Open Components *

[Position Paper]

Andreas Gal, Wolfgang Schröder-Preikschat, and Olaf Spinczyk
University of Magdeburg
Universitätsplatz 2
39106 Magdeburg, Germany
{gal,wosch,olaf}@ivs.cs.uni-magdeburg.de

ABSTRACT

Traditional binary component models do not support the application-specific customization of components at integration-time [1]. This problem cannot be solved by introducing new component languages, because it is rooted in the properties of executable, binary code and not in the implementation language of the component. In this paper we suggest using *"open components"* instead of binary component models. This novel approach uses source code as the distribution form of components and allows a flexible, tool-supported customization with respect to the application environment. The result is a better chance to reuse a component through adaptable interfaces and a very efficient implementation of the composed software product.

1. MOTIVATION

A key factor for the success of component systems, especially in resource restricted application domains, is the overhead imposed by the component framework. Component software should not require significantly more resources than a non-component based implementation.

In traditional binary component models, especially those which are independent of the implementation language and thus have to use mechanisms like function argument marshaling, the communication over component boundaries is significantly more expensive than a similar plain language-level procedure call. Arising from this, real-world components are often limited in their granularity as many small components would increase the overall communication overhead.

*This work has been partly supported by the German Research Council (DFG), grant no. SCHR 603/1-1 and SCHR 603/2.

[1]the point in time where several components, which might come from different vendors, are composed to form an application

Designing the inner structure of a component is a trade-off between flexibility with respect to all possible applications and optimality with respect to the specific contexts the component could be used in. The problem becomes more obvious by looking at a simple example (figure 1).

Figure 1: Flexible vs. specialized structure

In this example a part of a memory management component is presented. The left structure allows the selection of a memory allocation strategy at run-time through an abstract interface (`Strategy`). This flexibility can be useful for some applications, but can impose a severe overhead for applications that only need a single allocation strategy. The requirements of these applications could already be served with a simplified component structure as shown on the right-hand side.

Binary components can be parameterized at run-time using application-specific configuration information. This allows to disable certain functionalities inside a component, which might not be needed for all applications. Typical component implementations follow this "one fits all" approach. On the other hand the code for the disabled functionalities still resides in the component. This imposes an overhead on those applications that do not require these particular functionalities. To overcome this limitation it would be necessary to adjust both, the internal architecture of a component, and its external interface.

With a binary component model, it is close to impossible to create an optimized version of a complex, feature-rich component according to the specific application context. Much more information about the component internals would be needed than the plain interface description of a binary component can deliver. On the other hand, even with sufficient information available to decide for certain optimizations of the component, it is technically very difficult to perform the necessary manipulations like exchanging base classes and eliminating dead code on binary code.

Using run-time mechanisms like dynamic (on demand) load-

ing of classes is only a partial solution here, because it saves some memory space but requires an expensive, run-time consuming, infrastructure. Better than a blind run-time optimization would be an informed customization of the components at integration-time.

In the following sections of this paper we will present *"open components"* as a solution that follows the ideas discussed so far. Open components reconcile the advantages of binary components such as reusability, robust integration, and information hiding with the efficiency of application-specific structures and interfaces.

We will start with the rationale of open components in section 2. In section 3 we will discuss the application-specific customization of components using a "component customizer"tool. The paper ends with a discussion of related work in section 4 and our conclusions in section 5.

2. OPEN COMPONENTS

To overcome the described problems to customize binary components, we suggest using components distributed in source code. A main requirement on components is that *"a software component can be deployed independently and is subject to composition by third parties"* [7]. In our opinion with source code components this is well possible, if an adequate infrastructure is provided. We call these source code components *"open components"*, because the component internals are accessible and thus can be subject to application-specific optimizations.

Possible customizations are the elimination of unused code or data and the avoidance of unnecessary condition checks at run-time. To express the customization requirements some abstract component view has to provided where needed parts can be selected. It has many advantages to use a class diagram for this purpose: Class diagrams are a widely-known and implementation language independent concept. They can be exported by case tools or automatically be derived from the source code. Classes are an abstraction of related code and data, thus removing a class removes both at the same time. Run-time condition checks are often implemented by dynamic binding in object-oriented software. So a class diagram requirement can remove this as well. An example requirement specification following this idea will be presented in section 3.

Open components may depend on classes in the environment to do their job. These classes could be part of other components or be provided by some application code. This allows components to delegate work to other components that are better specialized. Delegation is essential for reuse and this is the main aim of components. For example an I/O library component is specialized on formatting and buffering of data. A device driver component is specialized on hardware access. While the I/O library needs a device driver to be useful it is not interested in the configuration options of device drivers or any other technical details. Both can be configured independently when they are plugged together, only the required and the provided interface have to match. Reuse comes into play if a third component also uses the device driver.

