refine a mesh in order to satisfy a refinement criterion;
- improve a mesh according to certain quality criteria;
- smooth the mesh according to certain smoothing criterion;
- derefine a mesh according to derefinement criteria;
- evaluate the quality of the generated mesh;
- visualize the mesh.

The specification of the input geometry and physical values can be generated by different CAD programs or by other mesh generation tools. The algorithms that generate the initial volume mesh can receive as input the domain geometry described as a triangulated surface mesh or as a general polyhedron. The initial volume mesh is the input of the refinement step that divides coarse tetrahedra into smaller ones until the refinement criteria are fulfilled in the indicated region. Either the initial volume mesh or the refined mesh can be the input of the improvement process. The user must specify the improvement criterion and a region where the improvement is to be applied. The smoothing and derefinement processes are also applied according to a criterion and over a region of the domain. Once a mesh has been refined, improved, derefined and/or smoothed, the user has the possibility of evaluating the mesh quality according to different criteria. This is very useful if the user wants to see the distribution and percentage of good and bad elements in the mesh. Each mesh generation process can also be skipped by representing it with a dummy algorithm. At any time the mesh may be visualized using either an external or a built in visualization tool.

3.1.2. Quality attributes

Not only functionality is required in order to achieve a successful meshing tool, but also some quality attributes.

3.1.2.1. Maintainability. From all the processes just described, our framework should be able to easily evolve mainly in the following directions:

- add the possibility of reading and writing new data formats;
- add or modify strategies for the generation of initial volume meshes;
- add or modify strategies for the refinement, improvement, smoothing and derefinement of tetrahedral meshes;
- add refinement and improvement criteria;
- add new shapes for the regions where a criterion must be respected.

These are the expected variations of our framework. The incorporation of a new strategy, criterion or region shape should have a minimal impact on the tool implementation, and we should provide a design that considers these requirements.

3.1.2.2. Performance. Both, the response time and memory usage of the framework should not be more than 10% higher than those in TetGen for any of the implemented processes. This is because we not only want to build yet another tool with the same functionality than an existing one, but also one that has acceptable qualities for the meshing tool community.

3.2. Design and implementation

First, we describe the general structure of the framework and how TetGen algorithm implementations have been adapted to this structure. Then we describe each specific component within the design.

Fig. 1 shows the most important part of the meshing framework class diagram. We represent each mesh generation process as an abstract class (in italics in the diagram). Fig. 2 shows the design of MeshingToolGenerator that adapts and encapsulates the algorithms included in TetGen. Each different strategy that implements each process is represented in Fig. 2 as a concrete subclass. For example, the *Refine* abstract class is realized by two subclasses: `DelaunayRefinementTetGen` and `ManualRefinementTetGen`; they both represent different refinement strategies. In the case of TetGen, the Refinement algorithm used is also used as an improvement algorithm, so we followed the same philosophy in our framework, and we omit *Improve* as an independent class.

We represented all the criteria with the *Criterion* abstract class and all the region shapes with the *Region* abstract class in Fig. 1. This allows a programmer to add a new criterion or region shape by adding just a concrete class that inherits from the respective abstract class and without modifying the source code. As TetGen does, we have provided two different subclasses for the `Criterion` abstract class: `RadiusEdgeRatio` and *VolumeLongestEdge3Ratio*, respectively, as can be seen in Fig. 2. As TetGen does not manage the *Region* concept for refinement, our framework provides an abstract class for defining the refinement region, but it is always applied to the whole geometry; however, this can be extended without much effort.

If we want to evolve the framework in order to generate a particular meshing tool, we should use the abstract classes' code without any modification, and we must select which concrete algorithms we want to use for each mesh generation strategy, criterion and region shape. For example, *GenerateVolumeMesh* can be realized by `RobustDelaunayTetGen`. Similarly, we already have `OffFormatTetGen`, `PolyFormatTetGen`, `MeditFormatTetGen` and `SMeshFormatTetGen` as concrete implementations for *InputOutput*, as well as `TetMeshFormat` a typical format used by TetMesh.

