Designing a product family of meshing tools

María Cecilia Bastarrica*, Nancy Hitschfeld-Kahler

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

Abstract

Applying software engineering concepts can improve the quality of any software development, and this is even more dramatic for complex, large and sophisticated software, such as meshing tools. Software product families are series of related products that make intensive reuse of already developed components. Object-oriented design promotes reusability, so it is specially suited for designing the structure of product families. In this paper we present an object-oriented design of a product family of meshing tools, where all family members share the software structure. By instantiating the structure with particular algorithms and parameters, we can easily produce different tools of the family. A good family design allows us not only to combine existing algorithms but also to easily incorporate new ones, improving software family evolution. We show how the family design is used for the generation of finite element and finite volume meshing tools, as well as a new tool for image processing.

Keywords: Meshing tools; Object-oriented design; Software product family

1. Introduction

Software engineering has matured as a discipline in the last 30 years; it deals with techniques, methods and methodologies for developing good quality software. Developing complex software requires a software engineering approach, otherwise development usually gets out of control.

Software reuse is currently a trend in software development that promotes productivity and high quality [21]. Software already developed and used can be incorporated in new systems taking advantage of the savings in development time and costs, and also counting on the properties of the reused parts [7]. Complex systems could get the most out of software reuse: very sophisticated algorithms need to be developed only once, highly qualified people are paid for doing the work once and then it can be reused in several products, and debugging is mostly reduced to the task of integration tests because integrated components are already tested. Meshing tools are a very clear example of complex systems that could be benefited with the reuse-oriented development approach and it is shown in this paper. It is not usual in meshing tool development to have a systematic software reuse approach, even though ad hoc reuse is quite common. This practice allows developers to identify components that are common and potentially reusable in successive versions and also in different but related products.

The development of meshing technologies has become an intense theoretical and practical research area. The study of mesh generation issues, initially tackled by engineers, physicists, end-users in general and some mathematicians, has become also a field of interest for computational geometers, computer scientists and interdisciplinary teams both in academic and applied research centers.

In spite of its complexity, and perhaps due to its complexity, only in the last years the development of meshing software has been researched from the software engineering point of view mainly applying object-oriented design and programming. Some of the interesting published work include the development of a software environment for the numerical solution of partial differential equations (Diffpack) [6], the design of generic extensible geometry interfaces between CAD modelers and mesh generators [17,20,25,28], the design of object-oriented data structures and procedural classes for mesh generation [19],

* Corresponding author. Tel.: +56 267 843 65; fax: +56 268 955 31.

E-mail addresses: cecilia@dcc.uchile.cl (M.C. Bastarrica), nancy@dcc.uchile.cl (N. Hitschfeld-Kahler).

the computational geometry algorithm library CGAL [9], the definition of an optimal OO mesh representation that allows the programmer to build efficient algorithms (AOMD) [22], and algorithms that can be used independently of the concrete mesh representations [3], and a tool to support these algorithms (GrAL, Grid Algorithms Library) [4]. More recently, the use of formal methods for improving reliability of mesh generation software has also been addressed [8]; here, algebraic specifications are used in order to produce reliable components that could be used to build mesh generation software.

Several techniques for the refinement and improvement of meshes in two and three dimensions have been considered in the last 20 years. In particular, the use of two related mathematical concepts-the longest-edge propagation path of a triangle and its associated terminal-edge-, have allowed the development of algorithms for dealing with general aspects of the triangulation problem: triangulation refinement problem, triangulation improvement problem, and automatic quality triangulation problem [23, 24]. These mesh concepts have been later applied for the improvement of obtuse angles [13,14], as well as for the generation of approximate quality triangulations [26].

In this work we take advantage of our research experience in the field of meshing, which includes the development of algorithms and of several prototypes tested in an academic setting [13,14,26], as well as the development of an object-oriented mesh generator for semiconductor device simulation [12]. All these independent efforts are combined in the tool family we are presenting.

This paper describes the design of a family of object-oriented meshing tools for the generation of 2D triangulations based on the Lepp-concept. One of the most important requirements of this meshing tool family is extensibility on several aspects that can evolve: strategies for the generation of an initial mesh, new refinement and/or improvement algorithms based on the Lepp-concept, criteria for refinement or improvement of a mesh, and strategies for the generation of the final mesh. Several successful tools were built based on this design. The family design allows the user to generate new tools by selecting which strategy he/she wants to use for each mesh generation step. Software developers are able to add a new strategy, criteria or region shape by modifying only a few parts of the source code, if any at all.

A product family approach analyzing commonalities and variability of meshing tools has been used in [27]. Their main focus is the variability of the input and output formats, and the generation of the mesh as the whole. We take a complementary approach treating each step of the mesh generation and management processes as objects, and making it possible to interchange different algorithms for refining, improving and postprocessing the mesh. Other possible varying aspects are considered out of this paper's scope.

The paper is organized as follows. Section 2 presents the concepts of software qualities and software product families. Section 3 clearly states and explains the requirements for the meshing tool family. In Section 4, the complete development cycle of the tool family and its members is described: the methodology, analysis and design, and implementation. Section 5 presents two already published examples, now addressed as members of the presented tool family. Section 6 shows the development of a new tool. In Section 7, some of the results are presented, as well as a description of our ongoing work.

