

Modularity and Reusability of Algorithms – A Case Study using Caesar

Karl Matthias Hamel
Darmstadt University of Technology
Hochschulstr. 10
64289 Darmstadt, Germany
hamel@st.informatik.tu-darmstadt.de

Klaus Ostermann
Darmstadt University of Technology
Hochschulstr. 10
64289 Darmstadt, Germany
ostermann@informatik.tu-darmstadt.de

ABSTRACT

This paper investigates the usage of aspect-oriented techniques to increase the modularity and reusability of algorithms. We present a set of criteria to evaluate the modularity and reusability of algorithm implementations and argue that current algorithm libraries have serious limitations with respect to these criteria. We have implemented a small algorithm library using the aspect-oriented language Caesar in order to show that this solution provides higher modularity and reusability than conventional solutions and that it meets the previously defined criteria.

Categories and Subject Descriptors

D.2.13 [Software Engineering]: Reusable Software – *Reuse Models*.

General Terms

Algorithms, Measurement, Design, Languages.

Keywords

Aspect Orientation, Caesar, Reusability, Modularity, Algorithms.

1. INTRODUCTION AND IDEA OF THE WORK

Algorithms are complex and specific sets of code. Their development, adjustment and testing often requires high efforts, though they do not require many lines of code. Their correctness is also essential for the quality of a system. Therefore, algorithms can be crucial for a development process. For all these reasons, reusability and modularity are important properties of algorithms.

We will show that the approaches of algorithm development, represented by current algorithm libraries, do not support an appropriate level of these properties. Therefore, in this paper, we will apply an aspect-oriented approach to this field of research. To be more specific, we will use Caesar [8, 9] and try to improve the reusability and modularity of algorithm development.

We will base this approach on a set of criteria to evaluate reusability and modularity of algorithms in general. After showing that these criteria are not met by current algorithm libraries, we will present a framework for algorithm development, which uses Caesar's collaboration interfaces and its dynamic weaving to achieve better modularity and reusability. In addition, we will evaluate it using the criteria defined previously.

However, we want to point out that the current state of our work does not include the analysis of the speed of the algorithms. So far, we only concentrate on modularity and reusability.

For the field of algorithm development this work is interesting since aspect orientation offers a wide range of possibilities to achieve many of the goals of modular and reusable algorithm development. These goals are represented by the criteria Weihe set in his publications [12, 13].

We will first describe related work on algorithm development and on reusable and modular algorithms in chapter 2. In chapter 3 we will present a set of criteria of modular and reusable algorithms. Chapter 4 describes current approaches with respect to these criteria. Chapter 5 presents the framework, explains how to use it and why it has advantages over current approaches. Furthermore, we will evaluate the framework in chapter 6 and finally present results and future work.

2. RELATED WORK

2.1 Algorithm Development

To describe the current situation of algorithm development we will take a look at current algorithm libraries. We want to start with the programming language, which is used for most current algorithm implementations. This language undoubtedly is C++. This is the case since parts of the ANSI/ISO Standard C++ definition already aim at highly generic algorithms and data structures.

These parts of C++ are a subset of the STL, the standard template library [10], a library of algorithms, container classes and iterators, which uses generic classes or templates, as this technology is called in the C++ community. STL builds on top of object-oriented programming and combines it with certain features like templates.

The idea of STL is to achieve generic algorithms and data structures by defining categories of data structures. All members of one category share the same semantic requirements. Thus, the algorithms being implemented on such generic data structures become themselves generic. The idea is that this will lead to standard catalogues of algorithms and other functionality. This way code would not have to be rewritten and we would achieve fully reusable generic components.

Another library is LEDA, the library of efficient data types and algorithms [6]. It is also a library for C++ and it is specialized in graph problems as well as geometric and combinatorial

computations and more. Like STL, it is based on an object-oriented framework using parametric classes. LEDA also claims to be very easy to use, to create very reusable code while still remaining highly efficient, and to be easily extensible.

2.2 Reusable and Modular Algorithms

Though we have several libraries aiming to create reusable algorithms, there is only a very limited amount of work on how to achieve reusability and modularity. One can hardly find publications describing guidelines or criteria of algorithms to increase the reusability and modularity.

The only recent work, which analyses the use of different software engineering approaches to algorithms, is the work of Weihe [12, 13]. This work [12] focuses on reusable and efficient complex algorithms using object-oriented programming. As far as we know, this is the only major publication on problems of using object-oriented programming to improve algorithm reusability and modularity.