The mesh is modeled as a container object that holds the mesh information, and it is a commonality of the complete framework, i.e. that is to be included in any evolution of the framework. The `MeshTetGen` class provides methods for accessing and modifying

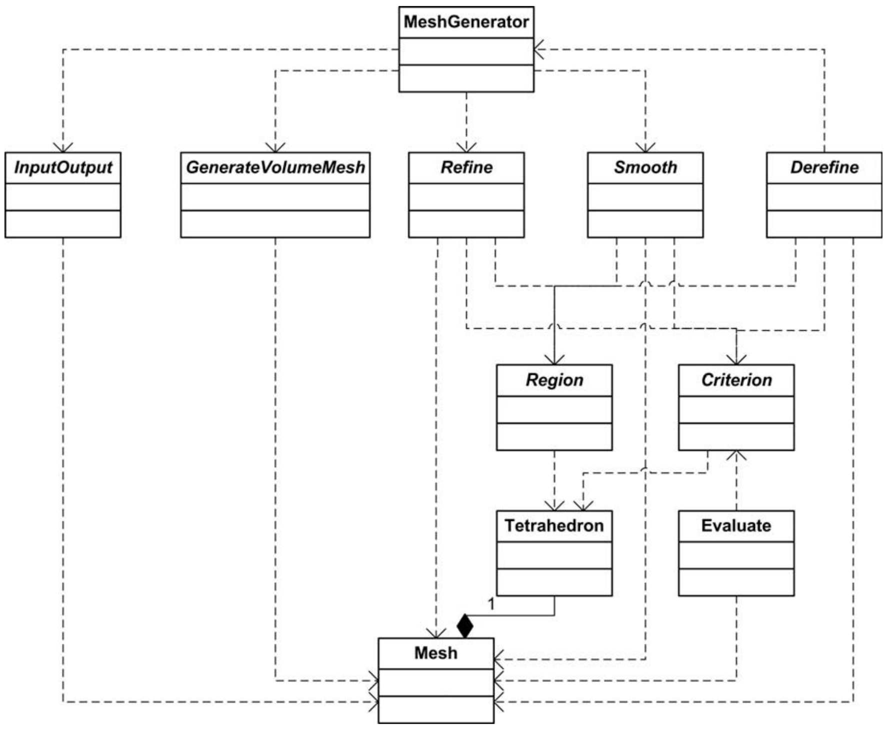

Fig. 1. Framework general class diagram.

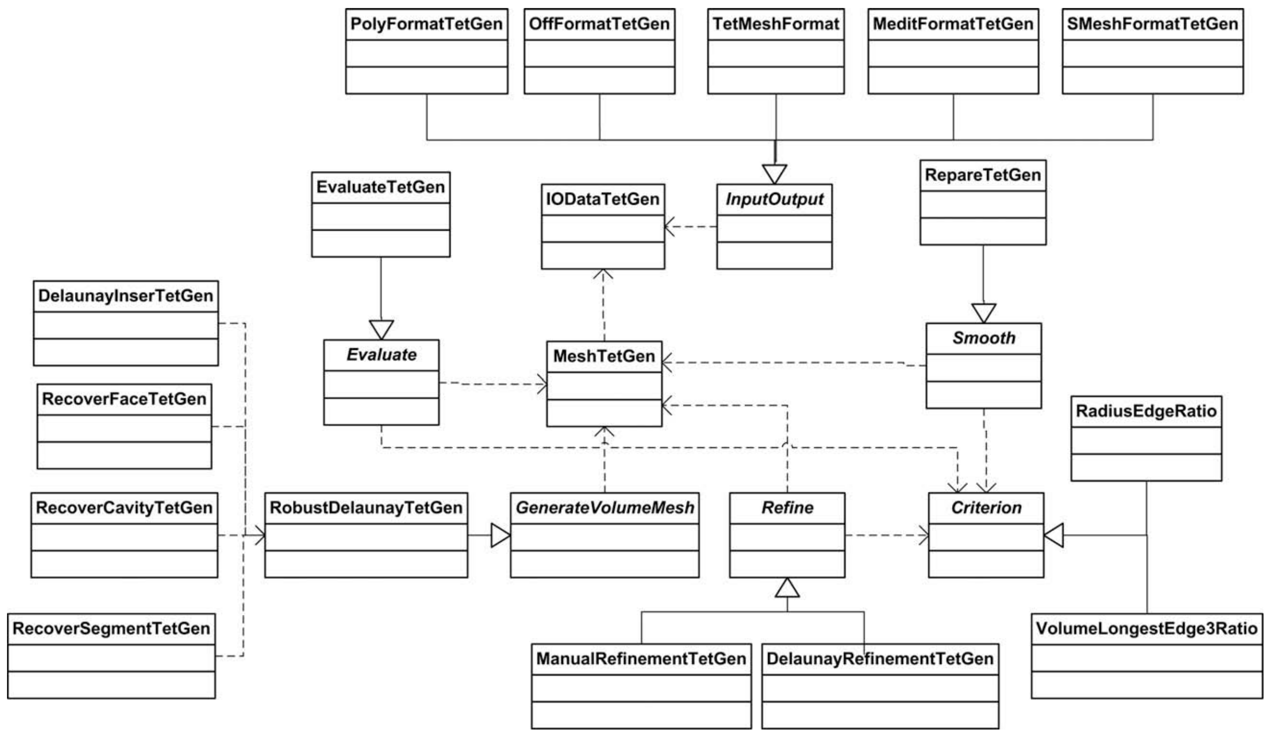

Fig. 2. Partial detailed class diagram.

its constituent elements (tetrahedra, faces, edges and points). `Tetrahedron`, `Face`, `Edge` and `Vertex` are also classes, each of them providing concrete functionality and also providing access to the neighborhood information within the mesh (not shown in Fig. 2 for clarity).