2. Product families and good software design

A good implementation of a software system is not only one that satisfies its functional requirements, but also satisfies its non functional requirements. Functionality is the definition of the output the system must produce, given a certain input. But if other qualities were not important, any monolithic built, slow performing and resource consuming system would do, and this is not the case.

A quality attribute such as performance is quite popular and recognized as important, i.e. efficient use of resources, mainly response time, but also memory, disk and bandwidth usage. Algorithms research has focused on improving this attribute since more than four decades. However, with new faster and cheaper computers, performance has been deemphasized lately in favor of other qualities.

More recently, maintainability has become the most important target with all its variants: modifiability, extensibility, portability, and interoperability. Successful software is always in need of changes: new functionality needs to be added, new platforms are required, or change in the algorithms implemented.

Software systems with an acceptable performance but with a design and an implementation that does not help evolution, e.g. with a very intricate implementation, are not able to adapt to changing requirements, and when this is the case, the complete system needs to be reconstructed.

Software reuse is also a highly desirable quality because of the high productivity and quality it brings. Modifying software can be seen as a form of reusing all the unchanged parts. If software is well structured, this reuse can be done more easily. Software classes or components are the most obvious items to be reused, but all the other artifacts built in the software development process can also be reused, e.g. test cases, user interfaces, user manuals, software architecture design, requirement specifications. We will use the term component to refer to the reusable units from which software systems are built.

Software families is a modern approach towards software development based on planned massive reuse. A product family is a set of products that are built from a collection of reused elements in a planned manner [7]. Opportunistic reuse does not usually work [5]; thus, reusable elements should be developed, tested, documented, classified and stored in such a way that reuse is promoted.

This development process is evidently longer and more expensive than developing one product at a time, but if they are reused enough times, it is still cost-effective. Experience has shown that the costs of developing reusable elements is paid off after the second or third product is built [30].

In this paper we take a software family approach for developing a family of meshing tools. In [27], the application of product family concepts in the development of mesh generation tools has already been applied.

3. A Product family of meshing tools

Our effort was originally oriented towards generating an extensible 2D triangulation tool [2,13,14]. However, the flexibility achieved in the design let us restate our goal as to build a tool family. A specification of a mesh generator family architecture can be found in [1].

The main steps of any mesh generation process can be summarized as follows:

- generation of an initial mesh that fits the domain geometry;
- generation of an intermediate mesh that satisfies the density requirements specified by the user;
- generation of an improved mesh that satisfies the quality criteria;
- generation of the final mesh.

These steps can be identified as some of the main commonalities among all members of the mesh generator tool family.

The algorithms for generating an initial mesh receive as input the geometry of the domain and generate the initial mesh as output. We are interested in both, initial triangulations that satisfy the Delaunay condition and initial triangulations that do not satisfy this condition. The initial mesh is the input of the refinement step that divides coarse triangles into smaller ones until the refinement criteria specified by the user are fulfilled in the indicated region. The refined mesh is the input to the improvement algorithm. The user specifies several regions with their respective improvement criteria. We are interested in refinement and improvement algorithms based on the Lepp-concept [24]. The refinement and improvement algorithms used at the moment are h-refinement. A proper final mesh might have additional requirements that are always applied to the whole mesh. For example, the elimination of boundary/interface obtuse triangles is applied if the finite volume method is used. This step can also be empty.

Our product family should easily evolve in the following aspects:

- strategies for the generation of initial meshes;
- strategies based on the Lepp and terminal-edge concepts;
- refinement and improvement criteria;
- strategies to generate proper final meshes;
- shape of the refinement and improvement regions.

These are the variations of the family members [27]. This separation of concerns is the basis for the design of the product family [16].

The incorporation of each new strategy, criterion or region shape to generate a new family member should normally not modify at all the source code, or should, in the worst case, have a minimal impact.

4. Tool implementation

4.1. Methodology

One of the most difficult problems in the development of a large object-oriented software system is the organization of the complex relationships that exist among objects in the application domain [11]. The object relationships can be inheritance, aggregation, association and use. Objects with a similar behavior are grouped into types and they are known as instances of their types. Subtyping allows the developer to build good type hierarchies. This is implemented in programming languages through the concepts of classes and inheritance. The idea is to model first the most general concepts in term of types, and then subtypes are used for concepts that are a specialization of a type. The most natural way to recognize subtypes are subsets. Some authors [11] recommend the use of inheritance only under the subtype relationship. Other authors also recommend the use of inheritance under generalization, limitation, variance and reuse [18]; this last approach is not followed in our work.

Our design follows these guidelines: (a) the use of types and inheritance for achieving software that is easy to maintain, extend and understand [11], (b) the design of good classes according to [18], and (c) the identification of design patterns previously used in other applications [10].