In the second work [13], Weihe chooses a much broader view on using software engineering approaches on algorithms. This paper can be seen as the basic work on evaluating approaches to increase modularity and reusability in the field of algorithm development.

It introduces the term *algorithm component* and concentrates on the term *adaptability* as a key goal to create algorithms, which are flexible enough to cope with unforeseen requirements, and which can be used in libraries of algorithms. Furthermore, it identifies two parts of the challenge to achieve adaptability. Part one is the adaptability to the so-called lower level, which means the underlying data structures. Part two is the adaptability to the higher level, which stands for the client using the algorithm component. Weihe also presents a collection of challenges and problems in each of these parts, a concrete and more detailed example (Dijkstra's algorithm), and a list of goals for reusable and modular implementations of algorithms.

Previous work on Caesar [8] aims at independent reusable building blocks and the integration of pre-build generic software components. This work gives an example of how to build a reusable and modular graph colouring algorithm using aspect-oriented programming. It also analyses the achieved degree of reuse and polymorphism.

Irwin et al. report on the implementation of sparse matrix computation using aspect orientation [3]. This publication is using different aspect-oriented languages and the main result is that it is possible to decrease the complexity and the lines of code, without significantly decreasing the efficiency of the computation.

Additional related work on software engineering and algorithms can be found in [1], which describes a conceptual and practical gap between object-oriented approaches on the one hand, and the way algorithms are taught on the other hand.

A working paper by Kuehl and Weihe [4] presents an approach to avoid a huge library of standard data structures. The authors use so-called domain design patterns to create reusability by reusing the algorithms and not the data structures.

In [5] and [11] the authors combine the technology of LEDA [6] with several concepts to achieve higher flexibility and efficiency while taking care of adaptable algorithms.

3. CRITERIA OF MODULARITY AND REUSABILITY

The usual problem of the work with terms like modularity and reusability is the problem of measuring. Therefore, we will present a structured set of criteria of modular and reusable algorithms.

3.1 Weihe's criteria

Especially for the world of algorithm development Weihe introduced a certain set of criteria [13]. As stated above, he defined two groups of criteria: the so-called abstraction from the lower level and the so-called abstraction from the higher level.

Therefore, we want to use the following criteria:

abstraction from the underlying data structure - lower level

- the underlying data structures should be kept variable without losing efficiency.
- an additional layer of indirection between the underlying data and the algorithm should be used.
- this layer should be modular and component-based to create a component structure of adaptors.

abstraction from the client code - higher lever

- at least in every main loop of the algorithm there should be a possibility to invoke code.
- the algorithm component should offer frequent stop points to adjust it to the client's needs, and maybe start it again later.
- full read access to the logical state at each stop point should be possible.
- preprocessing and postprocessing (general initializations and de-initializations) should be separated from the core of the algorithm.
- the core loops of several algorithms should be mergable into one loop.
- the specific format of the output should be modifiable with minor efforts.

These criteria directly aim at algorithm development. Thus, we base our set of criteria on this work. Appliances can be found in chapter 4 and in [13].

3.2 Criteria of Modularity

While Weihe concentrates on adaptability of the algorithm, we want to consider an even broader range of criteria. He decided to put the focus of his work on the properties needed to create an algorithm, which is adaptable to its surrounding, i.e. the data structure and the client. In the context of this work we also want to take a look at the reusability of the algorithm's sub-parts. We want to analyze the reusability that is inherent to the algorithm and its modules.

An approach for general criteria of modularity was created by Meyer [7]. His set of criteria consists of five goals. The first is *decomposability*, which means that the system can be and is decomposed into an efficient and detailed set of modules. *Composability*, the second goal, demands that every system is built out of modules. *Understandability* means that each single module is understandable for its own and *continuity* describes that code changes should only involve limited parts of the system. The

last criteria is *protection*, which demands that errors do also only affect a limited part of a system.

However, we will base our criteria of the inner structure on an adjusted set of Meyer’s work. An appliance of these criteria can be found in the chapter 4.

inner structure

- the algorithm should consist of well-structured modules (modules with completely defined contracts).
- independency of the modules should be guaranteed.
 - each module encapsulates its own functionality
 - changes have only to be made in one module
 - errors do only affect one module
 - each module can be tested on its own
- cooperation of the modules should be guaranteed.
 - modules can be substituted
 - modules can be added
 - modules can be removed (not essential)
 - modules are reusable in different contexts

3.3 Criteria of Usability

Weihe’s criteria of the lower level aim at maximal adaptability to different data types. This means that they favor algorithms, which can adapt to a maximal amount of different data structures. His higher level aims at maximal adaptability to different clients. Our new inner structure aims at maximal modularity of the algorithm itself and on maximal reusability of its parts. So, all the presented criteria favor algorithms with extreme flexibility: adaptability to data structures, adaptability to clients, and modularity.