The mesh quality evaluation is also modeled as an object by using the `EvaluateTetGen` class, a subclass of the *Evaluate* abstract class. This class uses a criterion and, according to some user parameters, it classifies the elements and generates a file with the evaluation results as output.

In this framework implementation, we have used several design patterns to model different parts of the system [9]. The Adapter pattern was used all over the application because we needed to adapt the interfaces provided by TetGen to the interfaces required by our framework. Also, each different mesh generation process and each criterion follows the philosophy of the Strategy pattern. The mesh evaluation class follows the Observer pattern where the observed object is the mesh. The interface is organized using the Command pattern. The mesh is a Singleton.

4. Framework evaluation

We needed to balance flexibility and performance, so we evaluated our MeshingToolGenerator framework and TetGen according to both quality attributes. First we provide a series of well established metrics for object-oriented code that allow us to evaluate the tools from a flexibility and maintainability point of view, and we compare the results for both applications. Next we present the results of applying a series of performance tests to both applications.

4.1. Design evaluation

Metrics for object-oriented design provide a quantitative mechanism for estimating design quality. The main goals of these metrics are better understanding product quality, estimating the development process effectiveness, and improving the quality of the work done. In object-orientation, design metrics relate to object definition (OD), attributes (A) and communication between objects (C) [6]. In this work, we use the metrics proposed by Chidamber and Kemener [3] because they are widely used for measuring flexibility and extensibility. A brief description of each metric is included in Table 1, as well as their classification according to the aforementioned categories.

4.1.1. TetGen

Fig. 3 shows the complete class diagram for TetGen. As can be seen, there are only a few classes implementing the whole application. Applying the defined metrics to TetGen we obtained the values contained in Table 2.

TetGen has a high average value in WMC and also a high standard deviation because it has only a few classes, some of them with a lot of methods and others with only a few methods. This situation hinders understandability and thus also maintainability, one of our goals.

Only mesh object queues are managed with a hierarchy in TetGen. This is shown with a very low value of the DIT metric. Similarly, NOC is only one, showing that inheritance is used in one case and only for adapting a class and not for specializing a concept. Clearly, even though TetGen is developed using C++ that is an object-oriented language, its design does not take full advantage of the object-orientation philosophy of the language.

The CBO metric shows a low value for TetGen's classes, probably influenced by the fact that there are few classes in the system. Among them, the class that implements the mesh – `tetgenmesh`, defines internally the most accessed classes such as `face`, `metric` or the whole `memorypool` hierarchy, yielding a low coupling. However, it should be noticed that the average CBO is high if compared to the number of classes in the system. The maximum CBO is 14 and corresponds to the `tetgenmesh` class, meaning that it is coupled with almost all other classes.

Table 1
Design metrics.

Name	Description	Classification
Weighted Methods per Class (WMC)	Sum of all method's complexity within a class. The number of methods and their complexity indicate the effort required for implementing a class. Also, the larger the number of methods the more specific a class becomes, limiting its reusability	OD
Depth of Inheritance Tree (DIT)	Maximum length between a node and the root in the inheritance tree. The deeper the class, the more probable the class inherits a lot of methods favoring reuse. However, a deep class hierarchy may imply a complex design	OD
Number of children (NOC)	As the number of children grows, the abstraction represented by a class becomes more relevant, and its reusability increases, so it may require more rigorous testing	OD
Coupling Between Objects (CBO)	It is the number of collaborations between a class and the rest of the system. As this number grows, the class reusability decreases. High values also make modifications and testing harder	C
Response for a Class (RFC)	It is the number of methods that may be potentially executed as a response to a message received by a class object. As this metric grows, testing the class becomes harder, and the class complexity also grows	A, C
Lack of Cohesion in Methods (LCOM)	A high LCOM indicates that methods can be grouped in disjoin sets with respect to attributes, and the class should be partitioned in two or more classes	A
Public Variables per Class (PVC)	It is the number of public instance variables declared as part of a class. A high PVC shows a poor class encapsulation	OD
Lines of Code per Class (CLOC)	Having large classes implies that changes are harder to localize, so for more maintainable code this metric should remain small	OD
Lines of Code per Lines of Comments (%COMM)	Commented code is more understandable and thus more easily maintained. A larger code is a priori more complex, but if it is longer just because of the amount of comments it is not bad. %COMM is the percentage of lines of code per lines of comment	OD
Method Cyclomatic complexity (MCC)	The method cyclomatic complexity counts the number of linearly independent paths that a particular method exhibits	OD
Class Cyclomatic complexity (CCC)	The cyclomatic complexity indicates the testing complexity of a certain piece of code. It is the average number of linearly independent paths that the methods of a class exhibit	OD, C