4.2. Analysis and design

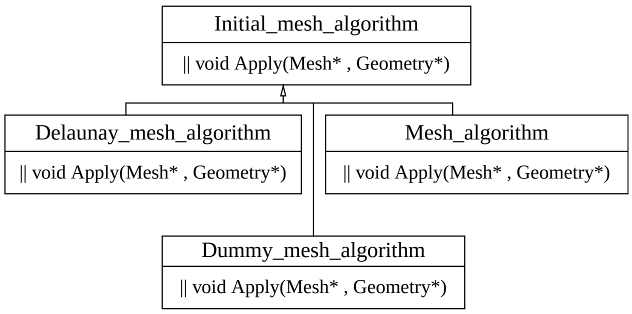
The mesh is modeled as a container object that holds the mesh information and it can be considered as a commonality among all family members. The `Mesh` class contains the information about the mesh at hand; in our application it is a 2D triangulation composed of vertices, edges and triangles. As part of its interface, the `Mesh` class provides methods to access its constituent elements, to load a mesh from a file, to store a mesh in a file, and some mesh validation methods. `Vertex, Edge` and `Triangle` are also modeled as classes; each of them providing their most concrete/ad hoc functionality and also providing access to neighborhood information within the mesh as part of their interface. For example, the `Vertex` class contains the point coordinates and provides access to its coordinates and the list of edges that share this point. The Edge class contains its endpoints and provides operations to get its endpoints, to compute the edge length, to insert a point between its endpoints and to get the triangles that share it. The Triangle class contains its vertices and edges, and provides operations to get its vertices and edges, to get the longest edge and to compute the minimum angle, among others.

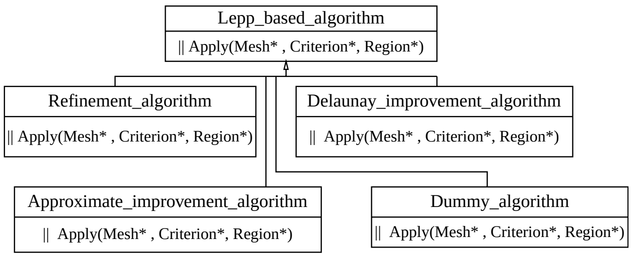
The key decision is then how to organize the complex processes associated to a mesh such as generating an initial triangulation, and the refinement and improvement strategies, among others. Should they be included as methods in the Mesh object or handled as separate types? Should the refinement/quality criteria be modeled as separate types or should they be a method of the Triangle object?

We have decided to handle the mesh generation steps as separate types following the strategy pattern [10] because (1) there are several ways to implement the same process and we want them to be interchangeable; (2) our software should be extensible in order to be able to generate new family members by incorporating new strategies and criteria and doing very few modifications to the source code.

We have identified four general algorithms: (a) Initial mesh algorithm, (b) Refinement algorithm, (c) Improvement algorithm, and (d) Final mesh algorithm. Algorithms (b) and (c) apply a criterion over a region of the domain. There are two different approaches towards improvement: Delaunay improvement and approximate improvement. In our case, the refinement and both types of improvement are subsets of what we have called Lepp-based algorithms. Thus, we have built an abstract class called Lepp_based_algorithm, whose subclasses are the refinement, improvement and approximate improvement algorithms. General algorithms (a) and (d) are also implemented as abstract classes and the different strategies for each of them are implemented as subclasses. Each general algorithm has a Dummy_algorithm as a subclass in order to be able to create objects that do nothing. These objects are created and used by default whenever the user wants to skip one of the mesh generation steps.

Fig. 1 shows the class diagram that represents different algorithms to generate an initial mesh: one subclass is to generate Delaunay meshes and the other to generate any triangulation. Since there are several algorithms to generate

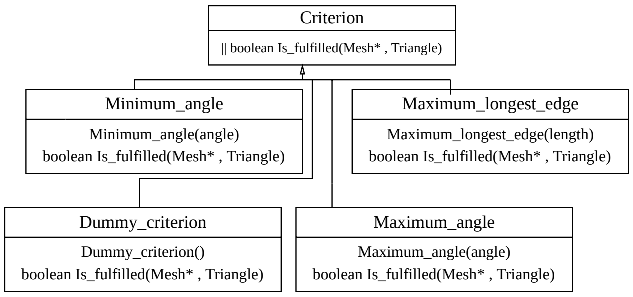

Initial_mesh_algorithm
|| void Apply(Mesh* , Geometry*)
Delaunay_mesh_algorithm
|| void Apply(Mesh* , Geometry*)
Mesh_algorithm
|| void Apply(Mesh* , Geometry*)
Dummy_mesh_algorithm
|| void Apply(Mesh* , Geometry*)
Fig. 1. Class diagram for the initial mesh algorithms.


Lepp_based_algorithm
|| Apply(Mesh* , Criterion*, Region*)
Refinement_algorithm
|| Apply(Mesh* , Criterion*, Region*)
Delaunay_improvement_algorithm
|| Apply(Mesh* , Criterion*, Region*)
Approximate_improvement_algorithm
|| Apply(Mesh* , Criterion*, Region*)
Dummy_algorithm
|| Apply(Mesh* , Criterion*, Region*)
Fig. 2. Class diagram for the Lepp-based algorithms.

Delaunay triangulations, each one should be added as a subclass of Delaunay_mesh_algorithm. In a similar way, there are several algorithms to generate any triangulation and each one should be a subclass of Mesh_algorithm. Each subclass must implement the virtual method Apply, which receives as input the geometry of the domain and an empty Mesh object, builds the corresponding initial triangulation and adds the vertices, edges and triangles to the mesh object. Notice that there is also a Dummy_mesh_algorithm that does not really generate an initial mesh, but it allows the user to read an already generated initial mesh; in this case the tool is used only for improving and/or refining this initial mesh.

Fig. 2 shows the class diagram that represents all the Lepp-based algorithms. Our application includes the Refinement_algorithm, the Delaunay_improvement_algorithm and the Approximate_improvement_algorithm. Each subclass must implement the virtual method Apply that receives a mesh, a criterion and a region as input, and refines or improves the mesh until the criterion is fulfilled in any triangle that intersects the region. The changes on the mesh are stored in the input Mesh object. Again, the Dummy_algorithm subclass is used when non of the other Lepp-based algorithms is required.