The interesting questions are, how these components can be developed independently and how the interface compatibility can be checked. Of course the component cannot be developed without the classes it depends on. Instead, the component code can contain a "dummy" class that has exactly the required interface. The methods of this class do not need to be shipped because the component customizer forces the user to change the class relations so that a real class is used instead. For the development of the component a testbed is used, which has to simulate possible environments (see figure 2).

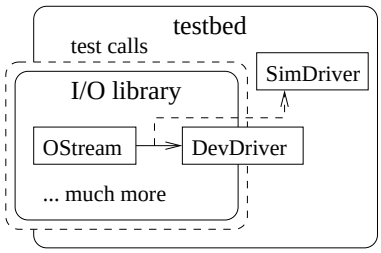

Figure 2: A component testbed

The testbed has to issue test calls and provide all classes the tested component depends on (`SimDriver` in figure 2). The interface compatibility can be checked by the component customizer with the integrated parser. It checks whether all method calls and member accesses that are possible on the dummy class will also be successful on the checked class. With the dummy class mechanism components can be developed independently and connected overhead-free at integration-time.

If a component A depends on another component B there must be a requirement specification, which describes the required parts and structure of B on behalf of A. An interesting situation arises when a third component C also has some requirements on B, because there could be a conflict. As an example consider the memory management classes from figure 1. A could require an aggregation of `BestFit` into the `MemoryManager` while component C could prefer `FirstFit`.

The conflict can be solved by keeping either the abstract base class or by generating both versions of class `MemoryManager` [2]. The selection of one solution is a run-time vs. code size trade-off, which is decided with a hint from B's component description. In both cases there will be a feed-back into the requiring components A and C. For example, for the second solution the name of the class `MemoryManager` must be replaced. For the first solution a strategy object must always be instantiated and handed over to each `MemoryManager`. This manipulation is done automatically. The code of A and C can still use a simple API.

New components can be constructed in a hierarchical manner, i.e. a component can have sub-components. Such composite component decides how its sub-components are connected or if a dependency will be left open. The top-level component is the application itself. At least here all dependencies must be closed.

[2] this is similar to a template instantiation in C++.

3. COMPONENT CUSTOMIZATION

A robust integration of open components is only possible with tool support. Any manual access to the source code is error prone and must be avoided. In our implementation we have automated the component specialization and integration process with a component customizer tool. The implementation is based on the PUMA [6] code transformation system for C++. The tool provides the following basic transformations:

- Selection of classes: non-required classes can be removed if no other class depends on it.
- Renaming of classes: used to avoid name clashes if two independent components declare a class with the same name.
- Copying of classes: applied if the same class is needed with different class relations at the same time.
- Simplification of class relations: this can be a modification of a relation type or a change of the connected partners.
- Binding of components: done by replacing "dummy" class references with real references.

The whole transformation process is based on XML descriptions of the components, i.e. their interface and structure, and the component relations. Figure 3 shows a graphical representation of four component descriptions at once: the composite component `MemoryManagement` and its three subcomponents `Economist`, `Strategy`, and `MemBlockManager`. The open dependencies `EcoStrategy` from the `Economist` component and `BlockManager` from `Strategy` are closed. Two other dependencies are left open.

Besides that a component description may contain a specification of requirements on its sub-components. As an example figure 4 presents requirements on the memory management component as they might come from the top-level "Application" component.

This specification means that the classes `UserEco`, `StrategyDecorator` and `FirstFit` are required. All other classes will automatically be removed, if none of the required classes will need it after the customization.

The "binding" attribute defines the way that the corresponding class will be used by the enclosing component. `StrategyDecorator` and `FirstFit` do not needed a "binding" specification, because they should not be used directly. They are only mentioned to require a change of component-internal class relations (see below). The "binding" attribute is used by the component customizer to do a virtual function call optimization. For instance, if it is known which classes are instantiated the customizer can eliminate function implementations that can never be called. If it is further known which classes are referenced functions can be "devirtualized", which saves the code and run-time needed to perform the dynamic binding.

The "relation" tag is used to require changes of class relations. For example, `FirstFit` should aggregate an instance of the class `MDMDoubleLink` instead of referencing any kind of `MemBlockMgr` via an abstract interface class.

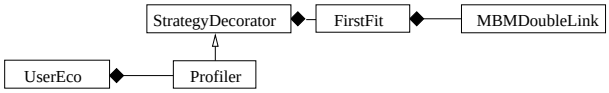

Figure 5: The memory management component after customization

Figure 5 shows the result of the component customization with the requirement specification of figure 4. With this structure the application gets a very specialized and easy to use API. The required class `UserEco` can simply be instantiated without any parameters. Furthermore the resulting code is very efficient. All abstract classes were removed, thus no virtual functions waste memory space and run-time. Measurements [2] showed that the needed memory space could be reduced to 33% and run-time to 30% of the non-specialized code without considering the size of classes that could be completely omitted.

