AN EVOLVABLE MESHING TOOL THROUGH A FLEXIBLE OBJECT-ORIENTED DESIGN

María Cecilia Bastarrica   Nancy Hitschfeld-Kahler

Computer Science Department, Universidad de Chile
`cecilia@dcc.uchile.cl` `nancy@dcc.uchile.cl`

ABSTRACT

There are many diverse algorithms for generating a first mesh, refining it and improving it. A tool that allows us to interchange these algorithms according to the requirements of the problem at hand and also to incorporate new algorithms, needs to be flexible. This paper presents an object-oriented design approach for a meshing tool that provides these flexibility features by encapsulating processes as objects. It also shows how some already published application examples can be easily generated by combining these encapsulated processes. This achieved flexibility is a typical case of software component reuse, so all the knowledge of this area can be applied.

Keywords: meshing tools, object-oriented design, software engineering, software reuse

1. INTRODUCTION

Software engineering has grown to become a mature area in computer science over 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 a trend in software development that promotes productivity and high quality. 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 [1]. 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 this approach and it is shown in this paper. It is not usual in meshing tool development to have a software reuse approach. This would allow 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. For an introduction to these topics see [2, 3, 4] and the proceedings of the twelve annual International Meshing Roundtable conferences that, since 1992 gather together researchers and developers from industry, academia and US government laboratories.

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) [5], the design of generic extensible geometry interfaces between CAD modelers and mesh generators [6, 7, 8, 9], the design of object-oriented data structures and procedural classes for mesh generation [10] and the computational geometry algorithm library CGAL [11]. More recently, preliminary discussions on the use of formal methods for improving reliability of mesh generation software have also been published [12].

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 [13, 14]. These mesh concepts have been later applied for the improvement of obtuse angles [15, 16], as well as for the generation of approximate quality triangulation [17].

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 [15, 16, 17], as well as the development of an object-oriented mesh generator for semiconductor device simulation [18]. All these independent efforts are combined in the flexible tool we are presenting.

This paper describes the design of an object-oriented meshing tool for the generation of 2D triangulations based on the Lepp-concept. One of the most important requirements of this mesh generation tool is extensibility on all the aspects that can evolve over time: 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 applications were built based on this design. The system allows the user to select 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 very few parts of the source code.

Section 2 clearly states and explains the reusability and flexibility requirements for the meshing tool. In Section 3, the complete tool development cycle is described: the methodology, analysis and design, and implementation. Section 4 presents two already published examples, now addressed with the presented tool. In the 5 Section, some of the results based on software engineering concepts are presented, as well as a discussion and a description of our ongoing work.

2. EVOLUTION AND REUSE

2.1 Software Product Lines

Software reuse is becoming more important in software development because of the big gains both in productivity and quality. Reusing part of the software already developed for other projects makes the new project simpler because it will have fewer new parts developed, and because we can also count on the quality of already developed, tested and used software. 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.

Software product lines (SPL) is a modern approach towards software development based on planned massive reuse. All elements subject to reuse are called core assets of the SPL. So, an SPL is a set of products that are built from a collection of core assets in a planned manner and that satisfies the needs of a market segment [1]. Opportunistic reuse does not usually work [19]; thus, assets in a SPL 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 assets are reused enough times, it is still cost-effective. Experience has shown that the costs of developing reusable assets is paid off after the second or third product is built [20].

2.2 Flexibility Requirements

This section describes the main characteristics and requirements of our mesh generation tool. Our effort was oriented towards generating an extensible 2D triangulation tool, but the design here described can also be adapted and extended for the generation of mixed elements or quadrilateral meshes. In addition, the general tool design is completely independent of the space dimension (2D or 3D).

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

- generation of an initial mesh that fits the device 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.

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 step that divides coarse triangles into smaller ones until the criteria specified by the user are fulfilled. The refined mesh is the input to the improvement algorithm. The user specifies several regions with their respective refinement and/or improvement criteria. We are interested in refinement and improvement algorithms based on the Lepp-concept [13]. 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.