However, flexible algorithms are not all we want. From our point of view reusability of a system also includes usability. In addition, we consider the ease of use of the technique and the methodology for achieving reuse. Thus, what is still missing is a group of criteria concerning the usability itself. Hence, we also want to introduce a group named “overall usability”.

overall usability

- natural (not artificial) separation of concerns
 - towards the user (what do I do, what does the algorithm do)
 - towards the algorithm and its modules
- minimal usage-complexity for the client
 - simple api
 - not too complex adaption
- possibility to debug the algorithm
- easy to learn
- easy to document
- efficient

We are aware that the criteria of overall usability are highly qualitative. But without these criteria a totally flexible but not usable solution could fulfill all criteria.

All together we come to a set of criteria, which we present in Table 1. We will use these criteria to evaluate the modularity and reusability of different approaches to algorithm development.

Table 1. Criteria of modular and reusable algorithms.

abstraction from underlying data structure - lower level
1. variable underlying data structures
2. additional layer of indirection
3. modular and component-based
abstraction from client code - higher lever
1. possibility to invoke code for the client
2. frequent points to stop the calculation
3. full read access to the logical state at each stop point
4. preprocessing and postprocessing separated from the core
5. mergable core loops
6. format of the output should be modifiable
inner structure
1. algorithm consists of well structured modules
2. independency of the modules <ul style="list-style-type: none"> - each module encapsulates own functionality - changes have only to be made in one module - errors do only affect one module - each module can be tested
3. cooperation of the modules <ul style="list-style-type: none"> - modules can be substituted - modules can be added - modules can be removed (not essential) - modules are reusable in different contexts
overall usability
1. natural separation of concerns, not artificial <ul style="list-style-type: none"> - towards the client - towards the algorithm and the modules it consists of
2. minimal usage-complexity for the client <ul style="list-style-type: none"> - simple api - not too complex adaption
3. possibility to debug
4. easy to learn
5. easy to document
6. efficient

4. CURRENT SITUATION

In this chapter we will show that current approaches to algorithm development, represented mainly by algorithm libraries, do not meet all criteria described above or create problems by doing it.

4.1 Lower Level

Today we are used to the fact that most of the time we have only two possibilities to use algorithms in a project. Possibility one is to use an already implemented one. In this case we have to adjust our data to fit to the algorithm. But if we have slightly complex data, most of the time we face a big problem. Sometimes we even have to translate the data to another representation.

Possibility two is to implement the algorithm again. This way we can customize it and use the data we want to use. However, both possibilities are not satisfying from a software engineering point of view.

STL achieves an abstraction from the underlying data structure. This is implemented using linear iterators, as they are known from [2]. As Weihe described in [13], these iterators can form an additional layer of indirection. He also points out that the data stays variable when polymorphism is used. This way the layer

would also be modular and component-based. Therefore, all criteria of the lower level would be met. A very simple example is the code of STL's find method, which shows the use of iterators in STL:

```
template <class InputIterator, class T>
InputIterator find( InputIterator first,
                  InputIterator last, const T& value) {
    while (first!=last&&*first!=value) {
        ++first;
    }
    return first;
}
```

However, this only works for containers of linear data. When working with trees for example, the simple iterator concept cannot be used any more. For this problem so-called adjacency iterators were developed, which iterate over a non-linear data structure by walking over the data in a pre-defined way. E.g. for a tree this may be a depth-first traversal.

But still it is questionable to wrap all possible data structures with iterators. Especially, since in this case, the algorithms also have to work only with linear iterators. For example, it would be more intuitive if a graph algorithm could work on data structured as a graph and not as a set of linear iterators.

However, the abstraction from the lower level is achieved by current algorithm libraries, but limited, since it focuses on linear data types and therefore provides only imperfect solutions.

4.2 Higher Level

The criteria of the higher level are usually achieved using design patterns like the strategy pattern or the façade pattern [2].

To invoke code for the client in an object-oriented environment certain hooks can be implemented. Though this slightly slows down the algorithm, it achieves the first criteria of this group. Using hooks and methods implemented by the algorithm (e.g. to stop the calculation or to save the current logical state) can be used to create stop points, as required by criteria 2. However, hooks and methods like this always imply that the programmer of the algorithm was aware that an extension might be needed. This is obviously a limitation.