The refinement or improvement criterion applied to a mesh depends on the application problem. For example, for finite element meshes, normally meshes without very small angles are required. For finite volume meshes, small angles are not a problem, but large angles and vertices with a high number of edges converging to them must be avoided. We have decided to organize the refinement and improvement criteria in the same type hierarchy because they both depend on the user needs.


Criterion
|| boolean Is_fulfilled(Mesh* , Triangle)
Minimum_angle
Minimum_angle(angle)
boolean Is_fulfilled(Mesh* , Triangle)
Maximum_longest_edge
Maximum_longest_edge(length)
boolean Is_fulfilled(Mesh* , Triangle)
Dummy_criterion
Dummy_criterion()
boolean Is_fulfilled(Mesh* , Triangle)
Maximum_angle
Maximum_angle(angle)
boolean Is_fulfilled(Mesh* , Triangle)
Fig. 3. Class diagram for refinement and improvement criteria.

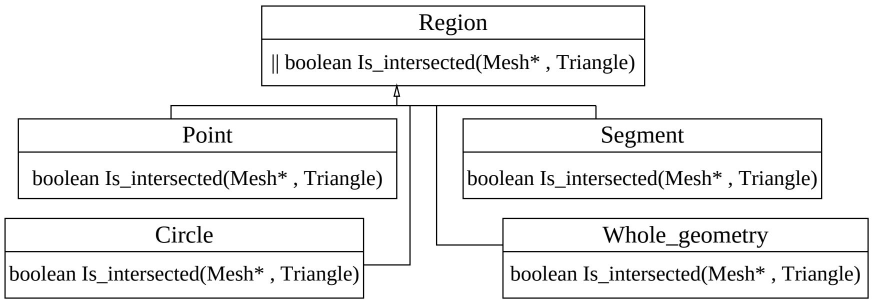

Region
|| boolean Is_intersected(Mesh* , Triangle)
Point
boolean Is_intersected(Mesh* , Triangle)
Segment
boolean Is_intersected(Mesh* , Triangle)
Circle
boolean Is_intersected(Mesh* , Triangle)
Whole_geometry
boolean Is_intersected(Mesh* , Triangle)

Fig. 4. Class diagram for regions.

Fig. 3 shows the class hierarchy diagram to model criteria. The virtual method of the `Criterion` abstract class is called `Is_fulfilled`. This method receives as input the mesh and the triangle that must be checked, it evaluates the mesh against the threshold value passed to the `Criterion` object when it was created, and returns either true or false. The class `Triangle` provides several methods to ask for triangle properties related to the evaluation of different criteria.

A criterion is only evaluated for a particular triangle if it is theoretically feasible and the intersection between the triangle and the `Region` is not empty. Fig. 4 shows the class hierarchy diagram to model the regions where the refinement or improvement algorithms are applied. The abstract class is called `Region` and its subclasses are `Point, Segment, Circle` and `Whole_geometry`. `Region` provides the virtual method `Is_intersected` which is redefined in each subclass. `Is_intersected` returns true if a triangle intersects the current region and false if it does not. If a region is defined as a point, it means that the triangle that contains this point or the triangles that share this point will be checked when the algorithm is applied. If the `Whole_geometry` region is chosen, all the mesh triangles will be checked; in this subclass, the method `Is_intersected` always returns true.

The `Final_mesh_algorithm` is not explicitly shown because it follows the same structure as the `Initial_Mesh_Algorithm`. One of the subclasses of `Final_mesh_algorithm` is the `Non_obtuse_boundary_algorithm`. Although the elimination of boundary obtuse angles could be seen as an improvement algorithm, it is not included as an improvement algorithm because it is a final step that is applied only over boundary triangles of the whole domain; additionally, it is not a Lepp-based algorithm. Any postprocess algorithm should be implemented as a descendant of the `Final_mesh_algorithm`.

The relationships among the abstract classes are shown in Fig. 5. We have not included the subclasses in order to keep the diagram simple and clear. There also exists a `Client` class that forms part of all tools and models the object that controls the order and the way user input requirements are executed by creating and coordinating the mesh generation steps, `Mesh, Criterion, Geometry` and `Region`. The `Geometry` object encapsulates the representation of the input mesh geometry.

4.3. Implementation

For the implementation of the `Mesh` class we have reused a C++library developed at the Integrated System Laboratories, ETH-Zurich [29]. This library manages a complete mesh representation [22] and provides more functionality than our application needs. However, reusing this implementation allowed us to use basic methods of classes `Triangle, Edge` and `Vertex` including methods for direct access to adjacency relationships, and the iterators over the triangles, edges, and vertices. The mesh geometry is obtained as an input of all family tools using the *dfise* format [15]; the `initial_mesh_algorithm` reads the geometry and stores it as a mesh object.

Fig. 6 shows the general structure of all members of the family of mesh generator. Each tool executes the same code but behaves differently according to the instantiation of each abstract class, following the idea of the template method software pattern [10]. After the `final_mesh_algorithm` is applied, the mesh can be stored into a file also in the *dfise* format, the same as the input.