The main motivation of this work is to develop a scalable system that can 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.

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

3. TOOL IMPLEMENTATION

3.1 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 [21]. 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 [21] recommend the use of inheritance only under the subtype relationship. Other authors also recommend the use of inheritance under generalization, limitation, variance and reuse; 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 [21] (b) the design of good classes according to [22], and (c) the identification of design patterns previously used in other applications [23].

3.2 Analysis and Design

The core object of our system is the mesh. We have decided to model the mesh object as a container of the mesh information. The Mesh class should contain 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 should provide methods to access its contituent 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 vertex. 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 because (1) there are several ways to implement the same proccess and we want them to be interchangeable; (2) our software should be extensible to incorporate new strategies and criteria by doing very few modifications to the source code.

We have identified four processes: (a) Initial_mesh_algorithm, (b) Refinemen_algorithm, (c) Improvement_algorithm, and (d) Final_mesh_algorithm. Algorithms (b) and (c) apply a criterion over a region of the domain and in our case, both processes 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. Processes (a) and (d) are also implemented as abstract classes and the different strategies for each process are implemented as subclasses. Each process 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.

Figure 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 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 refining this initial mesh.

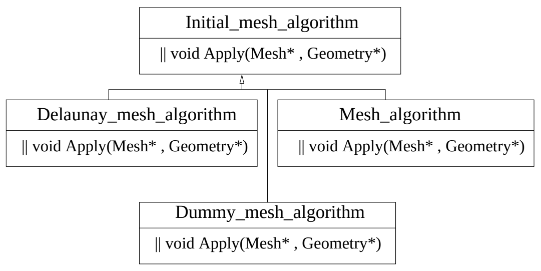
Initial_mesh_algorithm
|| void Apply(Mesh* , Geometry*)
Delaunay_mesh_algorithm
|| void Apply(Mesh* , Geometry*)
Mesh_algorithm
|| void Apply(Mesh* , Geometry*)
Dummy_mesh_algorithm
|| void Apply(Mesh* , Geometry*)

Figure 1: Class diagram for the initial mesh algorithms

Figure 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 are required.

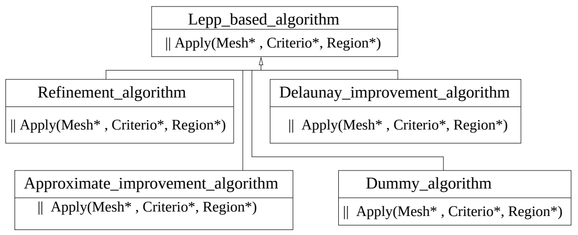
Lepp_based_algorithm
|| Apply(Mesh* , Criterio*, Region*)
Refinement_algorithm
|| Apply(Mesh* , Criterio*, Region*)
Delaunay_improvement_algorithm
|| Apply(Mesh* , Criterio*, Region*)
Approximate_improvement_algorithm
|| Apply(Mesh* , Criterio*, Region*)
Dummy_algorithm
|| Apply(Mesh* , Criterio*, Region*)

Figure 2: Class diagram for the Lepp-based algorithms

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.

Figure 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 user tolerance 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 if the intersection between the triangle and the region_shape is not empty.

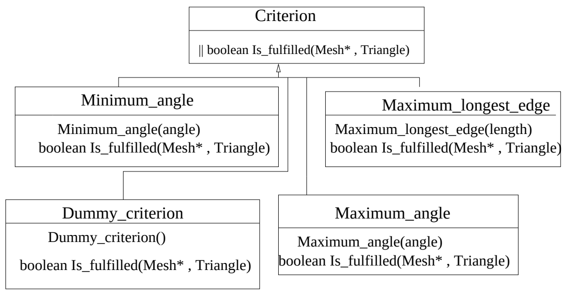
Criterion
|| boolean Is_fulfilled(Mesh* , Triangle)
Minimum_angle
Minimum_angle(angle)
boolean Is_fulfilled(Mesh* , Triangle)
Maximum_longest_edge
Maximum_longest_edge(length)
boolean Is_fulfilled(Mesh* , Triangle)
Dummy_criterion
Dummy_criterion()
boolean Is_fulfilled(Mesh* , Triangle)
Maximum_angle
Maximum_angle(angle)
boolean Is_fulfilled(Mesh* , Triangle)