4. RELATED WORK

Source code components can also be constructed using a meta language. For instance, the approach of Kamin et. al [4] describes a functional meta language that provides a data type *Code*. This can be used to generate components from "object language" code fragments. The result are customized, lightweight implementations as provided by open components. While this approach is very powerful, we doubt that an explicit representation of code is really necessary here. At the same time it introduces the complexity of a new language. Open components on the other hand can be easily designed and implemented with well known design patterns[3].

Some object-oriented programming languages offer metaprogramming constructs. C++, for instance, can be used with its template specialization feature to implement template metaprograms. We expect, that the structure modifications, which we do with a code transformation system, are possible with templates as well. Nevertheless this metaprogramming technique implicates that the structural variability is visible in the component source code. We would accept this if the variability were class specific, but it is of a global nature. For example, every pointer to an interface class is a point of configuration and can become a pointer to some other class. From this point of view the variability of open components that is derived from the class diagram without any differences between individual classes is an aspect in the sense of AOP[5]. The best way to implement an aspect is to separate the code.

5. CONCLUSIONS AND FUTURE WORK

In this paper we described the concept and some results of our experiments with open components. The presented integration-time customization is a technique to reconcile reuse and specialization: one of the most important questions in component-oriented programming.

The result of this approach is a different view of a component. It is not a "one fits all" implementation. Instead of

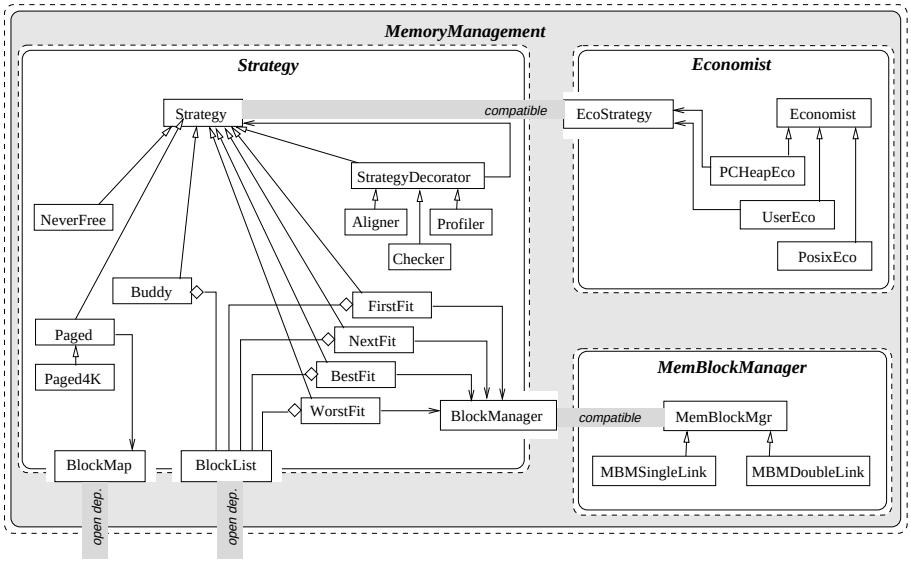

Figure 3: An example: memory management components

```
<requirement component="MemoryManagement">
    <class name="UserEco" binding="aggregate">
        <relation name="strategy" type="aggregate" class="Profiler">
    </class>
    <class name="StrategyDecorator">
        <relation name="strategy" type="aggregate" class="FirstFit">
    </class>
    <class name="FirstFit" binding="aggregate">
        <relation name="blockmgr" type="aggregate" class="MBMDoubleLink">
    </class>
</requirement>
```

Figure 4: A component requirement specification

that it is a building plan for a whole family of implementations from which a client can select the best.

On the language level this did not require a new "component-oriented" programming language. Besides the object-oriented implementation language only XML-based descriptions were used. This design decision was not a matter of convenience but of separation of concerns.

In future work we will investigate how feature models [1] could be used for the selection of open component family members and find solutions to protect a component vendor's intellectual property.

6. REFERENCES

[1] K. Czarnecki and U. W. Eisenecker. *Generative Programming - Methods, Tools, and Applications.* Addison-Wesley Publishing, 2000.

[2] A. Gal, W. Schröder-Preikschat, and O. Spinczyk. On Minimal Overhead Operating Systems and Aspect-Oriented Programming. In *Proceedings of the 4th ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSWS'2001)*, Budapest, Hungary, June 2001. ISBN 84-699-5329-X.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN 0-201-63361-2.

[4] S. Kamin, M. Callahan, and L. Clausen. Lightweight and Generative Components I: Source-Level Components. In *K. Czarnecki, U. W. Eisenecker (Eds.): Generative and Component-Based Software-Engineering. First International Symposium, GCSE'99*, Erfurt, Germany, Sept. 1999. Springer-Verlag. Revised Papers. LNCS 1799.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.

[6] O. Spinczyk and M. Urban. The PUMA Project Homepage, 2001. `http://ivs.cs.uni-magdeburg.de/~puma/`.

[7] C. Szyperski and C. Pfister. Workshop on component-oriented programming. In *ECOOP 1996 Workshop Reader*, 1997.