The logical state, defined by [13] as the set of information needed to rebuild the exact same situation where the algorithm was stopped, is implemented with a simple getter of the important data. Access can be gained by using hooks. Pre- and postprocessing are easy to separate from the algorithm by simply implementing the initialization and de-initialization in other classes. Mergable core loops are a big problem using object orientation and not achieved by current approaches. The output of the algorithm is easy to modify using object-oriented methods like polymorphism or patterns like the template method.

However, the criteria of the higher level are fairly achieved but include some downsides. The most obvious one is that a developer has to know where a client wants to invoke code or which methods she needs to customize the algorithm.

4.3 Inner Structure

So far many of the criteria have been achieved, but as we will see now, this was done for the price of a rather low modularity or a complicated structure.

Looking at modularity and at STL we can note that iterators used to access the data have to be written in the context of the container class. Thus, when implementing the container the

programmer has to know the iterators the container should work with. This obviously is not a good modularisation. It would be favourable if the iterator would be written outside of the data and outside of the algorithm.

Looking at the criteria we note that in STL there is no technique or method on how to build the inner structure of algorithms. Most algorithms are implemented like black boxes and offer only very limited possibilities of customization. The reason is that the implementation (e.g. using hooks) of these possibilities changes the algorithm even when they are not used. Instead of a lightweight and precise algorithm we would have a more flexible, but heavy and large algorithm. This situation leads to today's very static implementations, which are not using modules and which have very limited or no possibilities to customize or to extend them. However, an algorithm can be developed in a way to achieve these goals, but current algorithm libraries do not provide this possibility. Therefore, the criteria are not met.

4.4 Overall Usability

The separation of concerns in the current approaches is mainly object-oriented. For a client this separation is somehow intuitive, but limited, since there is no choice of any other decomposition. Since most current approaches use a black-box-like algorithm we have no modularisation and the second part of this criteria is not met.

An advantage of these approaches is the ease of running the algorithms, since the API is very small. However, it is not always easy to use the algorithms, since we might have to provide iterators, which translate the data to a linear structure to be used by the algorithm.

Since the current approaches are based on proven languages they are fairly easy to learn and to document. However, debugging a template based C++ code can be a problem.

Therefore, we can say that the first criteria is not achieved, since no modularisation can be found. The current approaches are fairly easy to use, but get into trouble when it comes to non-linear data structures. Learning and documenting is no problem, but debugging can be one.

5. THE FRAMEWORK

In order to avoid the described disadvantages of current algorithm libraries and to meet the criteria defined above, we have created a framework using the aspect-oriented language Caesar [8, 9]. Caesar is especially well suited to our problem domain, because it has special means to create reusable components that can easily be adapted to a new context. Compared to AspectJ in combination with aspect-oriented patterns, it also has some advantages when separating aspect functionality and weaving code, as shown in [9]. Caesar enables this using so-called collaboration interfaces, which we will describe in the following and which are one of the two key ideas of our solution. The second one is the possibility of dynamic weaving, which will also be presented.

The following framework includes several Caesar-specific patterns, which enable us to adjust the algorithm dynamically to our needs. An overview of the framework is presented at Fig. 1.

It is based on a set of collaboration interfaces (displayed in the rectangular gray-shaded area). A collaboration interface is an interface for a (crosscutting) concern, which differs from a Java

interface since every method and attribute has to be declared either *provided* or *expected*.

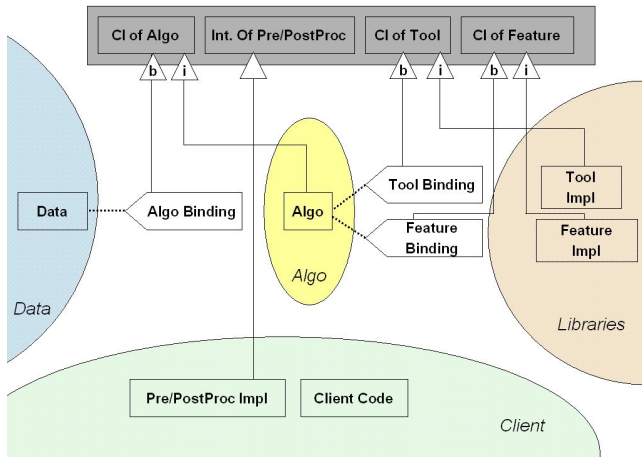


Figure 1. Inheritance-structure of the framework. Inheritance relations with *b* represent bindings, relations with *i* implementations.