Figure 3: Class diagram for refinement and improvement criteria

The Final_mesh_algorithm and Region_shape are also abstract classes. They are not explicitly shown because they follow the same structure of the previous ones. One of the subclasses of Final_mesh_algorithm is the Non_obtuse_boundary_algorithm. The subclasses of Region_shape are Point, Segment, Circle, Rectangle, Polygon and Whole_geometry. The last region shape is very useful to efficiently apply the criteria over the whole mesh: in this case, the method that checks if the target triangle intersects the region always returns true.

The relationships among the abstract classes can be observed in Figure 4. We have not included the subclasses in order to keep the diagram simple and clear. The Client is the class that models the object that controls the order and the way user input requirements are executed. The Client creates and coordinates the objects related to the mesh generation steps, Mesh, Criteria and Region_shapes.

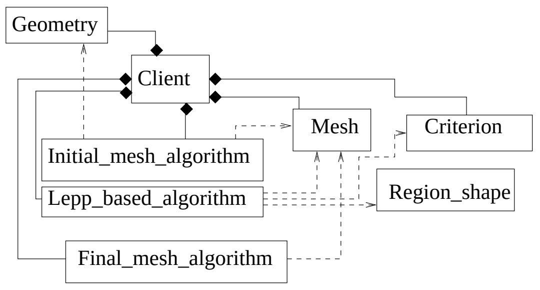

Figure 4: Relationships among the abstract classes

3.3 Implementation

For the implementation of the Mesh class we have reused and adapted a multidimensional C++ library developed at the Integrated System Laboratories, ETH-Zurich [24]. This library provides more functionality than our applicaction needs; however, reusing this implementation allowed us to use basic methods of classes Triangle, Edge and Vertex and the iterators over the triangles, edges, and vertices.

The tool's user interface allows the user to select an initial mesh algorithm, improvement and/or refinement criteria, region shapes and final mesh algorithms. An example of an initialization of main objects (in C++) that interact in our system is shown in Figure 5.

The Client object can execute each process separately or all processes together by using the sequence shown in Figure 6. Notice that this code is independent of the object that was created associated to a specific user requirement. The addition of new strategies, region shapes and criteria does not modify this source code, because of the use of polymorphism and dynamic binding.

By default, the Mesh and Criterion objects are empty and the objects that represent the processes are the respective dummy algorithms. This initialization looks as shown in Figure 7.

```
Mesh* mesh = new Mesh();
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();
```

Figure 5: Object initialization for generating a quality non-obtuse boundary mesh

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

Figure 6: General process

4. APPLICATION EXAMPLES

In order to illustrate the kind of meshes generated in our system, Figure 8 shows the geometry of a comma, Figure 9 shows an initial triangulation that satisfies the Delaunay condition, and Figure 10 an improved triangulation (using the Lepp-based algorithm) where each triangle has a minimum angle greater than or equal to 20°.

The algorithms applied in this example 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 initialization of this example is shown in Figure 11.

Figure 12 shows a more complicated geometry with interfaces. The input of our system was a Delaunay triangulation with 3342 points shown in Figure 13. This mesh was improved according to the following criteria: 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 ver-

```
Mesh* mesh = new Mesh();
Criterion* criterion = new Dummy_criterion();
Region_shape region = new Whole_geometry();
Initial_mesh_algorithm* = new Dummy_algorithm();
Refinement_algorithm* = new Dummy_algorithm();
Improvement_algorithm* = new Dummy_algorithm();
final_mesh_algorithm* = new Dummy_algorithm();
```