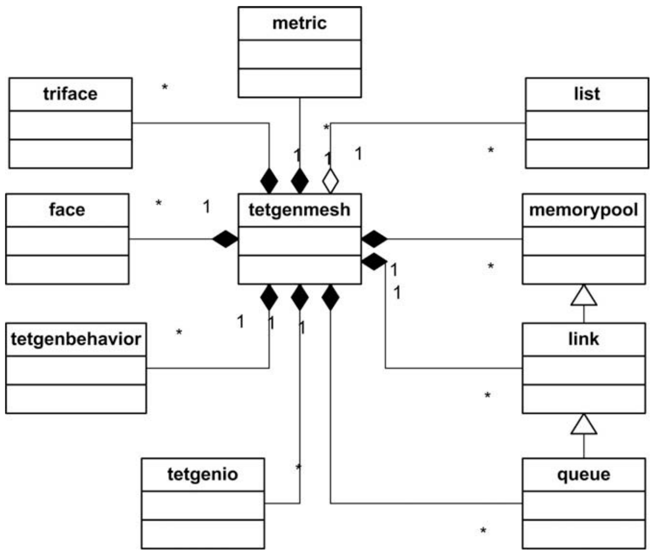

Fig. 3. TetGen class diagram.

In TetGen there are some classes with a very low response capacity and one, `tetgenmesh`, with a huge response. This situation yields a RFC metric with a very high standard deviation that again produces difficulties in maintainability.

The cohesion within TetGen classes is very low (LCOM). In general it is expected that if we have classes with large amounts of methods, not all of them are related. A better design would separate them into classes that group methods and attributes taking better care of their cohesion.

TetGen defines all its variables as public showing absolutely no encapsulation. So PVC is always high.

The number of lines per class varies considerably. While there are huge classes with very high CLOC, there are some others very specialized and small. Large classes are less reusable, and even though small classes may have the potential to be more reusable, the savings achieved in this reuse is limited.

TetGen is well documented with a line of comment every 24 lines of code. This makes it understandable.

The maximum class cyclomatic complexity is not very high because class `tetgenmesh` has a large number of methods; although there are highly complex methods in this class, as apparent in MCC, the average complexity for the class is also influenced by many other simple methods. This makes the class difficult to maintain because debugging is more difficult in these circumstances. The method cyclomatic complexity on the other hand is high in average and its standard deviation is even higher.

4.1.2. MeshingToolGenerator

Applying the same metrics to MeshingToolGenerator we obtained the values contained in Table 3.

MeshingToolGenerator has a low average value in WMC, showing a cleaner design. However, even though the standard deviation value is also much lower than that of TetGen, it is still somehow high mainly because the `MeshTetGen` class still has a very high number of methods (182). Nevertheless, the original `tetgenmesh` class in TetGen had 366 methods, so the decoupling effort allowed us to improve this metric. `MeshTetGen` still manages all the operations relating basic geometric elements; it remains as future work to decouple these functions.

Inheritance is used in a more appropriate way in MeshingToolGenerator. So DIT is improved mainly due to the use of inheritance

Table 2
TetGen design evaluation according to OO metrics.

	WMC	DIT	NOC	CBO	RFC	LCOM	PVC	CLOC	%COMM	CCC	MCC
Minimum	4	0	0	0	4	0	0	9	0	1	1
Maximum	366	2	1	14	366	97	103	18,930	24	14	86
Average	45.6	0.3	0.2	2.4	49.4	45.2	21.2	2212.3	14.4	4.6	9.2
Std. dev.	112	0.7	0.5	4.4	111.1	36.5	32	5908.4	7.1	6.0	36.3

Table 3
MeshingToolGenerator design evaluation according to OO metrics.