In general, collaboration interfaces can have a nested structure of sub-interfaces, thereby enabling the modeling of a collaboration of multiple abstractions. For an example of shortest-path-algorithms, we use two nested interfaces: one representing vertices and another one representing edges. The code of this collaboration interface might look like the following:

```

collaboration interface ShortestPathAlgo {
    provided void calculate();
    ... //other methods

    nested interface Vertex {
        expected EdgeList getEdges();
        expected void setShortestPath(
            Value v);
        expected Value getShortestPath();
    }
    nested interface Edge {
        expected Value getLength();
    }
}

```

The method *calculate*, representing the method to start the algorithm, is declared *provided*. This stands for methods or attributes which belong to the functionality of the aspect.

The methods *getLength*, *getEdges*, *setShortestPath* and *getShortestPath* in the nested interfaces are declared *expected*. This stands for methods or attributes which belong to the so-called binding of an aspect. A binding encapsulates all the code, which is specific towards the surroundings of the aspect, including all the pointcuts.

To create a concrete collaboration interface the programmer has to implement at least one binding and at least one implementation. In the following we see the implementation *Dijkstra* and the binding *ShortestPathAlgo_Database_Binding*:

```

cclass Dijkstra implements ShortestPathAlgo {
    void calculate () {
        ... //code
        Value foo = someEdge.getLength();
        ... //code
    }
}

```

```

cclass ShortestPathAlgo_Database_Binding binds
ShortestPathAlgo {

```

```

    cclass Vertex_Database_Binding binds
    Vertex {
        EdgeList getEdges() {
            return
                VertexDB.getEdges(vertex);
        }
        void setShortestPath(Value v) {
            ResultDB.store(vertex, v);
        }
        void Value getShortestPath() {
            return
                ResultDB.load(vertex);
        }
    }

    cclass Edge_Database_Binding binds Edge {
        Value getLength() {
            return
                EdgeDB.getLength(edge);
        }
    }
}

```

But simply writing the binding and the implementation separate of each other is obviously not enough to create a running class. Thus, a binding class and an implementing class have to be put together in a so-called weavelet. A weavelet, combining bindings and implementations of a collaboration class, can be instantiated and can be used to create objects. It might look like the following:

```

cclass Dijkstra_On_Database extends
Dijkstra & ShortestPathAlgo_Database_Binding;

```

The special point in this case is that a binding will use methods of the implementation, which were defined in the collaboration interface. It is also the same vice versa: an implementation will use the binding's methods, which were defined in the interface. Thus, every binding can be combined with every implementation so that we achieve a n:m relationship. The programmer can combine a binding with an arbitrary implementation or one implementation with whatever binding simply by defining a weavelet to do so.

5.1 The Algorithm and its Data

If we see an algorithm as an aspect, a binding can work like an interface to the data. It represents the data in the context of the algorithm. This way the binding translates the original, i.e. underlying, data structure to a language, which can be understood by the aspect's implementation, which is the algorithm. In this case, a binding might wrap around a database. So it will translate the database-access to the expected methods *getEdges*, *getLength*, *setShortestPath*, and *getShortestPath*, which can be understood by the algorithm. To write a new graph algorithm a programmer has to implement a collaboration interface. Looking at the example given above a developer has to implement the collaboration interface *ShortestPathAlgo*. Thus, the algorithm (i.e. the functionality of the aspect) can work on any kind of underlying data (in this case a database) as long as it is translated by a binding. Thus, criteria 1 of the lower level is met.

In other words, an algorithm can easily be reused on different data by exchanging the binding. Thus, the bindings work as an additional layer between the underlying data and the algorithm, which meets criteria 2 of the lower level. Vice versa, on one set of data, different shortest-path-algorithms can be run by simply

exchanging the algorithm. This is possible since the binding and the implementation are always combined in a weavelet, which is defined by the client.

This creates a high level of modularity since one functionality of an aspect can be reused in every surrounding simply by exchanging the binding. On the other hand different aspects can be used in one surrounding by simply exchanging the implementation-class in the weavelet.

Compared to the current approaches this enables us to use not only linear iterators. Since the collaboration interface defines the binding, the data structure, which fits best to the algorithm, can be used. This can be a linear iterator, but also a graph-like structure with vertices and edges, or any other structure. In addition, the combination of binding and implementation in a weavelet is not more difficult than using a template in STL. Thus, we gain some advantages concerning the criteria of the lower level.

It is also possible to inherit bindings from more than one other binding, implementations from more than one implementation, collaboration interfaces from other ones etc. So Caesar enables the programmer to use the full functionality of multiple inheritance. This allows a high modularity of the bindings, so that criteria 3 of the lower level is met.