Figure 7: Mesh and algorithm default initialization

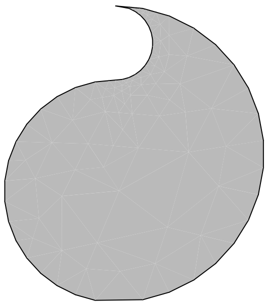

Figure 8: Geometry of example 1

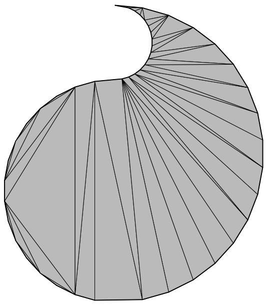

Figure 9: Delaunay triangulation of example 1

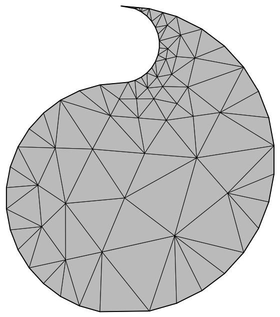

Figure 10: Improved Delaunay triangulation of example 1 using the minimum angle criterion, $\epsilon = 20^{\circ}$

```
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();
```

Figure 11: Mesh and algorithm initialization for the comma example

tex must be less than or equal to 10) and it is shown in Figure 14. Finally, the mesh was passed through a post-process algorithm to eliminate obtuse angles opposite to boundary or interface edges (Figure 15).

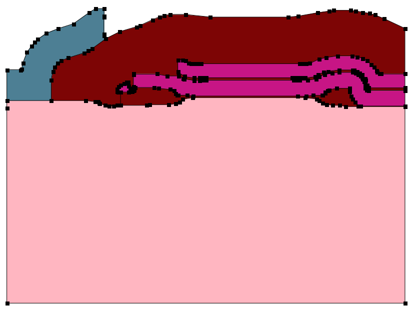

Figure 12: Geometry of example 2

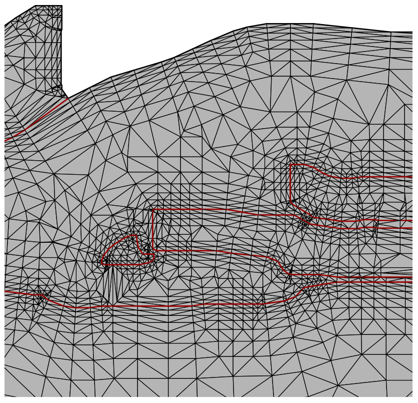

Figure 13: Delaunay triangulation of a densified example 2

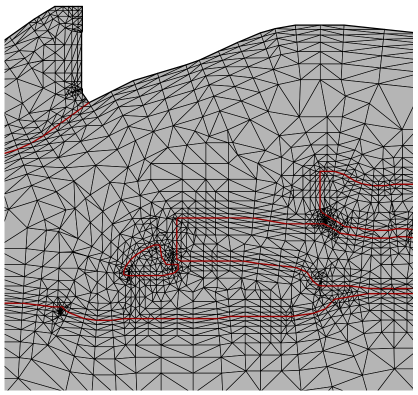

Figure 14: Improved triangulation

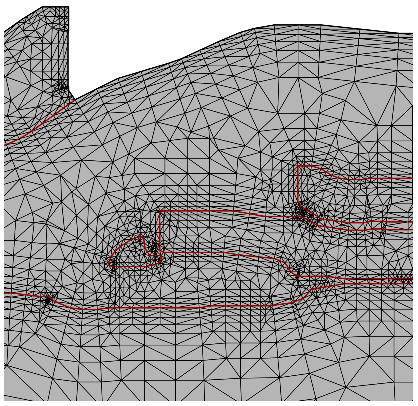

Figure 15: Non obtuse boundary/interface mesh of example 2

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 was used as a post process. This initialization is shown in Figure 16.

```
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();
```

Figure 16: Mesh and algorithm initialization for the complex geometry example

5. CONCLUSIONS

5.1 Results

The object-oriented design described in this paper and applied in the examples has allowed us to build an extensible and easily configurable meshing tool. Each new strategy, refinement/improvement criterion, or region shape, can be incorporated by (a) creating a new subclass that implements the new characteristic or strategy, (b) adding a new menu item in the user interface to select it (if this strategy must be visible to the user) and (c) creating the associated object when the user selects the respective item. The rest of the code remains unchanged. The unique 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 for calculating the value associated with the new criterion metric.