	WMC	DIT	NOC	CBO	RFC	LCOM	PVC	CLOC	%COMM	CCC	MCC
Minimum	1	0	0	0	1	0	0	4	0	0	1
Maximum	182	2	9	23	182	100	103	3697	48	31	86
Average	7.9	0.5	0.4	5.4	11.2	19.1	3.6	304.8	12.5	6.8	7.6
Std. dev.	20.9	0.6	1.2	5.4	21.1	28.7	14.0	551.1	11.6	7.0	11.6

in those places where the tool may need to be extended such as different criteria, input formats and meshing algorithms. Even though the maximum depth of the hierarchies did not increase, the average depth is higher than that in TetGen because inheritance was used several times to model the potential variabilities of the meshing tool. Similarly, NOC is also increased mainly by decoupling into different subclasses some methods that were originally coded as part of the same class. This also shows a better use of inheritance, and even though MeshingToolGenerator only includes those functionalities already in TetGen, with the new structure it would be easier to add new functionality in the form of a new child to a hierarchy than before.

The CBO metric grew in MeshingToolGenerator because there are more classes in the whole design than in TetGen. However, its average value is reasonable with respect to the size of the complete system. `TetGenMesh` is still the most coupled class (23), but this value is less than one third of the total number of classes in the system (almost 80). In TetGen the most coupled class had two times the number of classes in the system.

In MeshingToolGenerator the average RFC metric is much lower than before. Now the class representing the mesh – `MeshTetGen` – has fewer methods even though it is still high. The standard deviation of RFC also decreased significantly showing a more symmetric design.

The average lack of cohesion within MeshingToolGenerator classes is much lower than that in TetGen, showing a more cohesive design in general. Moreover, if we take a closer look, we can see that we are now using abstract classes that yield 100% lack of cohesion, and this situation increases the average even though it is not a bad design decision; these classes are also those responsible for the high standard deviation.

The PVC metric decreased dramatically in MeshingToolGenerator showing a better class encapsulation. Nevertheless, an optimal encapsulation still requires improving this metric even more.

The average number of lines of code per class is still high, but what is even worse is the high standard deviation, even though it is much lower than in TetGen. Classes implementing the mesh and the algorithms still tend to have many lines of code making them complex and thus difficult to maintain. We have improved the design with respect to this metric but more work is still required.

The maximum percentage of comments has increased because we have added small size classes like *Criterion* with a high percentage of comments, but the average is almost the same because all comments included in TetGen were also copy-pasted into MeshingToolGenerator.

The maximum method cyclomatic complexity is higher than in TetGen because we did not change the method that causes it, but the average and standard deviation is lower than in TetGen due to the simplification of existing methods or the added new ones. The maximum class cyclomatic complexity is much higher than in TetGen because the class that contains the method with maximum cyclomatic complexity has fewer methods than before. Then the average of method complexities of that class is higher that in TetGen.

4.1.3. Consequences of the improved flexibility

The gains in structural metrics have direct consequences on software flexibility. TetGen is built for generating Delaunay meshes, and so is MeshingToolGenerator provided that it is intended to fulfill the same goals. These meshes tend to deal with nearly equilateral tetrahedra. However, there are some applications that require other kinds of meshes, such as anisotropic meshes [14]. The shape of the elements in these meshes tend to follow a prestablished direction, and they are not required to be nearly equilateral. MeshingToolGenerator, as it is, can manage anisotropic meshes just by relaxing the refinement criterion, e.g. $\texttt{RadiusEdgeRatio} > 2$. Nevertheless, anisotropic meshes cannot be generated on purpose because none of the criteria included in TetGen allow a direction to be provided as a parameter neither to generate the initial mesh nor to refine the existing mesh. If we want to count on this functionality, three new classes should be incorporated: `GenerateAnisotropicMesh` as a subclass of *GenerateVolumeMesh*, `RefineAnisotropicMesh` as a subclass of *Refine*, and `VerticalStretch` or `VerticalStretch` as a subclass of *Criterion*. These changes do not impact the existing code and the functionality, even though it may be complex, it is easily incorporated once developed.

Another interesting modification that may be included in MeshingToolGenerator is to either input or output the mesh data directly from or to another application. For example, it may be useful to visualize the mesh in a visualizer or read the mesh directly from another mesh generator such as TetMesh, without saving the mesh to a file. Currently, *InputOutput* is realized as a series of classes that implement both functions: transform data to or from the appropriate format, and either input or output data from or to a file. If we want the tool to be able to transfer data directly, this structure should be modified so that these two functionalities are decoupled: *InputOutput* will become *FormatInputOutput*, a hierarchy where each subclass transforms mesh data to or from the appropriate format, and another hierarchy *Transfer* for directly transferring data either to or from a file or to or from another application in the form of a pipeline through its subclasses `File` and `Pipeline` respectively. This second hierarchy is related to *FormatInputOutput* as an aggregation dependency. This solution follows the Bridge design pattern. Current code should be modified to incorporate this new structure. However, if we require communication with other applications whose format is not supported yet, either through a file or a pipeline, a new subclass of *FormatInputOutput* should be included without affecting the rest of the code.