Notice that in the pseudocode in Fig. 6, the addition of new strategies, region shapes and criteria does not modify the source code, because of the use of polymorphism and dynamic binding. Moreover, the addition of new criteria and region shapes does not modify the definition of the `Apply` method in the `Lepp_based_algorithm` class or its implementation in the subclasses.

Fig. 7 shows a possible instantiation of all abstract classes. In particular, the improvement algorithm is instantiated as the `Delaunay_improvement_algorithm` whose pseudocode is shown in Fig. 8.

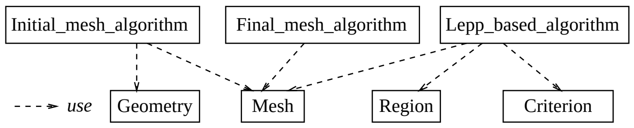


Fig. 5. Relationship among abstract classes.

```
initial_mesh_algorithm->Apply(mesh, geometry);
while(There is a pair (refinement_criterion, refinement_region){
    refinement_algorithm->Apply(mesh, refinement_criterion, refinement_region);
}
while(There is a pair (improvement_criterion, improvement_region){
    improvement_algorithm->Apply(mesh, improvement_criterion, improvement_region);
}
final_mesh_algorithm->Apply(mesh);
```

Fig. 6. General structure of all tools of the family.

```
Mesh* mesh = new Mesh();
Region* refinement_region = new Whole_geometry();
Region* improvement_region = new Whole_geometry();
Criterion* refinement_criterion = new Maximum_longest_edge(2);
Criterion* improvement_criterion = new Maximum_angle(120);
Initial_mesh_algorithm* initial_mesh_algorithm = new Delaunay_algorithm();
Lepp_based_algorithm* refinement_algorithm = new Refinement_algorithm();
Lepp_based_algorithm* improvement_algorithm = new Delaunay_improvement_algorithm();
Final_mesh_algorithm* final_mesh_algorithm = new Non_obtuse_boundary_algorithm();
```

Fig. 7. Object initialization for generating a quality non-obtuse boundary mesh.

```
void Delaunay_improvement_algorithm::Apply(Mesh *mesh,Criterion *criterion,Region *region){
List bad_elements;
Elements* elements = Mesh->Elements();
for(element_c = 0; element_c < elements->Number(); element_c++){
    if (region->Is_intersected(mesh, element_c) && !criterion->Is_fulfilled(mesh, element_c) )
        bad_elements.Add(element_c);
    }
while(bad_elements.Count()) {
    element_c = bad_elements[0];
    Improve(Mesh, element_c, criterion, bad_elements);
    }
}
```

Fig. 8. Apply method of the `Delaunay_improvement_algorithm` class.

The `Apply` method of the `Delaunay_improvement_algorithm` class shown in Fig. 8 builds first the `bad_element` list, which contains all the elements (triangles in our application) that intersect the region and that do not fulfill the criterion. Later, this method calls the `Improve` method for each bad triangle in the list. Since the improvement strategy based on the Lepp concept might improve a set of triangles at each time and each improved triangle may still not fulfill the criterion, we let the `Improve` method to update the `bad_element` list by deleting the improved triangles and adding the newly created triangles that still do not fulfill the criterion.

To illustrate how little work is to adding a new criterion, Fig. 9 shows the implementation of the constructor

```
Minimum_angle::Minimum_angle(float minimum_angle){
    threshold_value = minimum_angle;
}

boolean Minimum_angle::Is_fulfilled(Mesh* mesh, int element_index){
Elements* elements = mesh->Elements();
Triangle* triangle = elements[element_index];

if (triangle->Minimum_angle(mesh) < threshold_value) return false;
else return true;
}
```

Fig. 9. The constructor and the method `Is_fulfilled` of the `Minimum_angle` class.

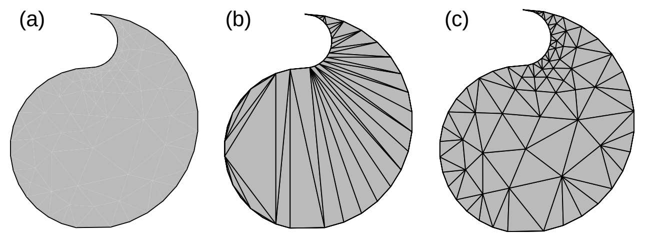


Fig. 10. Example 1: (a) Geometry, (b) Delaunay triangulation, and (c) Improved Delaunay triangulation using the minimum angle criterion, $\in = 20°$.

and the method `Is_fulfilled` of the `Minimum_angle` class. The constructor initializes the instance variable `threshold_value` with the minimum angle specified by the user. In order to implement the method `Is_fulfilled`, a method that computes the minimum angle of a triangle must either be available in the `Triangle` class or be added to it. The addition of a new criterion as a subclass of the `Criterion` class might require to add a new method to the `Triangle` class if the triangle property that the new criterion evaluates is not already available.

5. Application examples

In order to illustrate the meshing tools that can be generated as part of the family, Fig. 10(a) shows the geometry of a comma, Fig. 10(b) shows an initial triangulation that satisfies the Delaunay condition, and Fig. 10(c) an improved triangulation (using the Lepp-based algorithm) where each triangle has a minimum angle greater than or equal to 20°.

The algorithms used in this example tool are the `Delaunay_algorithm` for generating the initial mesh, the `Delaunay_improvement_algorithm` for the improvement of the mesh with the criterion `Minimum_angle`, applying the algorithms to the `Whole_geometry`. The instantiation of the tool in this example is shown in Fig. 11.