The meshing tool presented in Section 3 can be seen as a software product line, where the shared architecture is the one shown in Figure 4, and the reused components are the extensible class hierarchies of Figures 1, 2 and 3. Combining different instances of each class hierarchy we can obtain different particular tools. Combining the Delaunay_mesh_algorithm, Refinement_algorithm and Delaunay_improvement_algorithm, with the Minimun_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.

5.2 Discussion

Object-orientation has raised the level of abstraction for developing complex software systems. Encapsulation and inheritance have been extensively used for meshing tools design. However, the possibility of building even more complex software brought newer problems: object-orientation by itself is not enough to deal with very high complexity software. Productivity and quality have become also very important for meshing tool development as a commercial activity.

Object-orientation has promised to allow systems to be more easily maintained and flexible for evolution. However, software is not flexible independently of the change we want to perform; software can be more or less flexible for some particular changes. The SPL approach allows us to think about possible products that will be needed in the future and develop the whole SPL accordingly. All products in the SPL share the software architecture and the properties it enforces, and already implemented software components can be reused according to this architecture.

Encapsulation and polymorphism, basic concepts of object orientation, are used for promoting interchangeability, i.e. an implementation can be changed by another whenever it satisfies its interface. This feature allows us to extend our current tool framework to be able to deal with 3D meshes only by changing the internal structure of the mesh object and providing the appropriate algorithms for processing this 3D mesh, but still preserving the general tool architecture.

According to our experience, the SPL approach fits well with the planning and development of complex meshing tools. On the other hand, the high level of abstraction required has helped us to identify critical issues, and to appropriately deal with them. 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 SPL approach can also enhance the development process.

5.3 Ongoing Work

Currently, our mesh tool only contains criteria related to the geometry of the triangles that are useful for finite or control volume methods, but in the near future, we are going to add criteria related to other applications, in particular for the generation of triangulations to represent images. Note that we do not need to add new meshing strategies for this application, we just need to add a new refinement criterion that contains the image. The part of the image under the target triangle will be used to decide if the triangle will be refined or not. In a similar way, criteria related with the error in the numerical solution or related to the physical values of the domain can also be added. We have implemented 2D tools and at present we are working in the implementation of 3D tools. Parallel implementation and the use of data base technologies will be also considered in the future. All these similar but different meshing tools form our SPL, and component reuse will be fundamental for the quality of the results and the productivity of the development process.

ACKNOWLEDGEMENTS

We would like to thank Bruce Simpson (U. Waterloo) for lending us the geometry of example 1 and to Jens Krause (ISE-AG, ETH-Zurich) for example 2. The work of Nancy Hitschfeld was partially supported by Fondecyt Project N° 1030672.

References

[1] Clements P., Northrop L.M. *Software Product Lines: Practices and Patterns.* Addison Wesley, first edn., August 2001

[2] Babuska I., Zienkiewicz O.C., Gago J., de A. Oliveira E.R. *Accuracy estimates and adaptive refinements in finite element computations.* John Wiley-Sons, 1986

[3] Bern M., Eppstein D. *Mesh Generation and Optimal Triangulation.* Palo Alto Research Center. Xerox, March 1992

[4] Rivara M.C. "Design and Data Structure of Fully Adaptive Multigrid, Finite-Element Software." *ACM Transactions on Mathematical Software*, vol. 10, no. 3, 242–264, September 1984

[5] Bruaset A., Langtangen H. "A comprehensive set of tools for solving partial differential equations; Diffpack.", 1996. `citeseer.nj.nec.com-/bruaset96comprehensive.html`