4.2. Performance evaluation

As we already mentioned, one of the most important characteristics of meshing tools is performance. That is why we followed a formal performance analysis both of TetGen and of MeshingToolGenerator so as to get an objective comparison. We first considered performance as CPU response. The analysis performed consists of two different actions:

1. Generate a volume mesh starting from a geometry specified as a PLC (piecewise linear complex). This test was applied to eight different objects obtained from the TetGen repository [24].
2. Generate a good quality volume mesh. For this, a refinement is performed using the radius-edge criterion. This action is applied to two different geometries specified as PLC also obtained from the TetGen repository: Example.poly and PMCD.poly, requiring different quality values between 0.5 and 2.

We also analyzed the memory used by both implementations using the Valgrind [5] software. However, since MeshingToolGenerator was built just by decoupling the algorithms from the main class in TetMesh, the memory used in both applications was always exactly the same. So there is absolutely no difference in this dimension of performance.

4.2.1. Volume mesh generation

Table 4 shows the time it took to generate each of the eight different volumes specified as PLC.

The times it takes to generate the different volume meshes using either TetGen or MeshingToolGenerator are indistinguishable in all cases, independently of the object geometry complexity. With respect to this performance dimension we can conclude that decoupling TetGen into a MeshingToolGenerator had no impact.

4.2.2. Volume mesh refinement

Two volume meshes, Example and PMDC, were used as benchmarks for comparing the refinement time of both tools. Fig. 4 shows the original PMDC and two successive refinements.

Figs. 5 and 6 show the comparison of time consumed for refining the two volume meshes as a function of the expected quality of the obtained mesh defined using the radius-edge criterion.

In both cases the time it takes to refine the mesh is indistinguishable using either TetGen or MeshingToolGenerator. Moreover, with a ratio higher than 1.2, the performance is identical and quite fast, and most users would consider as acceptable any

Table 4
Performance evaluation for the generation of a volume mesh using TetGen and MeshingToolGenerator.

Figure	Initial vertices	Final vertices	Tetrahedra	Exec. time (TetGen)	Exec. time (MeshingToolGenerator)
Example	54	79	188	0.03	0.03
Tahol	386	396	1045	0.11	0.10
Socket	836	891	2493	0.26	0.26
Aparatous	1819	3113	8322	1.61	1.60
PMDC	972	3401	9222	1.73	1.67
CutSphere	3195	4584	13,640	2.67	2.68
Balls3astr	748	4092	13,792	1.50	1.50
Balls3astr-20	1510	5330	30,793	2.05	2.06
Brain	10,477	11,605	36,331	3.88	3.88

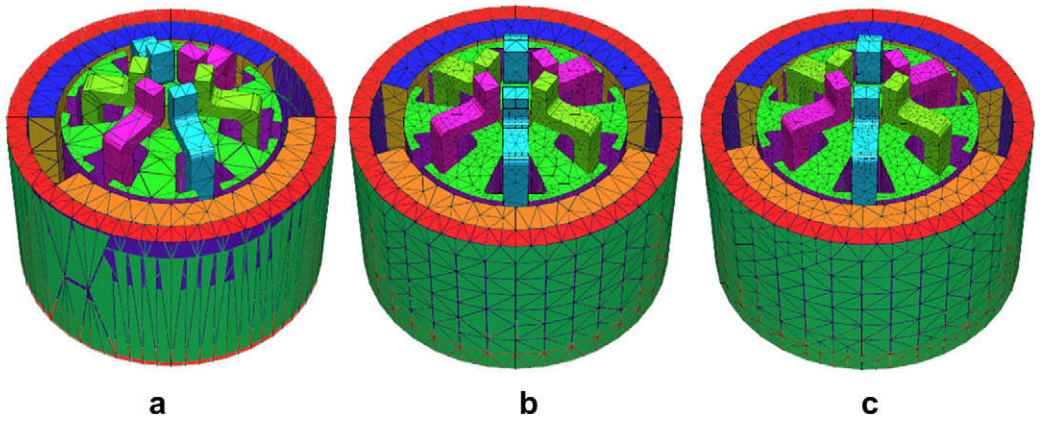

Fig. 4. Refinement of PMDC using the RadiusEdgeRatio criterion. (a) is the original volume Delaunay mesh, (b) the original mesh refined with a 1.2 ratio, and (c) the original mesh refined with a 1.5 ratio.