5.2 Tools and Features

To create reusable parts of the algorithms we defined two types of modules. The first group is called *tools*. A tool is an essential part of an algorithm. In other words, the algorithm would not work without it. An example would be a comparator-like module, which is required by an algorithm.

In addition we have so-called *features*, which are optional parts of algorithms. Examples are some kind of progress information (e.g. used in a progress bar) or a feature to stop the algorithm at a certain point.

For every specific tool and feature a collaboration interface has to be defined. The implementation of it represents the functionality of the tool or feature. The binding represents the connection to the algorithm.

An example of a tool would be a comparator. A simple comparator obviously needs no pointcut or algorithm-specific code and therefore also no binding. It would work with a simple implementation. But we can also offer the possibility to use a comparator with time-stamps. Instead of changing the algorithm to initialize the timestamps, we might simply weave in the needed code using a binding. In this case, the implementation of the comparator encapsulates the usual comparator functionality, which is specific for a certain data-type, while the binding encapsulates the algorithm-specific weaving code. This way we can reuse the implementation of a comparator by combining it with several algorithms. We can also use one algorithm with arbitrary comparators and data types by exchanging the implementation, while the binding stays the same. Thus, we use the possibility of weaving to connect the (crosscutting) tools to the algorithms in a very flexible and dynamical way.

To give an example of a feature we may have one, which visualizes shortest-path-graph-algorithms. One possible implementation may produce a colored graph where all the updated vertices are displayed in real-time, another one may simply print out the last changed vertex and its shortest path.

The bindings connect this information to a specific algorithm (in our case *Dijkstra*) and weave in the necessary code. So here we use Caesar's dynamic weaving primarily to connect two modules. This way we can connect the algorithm and the feature in the most flexible way, i.e. dynamically and during run-time. Only a secondary aim is to handle crosscutting features. The implementations encapsulate the pure functionality of printing the texts or drawing the updated graph. A simple example of a feature like this is presented in the following.

```

collaboration interface VisualizeShortestPath
extends Feature {
    provided void updateVertex(
        ShortestPathAlgo.Vertex vertex);
}

cclass VisualizeShortestPath_Dijkstra_Binding
binds VisualizeShortestPath {

    ... //other Dijkstra-specific code

    after (Dijkstra.Vertex vertex): call(void
        vertex.setShortestPath (Value))
        { this.updateVertex(vertex); }
}

cclass VisualizeShortestPath_StdOut implements
VisualizeShortestPath {

    public void updateVertex
        (ShortestPathAlgo.Vertex vertex){

        System.out.println(
            " Vertex #" + vertex.getNumber() +
            " New Shortest Path:" +
            vertex.getShortestPath().toString()
        );
    }
}

cclass VisualizeShortestPath_Dijkstra_StdOut
extends VisualizeShortestPath_Dijkstra_Binding
& VisualizeShortestPath_StdOut {}

```

All the tools and features are stored in a library and can be connected to every algorithm by using or writing a binding. The modularity of the tools and features leads to an algorithm, which looks nearly like pseudo-code. There are no additional lines of code (e.g. for hooks), only the necessary code to create the pure algorithm functionality.

Therefore, the framework presents a possible structure of component-based algorithm modules. Algorithms in this framework are built out of modules, which can be added, exchanged or removed and which encapsulate errors, local changes and testing. Thus, we achieve the goals of the inner structure.

5.3 Using the Framework

5.3.1 Connecting Algorithm and Data

The client obviously knows the data-structure the algorithm should work on. She can decide for the algorithm she wants to use and she can connect the algorithm to the data by simply defining a weavelet, which combines the wanted algorithm and a binding. This effort is equal to the effort of creating a concrete class out of a template in C++.

The binding might already exist for the type of data she is working on. If it does not, she obviously has to write it. But in

opposite to using STL iterators, she can create a new binding outside of the context of the container. The binding is written as a simple stand-alone module binding the collaboration interface.

5.3.2 Connecting Algorithm and Tools or Features

In a second step she has to define the tools and features her algorithm should have. (Here we again want to point out that the possibility of customization and modularity of the algorithm does not exist in approaches like STL or LEDA.) For example she could decide for an algorithm using a certain type of comparator and a special iterator, but also with a progress bar.