[6] Merazzi S., Gerteisen E., Mezentsev A. "A generic CAD-Mesh interface." *Proceedings of the 9th Annual International Meshing Roundtable*, pp. 361–370. New Orleans, U.S.A., October 2-5,2000

[7] Panthaki M., Sahu R., Gerstle W. "An object oriented virtual geometry interface." *Proceedings of the 6th Annual International Meshing Roundtable*, pp. 67–81. Park City, U.S.A., 1997

[8] Simpson R.B. "Isolating geometry in mesh generation." *Proc. of the 8th Int. Meshing Roundtable*, pp. 45–54. October 1999

[9] Tautges T.J. "The common geometry module (CGM): A generic, extensible, geometry interface." *Proceedings of the 9th Annual International Meshing Roundtable*, pp. 337–347. New Orleans, U.S.A., October 2-5,2000

[10] Mobley A.V., Tristano J.R., Hawkings C.M. "An object oriented design for mesh generation and operation algorithms." *Proceedings of the 10th Annual International Meshing Roundtable.* California, U.S.A., October 7-10, 2001

[11] Fabri A. "CGAL- The computational Geometry algorithm library." *Proceedings of the 10th Annual International Meshing Roundtable.* California, U.S.A., October 7-10, 2001

[12] ElSheikh A.H., Smith S., Chidiac S.E. "Reliability of Mesh Generation Software." *Proceedings of the 7th United States Congress on Computational Mechanics.* U.S. Association for Computational Mechanics, Albuquerque, New Mexico, USA, 2003

[13] Rivara M.C., Hitschfeld N., Simpson R.B. "Terminal edges Delaunay (Small Angle Based) Algorithm for the Quality Triangulation Problem." *Computer-Aided Design*, vol. 33, 263–277, 2001

[14] Rivara M.C. "New Longest-Edge Algorithms for the Refinement and/or Improvement of Unstructured Triangulations." *International Journal for Numerical Methods in Engineering*, vol. 40, 3313–3324, 1997

[15] Hitschfeld N., Rivara M.C. "Automatic Construction of Non-obtuse Boundary and/or Interface Delaunay Triangulations for Control Volume Methods." *International Journal for Numerical Methods in Engineering*, vol. 55, 803–816, 2002

[16] Hitschfeld N., Villablanca L., Krause J., Rivara M.C. "Improving the Quality of meshes for the simulation of semiconductor devices usin Lepp-based algorithms." *International Journal for Numerical Methods in Engineering*, vol. 58, 333–347, 2003

[17] Simpson R.B., Hitschfeld N., Rivara M.C. "Approximate Quality Mesh Generation." *Engineering with Computers*, vol. 17, 287–298, 2001

[18] Hitschfeld N., Conti P., Fichtner W. "Mixed Elements Trees: A Generalization of Modified Octrees for the Generation of Meshes for the Simulation of Complex 3-D Semiconductor Devices." *IEEE Transactions on CAD/ICAS*, vol. 12, 1714–1725, November 1993

[19] Bosch J. *Design and Use of Software Architectures. Adopting and Evolving a Product Line Approach.* Addison Wesley, first edn., May 2000

[20] Weiss D.M., Lai C.T.R. *Software Product-Line Engineering: A Family Based Software Development Process.* Addison-Wesley Pub Co, August 1999

[21] Halbert D.C., O'Brien P.D. "Using Types and Inheritance in Object-Oriented Programming." *IEEE Software*, vol. 5, no. 4, 71–79, September 1987

[22] Meyer B. *Object-Oriented Software Construction.* Prentice Hall, second edn., 1997

[23] Gamma E., Helm R., Hohnson R., Vlissides H. *Design Patterns: Elements of Reusable Object Oriented Software.* Addison-Wesley, 1995

[24] Villablanca L. *Mesh Generation Algorithms for Three-Dimensional Semiconductor Process Simulation.* Ph.D. thesis, ETH Zürich, Series in Microelectronics, Vol. 97, 2000. PhD thesis published by Hartung-Gorre Verlag, Konstanz, Germany