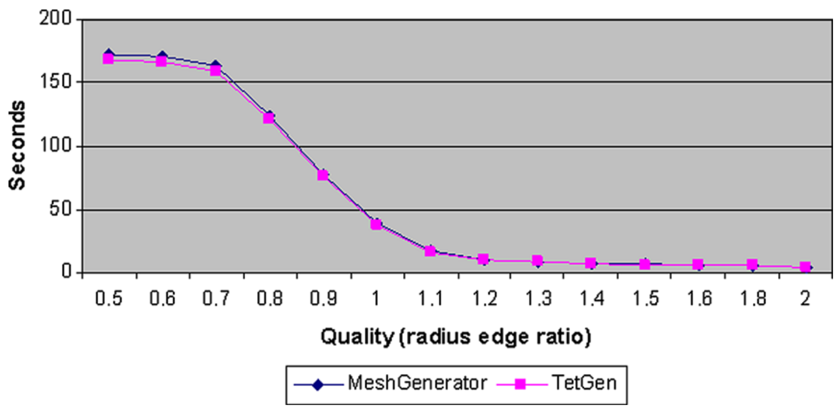

Fig. 5. Time for generating the PMCD quality mesh.

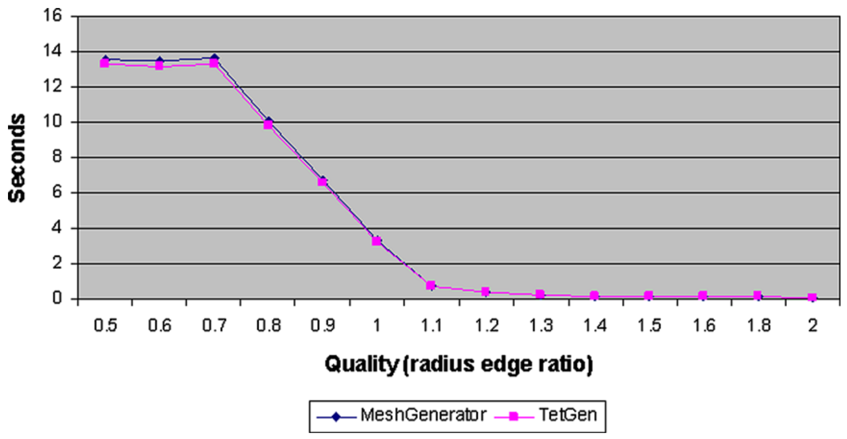

Fig. 6. Time for generating the example quality mesh.

mesh with a quality higher than 1.4. Again, we can conclude that the new structure of the tool does not affect this dimension of performance either.

5. Conclusions

We developed MeshingToolGenerator by refactoring the well known 3D meshing tool TetGen in order to improve its structure looking forward to achieve better flexibility. Doing so we wanted to be able to still have available all TetGen's functionality but also be able to extend it in the future by adding new input formats, algorithms implementations, and quality criteria, among others. We also wanted to preserve TetGen's performance qualities, or at least not sacrifice them severely.

MeshingToolGenerator's design structure is highly improved with respect to TetGen according to almost all OO design metrics applied, so the obtained software is definitively more flexible [16]. Performance both, according to response time and memory usage, is almost exactly the same in both implementations. So we obtained a tool that has the same functionality and performance, but now it has also good maintainability and flexibility properties, even though there is still some room for more improvement.

This new design flexibility also allows other designers to reuse all or just part of the tool in order to develop other meshing tools in other contexts. For example somebody could take the `RobustDelaunayTetGen` algorithm for generating the initial mesh, that is quite robust, and use it in another tool such as one that implements the Lepp algorithms for refining and improving the mesh [22].

We were able to decouple TetGen mainly by creating independent classes for algorithms, input and criterion definition. However, all definitions relating vertices, edges, faces, tetrahedra, and other geometric elements are still inside the `MeshTetGen` class. Decoupling these classes would probably improve most object-oriented metrics. Nevertheless, it is still to be determined how much this change would penalize the performance.

Acknowledgment

The work of Nancy Hitschfeld and Felipe Contreras was supported by Fondecyt No. 1061227.