Fig. 12(a) shows a more complicated geometry with material interfaces. The tool in this case receives as input a Delaunay triangulation with 3342 points; Fig. 12(b) shows only part of it. In this case there are two different improvement criteria to be applied: a maximum angle $\gamma = 120°$ (the largest angle of each triangle on the mesh must be less than or equal to 120°) and a maximum edge-vertex connectivity $c = 10$ (the maximum number of edges converging to a vertex must be less than or equal to 10). The result of applying both criteria is shown in Fig. 12(c). Finally, the mesh passes through a post-process algorithm to eliminate obtuse angles opposite to boundary or interface edges (Fig. 12(d)).

The algorithms applied in this example are the `Dummy_algorithm` for the initial mesh provided that the initial Delaunay mesh is already generated. The `Delaunay_improvement_algorithm` is used with the `Maximum_angle` and the `Maximum_edge_vertex_connectivity` criteria applied over the `Whole_geometry` for improving it. Finally, the `Non_obtuse_boundary_algorithm` is used as a post process. The initialization is shown in Fig. 13.

6. Building a new member of the family

This section shows how to build a new tool, in particular, a tool to generate triangulations to represent images and extract some information about them: images, in this case, represent tree sections and the triangulations are intended to help identify and count tree rings. This new tool seems a priori different from the ones we have already described but it follows the same structural pattern. We did not need to add new meshing strategies for this application, we have just added new refinement criteria as subclasses of the `Criterion` class mentioned above. The constructor of each of the new criteria receives as parameters the name of the file that contains the image and a set of threshold values. The method `Is_fulfilled` analyzes the part of the image under the triangle and returns true in case the criterion is fulfilled or false otherwise.

Since we were interested in finding regions of an image with high intensity variation, we have implemented a criterion called `Intensity_variation`. In this case, the threshold parameters are the maximum intensity variation allowed in each triangle and the minimum size of the triangle. A target triangle is refined until the intensity variation associated to it is less than or equal to the specified maximum intensity variation or if the limit size of the triangle was reached. Then, the `Is_fulfilled` method gets the minimum and maximum allowed intensities of the pixels under the target triangle and returns true if the difference between them is less than or equal to

```
Mesh* mesh = new Mesh();
Region_shape region = new Whole_geometry();
Criterion* criterion = new Minimum_angle(20);
Initial_mesh_algorithm* = new Delaunay_algorithm();
Refinement_algorithm* = new Dummy_algorithm();
Improvement_algorithm* = new Delaunay_improvement_algorithm();
final_mesh_algorithm* = new Dummy_algorithm();
```

Fig. 11. Mesh and algorithm initialization for the comma example.

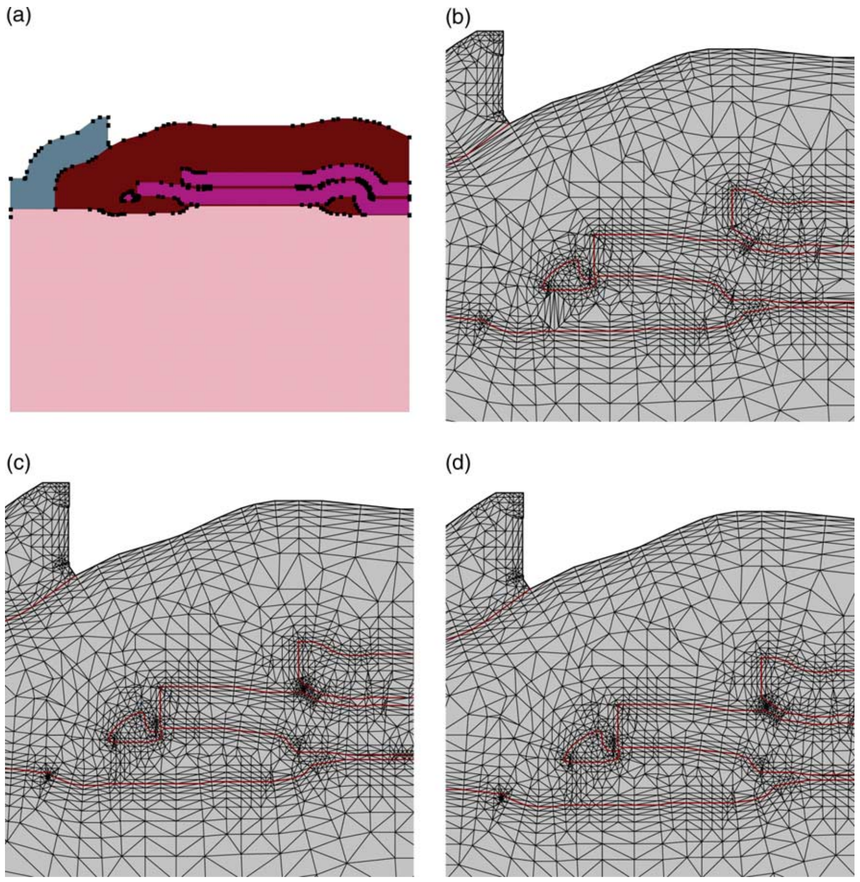


Fig. 12. Example 2: (a) Geometry, (b) Delaunay triangulation of a densified geometry, (c) Improved triangulation, and (d) Non obtuse boundary/interface triangulation.

```
Mesh* mesh = new Mesh();
Region_shape region = new Whole_geometry();
Criterion* criterion1 = new Maximum_angle(120);
Criterion* criterion2 = new Maximum_edge_vertex_connectivity(10);
Initial_mesh_algorithm* = new Dummy_algorithm();
Refinement_algorithm* = new Dummy_algorithm();
Improvement_algorithm* = new Delaunay_improvement_algorithm();
final_mesh_algorithm* = new Non_obtuse_boundary_algorithm();
```

Fig. 13. Mesh and algorithm initialization for the complex geometry example.

