Olena: a component-based platform for image processing, mixing generic, generative and oo programming

Thierry Géraud
École pour l'Informatique et les Techniques Avancées
82 PUBLICATIONS **833** CITATIONS

In the Proceedings of the 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE 2000), Young Researchers Workshop (published in ”Net.ObjectDays2000”; ISBN 3-89683-932-2), pages 653-659, Erfurt, Germany, October 2000.

Olena: a Component-Based Platform for Image Processing, mixing Generic, Generative and OO Programming

Alexandre Duret-Lutz (supervisor: Thierry Géraud)
EPITA Research and Development Laboratory
14-16 rue Voltaire, F-94276 Le Kremlin-Bicêtre cedex, France
{Alexandre.Duret-Lutz,Thierry.Geraud}@lrde.epita.fr

Keywords: generic programming, lazy compilation, bridge static-dynamic.

1 Introduction

This paper presents Olena, a toolkit for programming and designing image processing chains in which each processing is a component. But since there exist many image types (different structures such as 2D images, 3D images or graphs, as well as different value types) the platform has been designed with genericity and reusability in mind: each component is written as a generic C++ procedure, *la* STL.

Other libraries, such as Khoros [Kon94] have a different approach where a processing component contains an implementation for each type supported by the library. This makes code maintenance hard and prevents easy addition of new image types.

Still, Olena is not only a generic component library [Jaz95], it shall contain additional tools such as a visual programming environment (VPE). Those tools may be programmed in a classical object-oriented fashion (using operation and inclusion polymorphism) which may seems antagonist with the generic programming paradigm used in the library.

Section 2 outlines the architecture of Olena and elaborates more on the design problems resulting from the use of generic components. Section 3 presents the solution chosen to address these problems.

2 Overview of the platform architecture

Figure 1 shows the various parts of the architecture. A text-mode shell interface is needed to call the components and to prototype a processing, either interactively or via a script. Alternatively, a visual programming environment can be used to define the data flow diagrams for a processing chain. In each of these environment, components

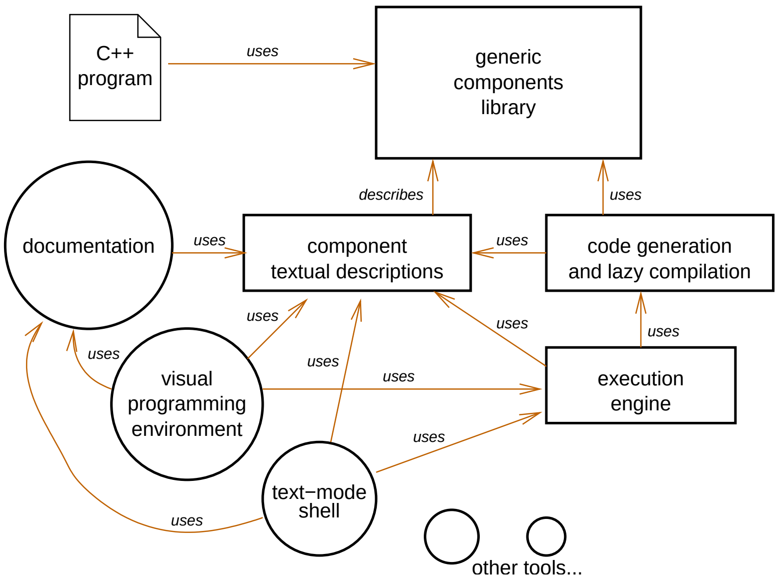

Figure 1: The Olena architecture.

can be grouped to form reusable composites components (such as the one shown in figure 2); and once a processing has been successfully prototyped, the corresponding C++ code can optionally be generated.

Moreover, for each component the representation in the environment (visual aspect in the VPE, option switches in the shell) as well as on-line help shall be generated automatically. To this end additional information is recorded by a database of textual description of each components.

The kernel of the platform is a generic library which provides generic components and data. A component is a processing, and is written as a C++ generic procedure. Data are the entities that transit between components in the processing chain; the library provides image classes, neighborhoods, value types, histograms, etc. This library can be used directly by the end user in a C++ program, independently of the other tools of the platform. A component that adds a constant value to an image can be written as follows.

```
template< class Aggregate_Model >
void add (Aggregate_Model& input, typename Aggregate_Model::data_type value)
{
  typename Aggregate_Model::iterator_type iter = input.create_iterator ();

  for (iter.first (); !iter.is_done (); iter.next ())
    iter.current_item () += value;
}
```

The library is written fully in generic programming in order to achieve good performances, using techniques such a those described in [Vel99]. The iterator pattern used in the sample code is an adaptation to generic programming of the classical OO design

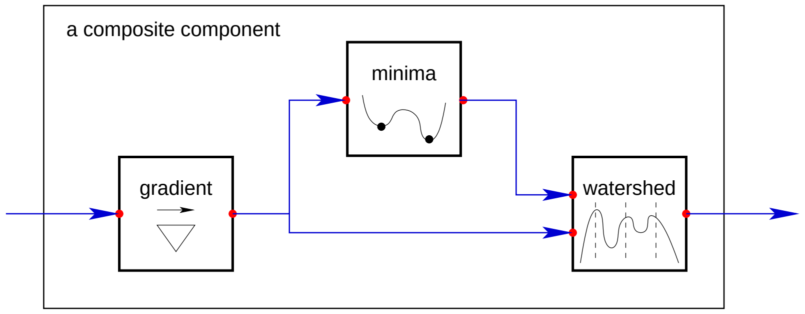

Figure 2: An image processing chain example ; here, a composite component made of three atomic components.

pattern, and is well know in generic programming; but other design patterns can also be adapted [Ger00a].