References

[1] Bastarrica María Cecilia, Hitschfeld-Kahler Nancy. Designing a product family of meshing tools. Adv Eng Software 2006;37(1):1–10.
[2] Beall Mark W, guest editor. Eng Comput 1999;15(1).
[3] Chidamber Shyan R, Kemerer Chris F. A metrics suite for object-oriented design. IEEE Trans Software Eng 1994;20(6):476–93.
[4] Contreras Felipe. Adaptación de una Herramienta de Generación de Mallas Geométricas 3D a una nueva Arquitectura. Master's thesis, Departamento de Ciencias de la Computación, Universidad de Chile; 2007.
[5] Valgrind Developers. Valgrind user manual; June 2006. <http://valgrind.org/>.
[6] Eliëns Anton. Principles of object-oriented software development. 2nd ed. Addison-Wesley; 2000.
[7] ElSheikh Ahmed H, Spencer Smith W, Chidiac Samir E. Semi-formal design of reliable mesh generation systems. Adv Eng Software 2004;35(12):827–41.
[8] Fabri Andrea. CGAL – the computational geometry algorithm library. In: Proceedings of the 10th annual international meshing roundtable (CA), USA; October 7–10, 2001.
[9] Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John. Design patterns: element of reusable object oriented software. Addison-Wesley; 1995.
[10] George Paul-Louis, Hecht Frédéric, Saltel Éric. TetMesh-GHS3D V3.1, the fast, reliable, high quality tetrahedral mesh generator and optimiser; 1986. White paper. <http://www.simulog.fr/mesh/gener2.htm>.
[11] Hitschfeld Nancy, Lillo Carlos, Cáceres Ana, Bastarrica María Cecilia, Rivara Maria-Cecilia. Building a 3D meshing framework using good software engineering practices. In: IFIP workshop on advanced software engineering. IFIP, vol. 219. Santiago, Chile: Springer; 2006. p. 162–70.
[12] Johnson CR, Parker S, Weinstein D, Heffernan S. Component-based problem solving environments for large-scale scientific computing. Concurrency Computat Practice Exp 2002;14:1337–49.
[13] Kumfert G, Bernholdt DE, Epperly TGW, Kohl JA, McInnes LC, Parker S, et al. How the common component architecture advances computational science. J Phys Conf Ser 2006;46:479–93.
[14] Labelle Francois, Richard Shewchuk Jonathan. Anisotropic voronoi diagrams and guaranteed-quality anisotropic mesh generation. In: Proceedings of the nineteenth annual symposium on computational geometry. San Diego (CA): Association for Computing Machinery; June 2003. p. 191–200.
[15] Mackie RI. Using objects to handle complexity in finite element software. Eng Comput 1997;13(2):99–111.
[16] Meyer Bertrand. Object-oriented software construction. 2nd ed. Prentice Hall; 1997.
[17] Mitchell Scott A, Vavasis Stephen A. Quality mesh generation in three dimensions. In: Proceedings of the eighth annual symposium on computational geometry. Berlin, Germany: ACM; 1992. p. 212–21.
[18] Mobley Anton V, Tristano Joseph R, Hawkings Christopher M. An object-oriented design for mesh generation and operation algorithms. In: Proceedings of the 10th annual international meshing roundtable. Newport Beach (CA), USA; October 7–10, 2001.
[19] Owen Steve. Meshing software survey; 1998. <http://www.andrew.cmu.edu/user/sowen/softsurv.html>.
[20] Remacle Jean-Fancois, Shephard Mark S. An algorithm oriented mesh database. Int J Numer Methods Eng 2003;58:349–74.
[21] Rivara María Cecilia. New longest-edge algorithms for the refinement and/or improvement of unstructured triangulations. Int J Numer Methods Eng 1997;40:3313–24.
[22] Rivara María Cecilia, Hitschfeld Nancy, Bruce Simpson R. Terminal-edges delaunay (small-angle-based) algorithm for the quality triangulation problem. Computer-Aided Design 2001;33:263–77.
[23] Si Hang. On refinement of constrained delaunay tetrahedralizations. In: Proceedings of the 15th international meshing roundtable; September 2006. p. 509–28.
[24] Si Hang. TetGen: a quality tetrahedral mesh generator and three-dimensional delaunay triangulator; 2007. <http://tetgen.berlios.de>.

[25] Si Hang, Gärtner Klaus. Meshing piecewise linear complexes by constrained delaunay tetrahedralizations. In: Proceedings of the 14th international meshing roundtable; September 2005. p. 147–63.
[26] Smith Spencer. Systematic development of requirements documentation for general purpose scientific computing software. In: 14th IEEE international conference on requirements engineering (RE 2006). IEEE Computer Society; September 2006. p. 205–15.
[27] Smith Spencer, Chen Chien-Hsien. Commonality analysis for mesh generating systems. Technical report CAS-04-10-SS, Department of Computing and Software, McMaster University; October 2004.
[28] Smith Spencer, Chen Chien-Hsien. Commonality and requirements analysis for mesh generating software. In: Proceedings of the sixteenth international conference on software engineering & knowledge engineering (SEKE'2004); June 2004. p. 384–7.
[29] Smith Spencer, Lai Lei, Khédri Ridha. Requirements analysis for engineering computation: a systematic approach for improving reliability. Reliab Comput 2007;13(1):83–107.
[30] Weiss David M. Commonality analysis: a systematic process for defining families. In: Proceedings of the development and evolution of software architectures for product families, second international ESPRIT ARES workshop. Lecture notes in computer science, vol. 1429. Springer; February 1998. p. 214–22.