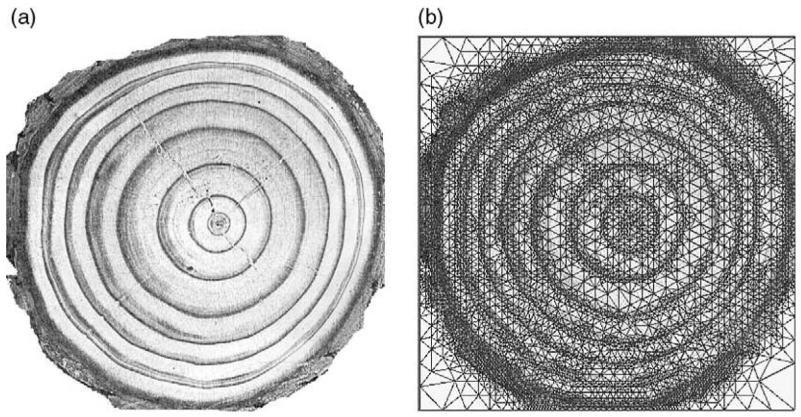


Fig. 14. (a) Image (b) the mesh over the image.

```
Mesh* mesh = new Mesh();
Region_shape region = new Whole_geometry();
Criterion* criterion1 = new Intensity_variation("rodela.pgm",128,4);
Initial_mesh_algorithm* = new Mesh_algorithm();
Refinement_algorithm* = new Refinement_algorithm();
Improvement_algorithm* = new Dummy_algorithm();
final_mesh_algorithm* = new Dummy_algorithm();
```

Fig. 15. Mesh and algorithm initialization for an image.

the maximum intensity variation and returns false if it is greater than this value or if the limit size of the triangle was reached.

In order to illustrate this application, Fig. 14(a) shows an image and Fig. 14(b) shows a corresponding mesh. The intensity in gray scale in this image varies between 0 and 256. The parameters for the refinement criterion are, 128 as the intensity variation threshold and 5 pixels as the minimum edge size for the triangle. The first mesh is generated inside a rectangle defined by the size of the image.

The algorithms applied in this example is first the `Mesh_algorithm` to generate a partition of the rectangle into two triangles and then the `Refinement_algorithm` for the refinement of the mesh according to the `Intensity_variation` criterion. The object initialization of this example is shown in Fig. 15.

7. Conclusions

The object-oriented design described in this paper and applied in the examples has allowed us to build an extensible and easily configurable family of meshing tools. Each new family tool is defined by instantiating a different combination of the existing classes for each processing step, as well as refinement and/or improvement criterion. It may be the case that a new algorithm or criterion should be created if it does not already exist. The rest of the code remains unchanged. The only exception might happen when adding a new refinement or improvement criterion, because its computation might require to add a new method in the Triangle class or any of its constituent elements for calculating the value associated with the new criterion metric.

We achieved the reusability and maintainability requirements without sacrificing performance dramatically. It is known that 2D meshing tools do not generally have an uncontrolled use of resources, neither memory nor processing time. Moreover, the use of an object-oriented language such as C++for the implementation provides a performance comparable to the one that can be achieved using standard C. In the family of meshing tools presented in Section 4, the shared architecture is the one shown in Fig. 5, and the reused components are the extensible class hierarchies of Figs. 1–4. Combining different instances of each class hierarchy we can obtain different particular members of the tool family. Combining the `Delaunay_mesh_algorithm`, `Refinement_algorithm` and `Delaunay_improvement_algorithm`, with the `Minimum_angle` criterion, we would get a finite element meshing tool. Instead, if we change the criterion for the `Maximum_angle` and add a postprocess `Non_obtuse_boundary_algorithm`, we have a finite volume method tool.

In the development of meshing tools, we take advantage of early definition of the architecture for setting object interfaces so that their implementation can be achieved in parallel. Thus, the product family approach can also enhance the development process.

7.1. Ongoing and future work

We have already developed criteria related with the geometry of the triangle like minimum angle and maximum edge length, and criteria that also require extra information, such as image properties. Similarly, we could also develop other criteria such as those related with the error in the numerical solution or related to the physical values of the domain.

The tool family design can be extended to include algorithms not based on the Lepp concept and also meshes not exclusively formed by triangles. Currently we are working in the implementation of 3D tools.

All these similar but different meshing tools form our tool family, and component reuse will be fundamental for the quality of the results and the productivity of the development process.

Acknowledgements

The second author thanks Prof. Wolfgang Fichtner for his hospitality at the ETH-Zurich during the development of part of this research. Also thanks to Luis Villablanca for explaining the use of the mesh representation library, to Bruce Simpson for lending us the geometry of example 1, and to Jens Krause for example 2. Finally, we would like to thank our anonymous reviewers for valuable comments on our manuscript. The work of Nancy Hitschfeld-Kahler was partially supported by Fondecyt Project No 1030672.