An issue with generic components is that they lead to many instances, i.e., several variants of binary code will be generated.. As long as it is used directly in a C++ program this is not a problem because the compiler will determinate and compile only the needed variants. But in interpreted environments such as the VPE, needed instances are not known until run-time. Moreover the number of available image types prevents the compilation of all variants once and for all (anyway since we want to allow user types, this would not be satisfying). We detail our solution in the next section. Type constraints on the argument types are also described for each component, because the interpreted environments will perform type checking at run-time.

3 Technical Solutions environment

As we said, at the VPE (or any other use interface) level only abstract data are handled. The only requirement is that in order to perform type checking, we need a `type()` method which returns the real type of the data as a string.

The abstract type used is not part of the generic library since it makes sense only for the VPE. Figure 3 shows how a set of concrete data types can be grouped in a hierarchy. We encapsulate each data of type `T` in a class `DataProxy<T>` which inherits from the superclass `AbstractData`. This an application of the *external polymorphism pattern* [Cle96].

Thus, the VPE can handle abstract data, but in order to apply the algorithm, the data should be downcast back to its concrete type. If the set of types is short and fixed, this can be done using a `switch` statement. Unfortunately neither conditions are true: there exists a lot of types and the users can add new ones. Our solution is to use lazy compilation.

Data flow is managed by an Execution Engine Figure 2. When all the inputs of a component are ready (i.e. the upstream components have completed their work) the processing can start. But before it begins, several steps must be performed:

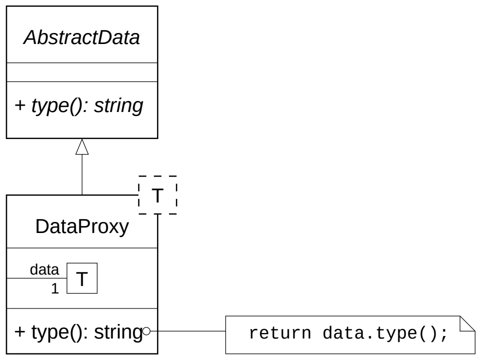

Figure 3: The `AbstractData` and `DataProxy` classes.

- type checking must be done on the inputs, this is done easily using the string-type of data and the rules given by the textual description of the component.
- input data must be casted to their real types (which are known as strings, at run-time).
- the generic algorithm must be instantiated for these types.

The last two steps cannot be done at run-time. Therefore we generate, with help from the textual description, the code for an intermediate procedure which performs these two tasks; this code is compiled as a dynamic library, it is then loaded by the Execution Engine and finally executed.

As an exemple, consider the “add” component. “add” takes two inputs - let’s label them `input1` and `input2` - and produce one output - named `output`. If this component receive a `Image2D<Float>` object and a `Float` value as inputs, the following code will be generated:

```
void add__Image2D_Float_ (Component& cmp)
{
  /* downcast the data */
  Image2D<Float> input1& =
    static_cast< DataProxy< Image2D< Float > >& >(cmp.entry ("input1")).data;
  Float input2& =
    static_cast< DataProxy< Float>& >            (cmp.entry ("input2")).data;
  Image2D<Float> output& =
    static_cast< DataProxy< Image2D< Float > >& >(cmp.entry ("output")).data;

  /* apply the algorithm */
  add  (input1, input2, output);
}
```

This procedure takes the OO version of the component as arguments and extracts its inputs and output with their concrete types in order to call the right variant of the “add” component. Finally, compiled variants of algorithms are cached, and therefore successive calls to these algorithms are not penalized by the slow compilation process. Moreover caching policies (such as component sharing by a group of users) can be applied.

4 Conclusion

The design presented here combine three advantages of components oriented designs (flexibility, reusability) of both dynamic approach (ability to assemble component interactively) and static approach (good performance) thanks to the bridge between static and dynamic worlds we presented. Still, Olena is a work in progress. So far, our work has been focused mainly on the generic library [Ger00a] but the solution presented in the previous paragraph have been tested in practice.

References

[Cle96] Chris Cleeland, Douglas C. Schmidt, and Timothy H. Harrison. *External polymorphism*. In Proceedings of the 3rd Pattern Languages of Programming Conference, Allerton Park, Illinois, September 4-6, 1996. `http://www.cs.wustl.edu/~cleeland/papers/`.

[Ger00a] Thierry Géraud, Yoann Fabre, Alexandre Duret-Lutz, Dimitri Papadopoulos-Orfanos, and Jean-François Mangin. *Obtaining genericity for image processing and pattern recognition algorithms*. In Proceedings of the 15th International Conference on Pattern Recognition (ICPR'2000), Barcelona, Spain, September 2000. To appear.

[Ger00b] Thierry Géraud and Alexandre Duret-Lutz. *Generic programming redesign of patterns*. In Proceedings of the 5th European Conference on Pattern Languages of Programs (EuroPLoP'2000), Irsee, Germany, July 2000. `http://www.coldewey.com/europlop2000/`.

[Jaz95] Mehdi Jazayeri. *Component programming - a fresh look at software components*. In Procedings of the 5th European Software Engineering Conference (ESEC'95), 457–478, September 1995.

[Kon94] K. Konstantinides and J.R. Rasure. *The Khoros software development environment for image and signal processing*. IEEE Transactions on Image Processing, vol. 3, no. 3, 1994, 243–252.

[Vel99] Todd L. Veldhuizen. *Techniques for scientific C++*. August 1999. `http://extreme.indiana.edu/~tveldhui/papers/techniques/`