By using inheritance of collaboration classes Caesar is able to use polymorphism on aspects. Thus, we are able to dynamically adjust the program to certain circumstances. This possibility creates a dynamic feature and the client can choose during runtime, which weavelet should be used. This is called on-demand-remodularization since depending on the situation during runtime, i.e. depending on the demand, different modularizations (in form of different combinations of binding and implementation) can be used dynamically. Thus, she combines the tools and features simply by defining the weavelets to use them on the wanted algorithm.

The tools are created as part of the preprocessing. This is the case since a weavelet of a tool's collaboration interface is required by the algorithm.

Caesar, like AspectJ, allows static weaving. But it also allows us to weave an aspect in a certain code dynamically. Unlike AspectJ, where there are dynamic pointcuts like cflow, but where the pointcut still is hard-coded at run-time, here we have the possibility to dynamically decide for an aspect and a pointcut. This can be done by a "deploy" statement, which weaves a certain aspect into the code. By using Caesar inheritance and polymorphism we can decide during runtime which aspect we want to weave in. Especially for modular design it is very important that different modules can be combined with small effort. This way the client can connect every feature to the algorithm. Extending the previous examples of our Dijkstra-implementation and of the visualization of shortest-path-graph-algorithms, we can present the following sample-code:

```
ShortestPathAlgo dijkstra =
    new Dijkstra_On_Database();

... //more code

VisualizeShortestPath visualize=null;

if (userWantsStdOutFeature) visualize = new
    VisualizeShortestPath_Dijkstra_StdOut ();

deploy(visualize){
    dijkstra.calculate();
}
```

5.3.3 Additional Possibilities for the Client

The client can also write new features, which can include arbitrary functionality, and which can be connected to the algorithm by writing corresponding bindings. This way she is able to weave code at an arbitrary join point in the algorithm to invoke the wanted functionality. Using a feature she can also access the logical state or stop the algorithm.

Even more, Caesar is also able to work with more than one binding or implementation. It is possible to write several bindings,

which each take care of one special element in the surrounding. The weavelet can afterwards simply combine a higher number of bindings and implementations to create the needed running class. Coming back to our example, we can split the binding of the algorithm. One part takes care of the database access to the *EdgeDB*, another one to the *VertexDB* and a third one to the *ResultDB*. Encapsulating the latter one means encapsulating the format of the output in a binding. Thus, the client can decide which output she wants.

The same is true for the implementations. It is possible to use several implementations to split up parts of the functionality in different modules and to combine them later on.

Important is only the fact that the weavelet combines enough bindings and implementations to get code for all methods and attributes, which were defined in the collaboration interface. This again increases the modularity of the system.

6. EVALUATION

6.1 Lower Level

As described in 5.1, the bindings enable the algorithm to use variable underlying data structures, and they work as a modular and component-based layer of indirection between the algorithm and the data. Therefore, the framework meets the criteria of the lower level. In addition, we achieve a higher modularity and avoid the problem of linear iterators by allowing any structure of a binding.

6.2 Higher Level

Using aspect orientation and the framework's features we can invoke code at any join point in the algorithm, as described in 5.3.3. Obviously, this also slows down the algorithm, but only in the case we do invoke code, i.e. deploy a feature. If we do not, then the algorithm's efficiency will not be altered. In addition, the number of points to invoke code is increased and it is possible to extend algorithms even in a way the algorithm's author did not anticipate. As described above, we can also stop the algorithm and access the logical state. The pre- and postprocessing is separated from the core algorithm as part of the framework itself, as one can see in Fig. 1. The format of the output is modifiable using different bindings, as we also described in 5.3.3.

A special point is that the core-loops of different algorithms are mergeable with Caesar. The technique is outside of the scope of this paper, but we want to present a short glance at the basic idea. Let us assume that an algorithm A has a method, which encapsulates the entire inner code of the core loop. An additional binding of A weaves into the code of algorithm B at the right positions and calls the core loop method of A. If a weavelet containing A and the additional binding is deployed around the calls of B, then the two algorithms will be merged.

Therefore, the criteria of the higher level are achieved and the problems of the current approaches are avoided.

6.3 Inner Structure

As described in 5.2, the inner structure of the framework's algorithms consists of well-structured modules. These modules (tools and features) encapsulate certain functionality and meet the criteria of the independency of the modules. They also can be substituted, added or removed in an easy and straightforward way,

as described above. In addition they are reusable. Thus, we achieve the goals of the inner structure.

6.4 Overall Usability

The separation of concerns of this framework is very strict and from our point of view, it is intuitive for the client and for the programmer of the algorithm.