References

[1] Bastarrica MC. Base Architecture in a Software Product Line. In Proceedings of the XXVIII Latin American conference of informatics, CLEI'2002, page 119, Montevideo, Uruguay, November, 2002 [in Spanish].

[2] Bastarrica MC, Hitschfeld-Kahler N. An Evolvable Meshing Tool through a Flexible Object-Oriented Design. In Proceedings of the 13th international meshing roundtable, pages 203–212, Williamsburg, Virginia, September, 2004. Sandia National Laboratories.

[3] Guntram B. Generic software components for scientific computing. PhD Dissertation, TU Cottbus; 2000.

[4] Guntram B. A generic toolbox for the grid craftsman. Proceedings of the 17th GaMM-Seminar Leipzig on construction of grid generation algorithms. 2001. pp. 1–28.

[5] Bosch J. Design and use of software architectures. Adopting and evolving a product line approach. 1st ed. Wokingham, UK: Wesley; 2000.

[6] Bruaset A, Langtangen H. A Comprehensive set of tools for solving partial differential equations; Diffpack; 1996. citeseer.nj.nec.com/bruaset96comprehensive.html.

[7] Clements P, Northrop LM. Software product lines: Practices and patterns. 1st ed.: Wesley; 2001.

[8] ElSheikh AH, Smith SW, Chidiac Samir E. Semi-formal design of reliable mesh generation systems. Adv Eng Software 2004;35(12):827–41.

[9] Fabri A. CGAL—the computational geometry algorithm library. In Proceedings of the 10th annual international meshing roundtable, California, USA, October, 7–10; 2001.

[10] Gamma E, Helm R, Hohnson R, Vlissides H. Design patterns: Elements of reusable object oriented software. Wokingham, UK: Wesley; 1995.

[11] Halbert DC, O'Brien PD. Using types and inheritance in object-oriented programming. IEEE Software 1987;5(4):71–9.

[12] Hitschfeld N, Conti P, Fichtner W. Mixed element trees: A generalization of modified octrees for the generation of meshes for the simulation of complex 3-D semiconductor devices. IEEE Trans CAD/ICAS 1993;12:1714–25.

[13] Hitschfeld N, Rivara MC. Automatic construction of non-obtuse boundary and/or interface delaunay triangulations for control volume methods. Int J Numer Methods Eng 2002;55:803–16.

[14] Hitschfeld N, Villablanca L, Krause J, Rivara MC. Improving the quality of meshes for the simulation of semiconductor devices using Lepp-based algorithms. Int J Numer Methods Eng 2003;58:333–47.

[15] Integrated systems engineering AG. df-ISE, 6.0 edition; 1999. Zurich, Switzerland.

[16] Krueger CW. Using separation of concerns to simplify software product family engineering. In Dagstuht Seminar No. 01161, Dagstuhl Castle, Wadern, Germany, April, 2001.

[17] Merazzi S, Gerteisen E, Mezentsev A. A generic CAD-mesh interface. In Proceedings of the 9th Annual International Meshing Roundtable, pages 361–370. New Orleans, USA, October, 2–5; 2000.

[18] Meyer B. Object-oriented software construction. 2nd ed., Upper Saddle River, NJ: Prentice Hall; 1997.

[19] Anton V, Mobley JR, Hawkings TCM. An object-oriented design for mesh generation and operation algorithms. In Proceedings of the 10th annual international meshing roundtable. Newport Beach, California, USA, October 7–10; 2001.

[20] Panthaki M, Sahu R, Gerstle W. An object-oriented virtual geometry interface. In Proceedings of the 6th annual international meshing roundtable, pages 67–81. Park City, Utah, USA; 1997.

[21] Parnas D. On the design and development of program families. IEEE Trans Software Eng 1976;SE-2(1):1–9.

[22] Remacle J-F, Shephard MS. An algorithm oriented mesh database. Int J Numer Methods Eng 2003;58:349–74.

[23] Rivara MC. New longest-edge algorithms for the refinement and/or improvement of unstructured triangulations. Int J Numer Methods Eng 1997;40:3313–24.

[24] Rivara MC, Hitschfeld N, Simpson RB. Terminal-edges delaunay (small-angle-based) algorithm for the quality triangulation problem. Comput Aided Des 2001;33:263–77.

[25] Simpson RB. Isolating geometry in mesh programming. In Proceedings of the 8th international meshing roundtable, pages 45–54, South Lake Tahoe, California, October, 1999.

[26] Simpson RB, Hitschfeld N, Rivara MC. Approximate Shape Quality Mesh Generation. Eng Comput 2001;17:287–98.

[27] Smith S, Chen C-H. Commonality analysis for mesh generating systems. Technical Report CAS-04-10-SS, Department of Computing and Software, McMaster University, October 2004.

[28] Tautges TJ. The common geometry module (CGM): A generic, extensible, geometry interface. In Proceedings of the 9th annual international meshing roundtable, pages 337–347, New Orleans, USA, October 2–5; 2000.

[29] Villablanca L. Mesh generation algorithms for three-dimensional semiconductor process simulation. PhD thesis, ETH Zürich, Series in Microelectronics, Vol. 97, 2000. PhD thesis published by Hartung-Gorre Verlag, Konstanz, Germany.

[30] Weiss DM, Lai Chi Tau R. Software product-line engineering: A family based software development process. Wokingham, UK: Wesley; 1999.