We think that this framework's usage complexity for the client is at least comparable to the usage complexity of any other algorithm library. If the data-structure the client is using is close to the one the algorithm expects, the binding is a simple wrapper. If the client's data is very different from the algorithm's data structures then the binding will get complicated, since it has to translate the client's data. But using current algorithm libraries with not compatible data-structures leads to the same problem. In addition, using these libraries the client's data always has to be translated to some kind of linear iterator. Also to mention is that the algorithm library presented in this paper enables the client to customize the algorithms (even during run-time).

The possibilities to debug Caesar are currently not developed. Therefore, this criteria is not met by our approach. Caesar is relatively easy to learn, since it is a Java and AspectJ related language. On the other hand, switching to multi-dimensional-separation-of-concerns is not easy for most programmers, since the old object-oriented thinking is very strong. However, a complete and quantitative evaluation of learnability was outside of the scope of this work.

The documentation is as easy as it is with all Java related languages. The efficiency is, as described before, not in the current focus of the work. However, we estimate that it would not be much slower than a Java implementation of the algorithm. Still, measurements do not exist, and this is an essential part of the future work.

7. RESULT AND FUTURE WORK

We presented an approach to use aspect orientation on the field of algorithm development. To be able to evaluate modularity and reusability we introduced a set of criteria for algorithm development, which is based on the work of Weihe and Meyer. We described the current situation in the field of algorithm development by presenting current algorithm libraries. In the following, we showed that these libraries lack several of the criteria.

In this paper we used Caesar to implement a framework for algorithms to avoid the disadvantages of these technologies. The key ideas of our approach were Caesar's technology of collaboration interfaces and the possibility of dynamic weaving. We presented the framework, explained how to use it and why it has advantages over current algorithm libraries. Finally, we evaluated the work based on the criteria and came to the result that the framework meets most criteria.

The future work consists of primarily testing the efficiency of the approach. This work is closely related to the analysis of the efficiency and speed of Caesar itself.

In addition, we will analyze the aspect-oriented patterns we applied in this work and try to use this knowledge to get more experience in designing aspect-oriented applications.

The described set of criteria might also be reused for further analysis on how to measure reusability and modularity in general.

8. REFERENCES

- [1] Bucci, P., Heym, W., Long, T. J., and Weide, B. W. Algorithms and Object-Oriented Programming: Bridging the Gap. *ACM SIGCSE Bulletin* 34. (Mar. 2002).
- [2] Gamma, E., Helm, R., Johnson, R., and Vlissides, V. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
- [3] Irwin, J., Loingtier, J., Gilbert, J. R., Kiczales, G., Lamping, J., Mendhekar, A., and Shpeisman, T. Aspect-Oriented Programming of Sparse Matrix Code. In *Proceedings International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE)*, (May 1997), Springer-Verlag. In Lecture Notes in Computer Science 1343.
- [4] Kuehl, D., and Weihe, K. *Using Design Patterns for Reusable Implementations of Graph Algorithms*. Working Paper - Konstanzer Schriften in Mathematik und Informatik, Universität Konstanz, Jan. 1996.
- [5] Kuehl, D., Nissen, M., and Weihe, K. Efficient, Adaptable Implementations of Graph Algorithms. In *On-Line Proceedings of the 1st Workshop on Algorithm Engineering (WAE '97)*, 1997.
- [6] Mehlhorn, K., and Naher, S. *LEDA: A Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
- [7] Meyer, B. *Object-Oriented Software Construction*. Prentice Hall International Series in Computer Science, Prentice Hall, 1988.
- [8] Mezini, M., and Ostermann, K. Integrating Independent Components with On-Demand Remodularization. In *Proceedings of OOPSLA '02*, ACM SIGPLAN Notices 37(11) (2002), 52–67.
- [9] Mezini, M., and Ostermann, K. Conquering Aspects with Caesar. In *Proceedings of the Conference on Aspect-Oriented Software Development (AOSD) (2003)*, ACM Press, pp. 90–99.
- [10] Musser, D. R., and Saini, A. *STL Tutorial and Reference Guide*. Addison-Wesley, Reading, MA. 1995.
- [11] Nissen, M., and Weihe, K. *Combining LEDA with Customizable Implementations of Graph Algorithms*. Technical Report 17, Fakultät für Mathematik und Informatik, Universität Konstanz, Oct. 1996.
- [12] Weihe, K. Reuse of Algorithms: Still a Challenge to Object-Oriented Programming. In *Proceedings of OOPSLA '97 (1997)*, Vol. 17, ACM Press, 34–48.
- [13] Weihe, K. A Software Engineering Perspective on Algorithms. *ACM Computing Surveys* 33, 1 (Mar. 2001), 89–134.