Scenario Adapters: Efficiently Adapting Components[*]

Antônio Augusto Fröhlich
GMD-FIRST
Kekuléstraße 7, 12489 Berlin, Germany
guto@first.gmd.de

and

Wolfgang Schröder-Preikschat
University of Magdeburg
Universitätsplatz 2, 39106 Magdeburg, Germany
wosch@ivs.cs.uni-magdeburg.de

Abstract

In this paper we consider the utilization of component-based software engineering techniques for the development of adaptable systems compromised with performance. The SCENARIO ADAPTER construct is proposed as an effective means to achieve this goal. It can be used to adapt scenario-independent abstractions to an execution scenario known at compile-time. The efficiency of SCENARIO ADAPTERS has been demonstrated in the Project EPOS, which aims to deliver, whenever possible automatically, a tailored run-time support system for each application.

Keywords: component-based software engineering, static adaptation, template metaprogramming.

1. Introduction

Object-orientation together with component-based software engineering is enabling the long dreamed production of software as an assemblage of ordinary components. Similarly to other industries, software developers can nowadays reuse components, reducing costs and accelerating production. Moreover, in comparison to other assembly lines, for instance to the largely acclaimed automotive industry, component-based software engineering shows an expressive advantage: software components, besides being reused, can easily be adapted to match particular requirements. This, for the car industry, would mean having a single engine that could be adapted either to propel a limousine or a small city car.

Conceiving a system as an assemblage of adaptable components, however, brings about new challenges. To begin with, few software engineers would dare to suggest a definition for the term software component. Module, class, class category, object file, and server are just some of the constructs commonly referred to as component. Perhaps a dictionary definition, such as the one from the Oxford English Dictionary, which defines a component as “any of the parts of which something is made”, would be more adequate. Nevertheless, the lack of a clear definition for the term does not seem to prevent this technology from been successfully used. Other open questions have a more restrictive impact. For instance: What is a good size for a component? How can a component be adapted? How can it be glued together with other components? How to grant that a component composition yields a system that matches the requisites? Questions like these have to be answered in order to make component-based software engineering really effective.

The main concern of this paper is the adoption of component-based software engineering techniques to construct efficiently adaptable systems. The paper begins with some considerations about adaptable components, after what the SCENARIO ADAPTER construct is described in details. Scenario adapters can be used to efficiently adapt an existing component to join a specific execution scenario. Next a case study about the use of scenario adapters in the project EPOS is presented.

2. Adaptable Abstractions

Despite the controversy about what a component is (or should be), in this paper we will assume any plausible abstraction in the system's application domain to be a component. Considering an operating system, abstractions such as `thread` and `mailbox` could be our components, be them classes, servers, or any other kind of “part”. If an abstraction can be adjusted to satisfy the requisites of sev-

[*]This research has been partially supported by the Federal University of Santa Catarina, by CAPES Foundation grant no. BEX 1083/96-1, by Deutsche Forschungsgemeinschaft grant no. SCHR 603/1-2, and by the Bundesministerium für Bildung und Forschung grant no. 01 IS 903 D2.

eral execution scenarios, we say it is an adaptable abstraction. Based on these assumptions, a strategy to implement efficiently adaptable abstractions is described next.

2.1. Scenario Independence

The advantage of a system in which abstractions are independent from the execution scenario they run in is obvious: it can be adapted to join many scenarios. Whenever the system is requested to join a new scenario, some sort of adapter has to be implemented, but the basic abstractions remain untouched. Achieving this goal, however, is not a trivial matter, since abstractions tend to embed several scenario peculiarities.

The pollution of abstractions with scenario specific aspects happens naturally along the abstraction life-cycle. During the analysis phase, whenever the application domain is successfully partitioned, abstractions are assigned clear responsibilities that are completely independent from execution scenarios. These responsibilities are eventually translated into behavior and structure specifications during the design phase. If carried out properly, this phase can still yield scenario-independent abstractions, with dependencies isolated in proper constructs. However, the migration to the implementation phase tends to extend each abstraction's specification to satisfy the predicted execution scenarios, thus breaking down scenario-independence.

As an example, consider a `mailbox` abstraction in an operating system. From the analysis phase, we could have got that a `mailbox` is the abstraction responsible for supporting n-to-n, bidirectional communication among active objects. During design, this specification would probably have been extended to accommodate operations such as the sending and receiving of messages, as well as relations to constructs that will enable `mailbox` identification and location, message buffering, operation synchronization, etc. However, it would not be unusual if, along the implementation phase, questions regarding possible underlying networks, buffer management, security, and many others, had distorted the `mailbox` abstraction, transforming it into a scenario-specific abstraction.

This phenomenon has several implications on software quality metrics. First, it impacts reusability, since abstractions that are polluted with execution scenario details are seldom reusable in other scenarios. Very often different versions of an abstraction have to be implemented, one for each execution scenario. This impacts maintainability and increases the complexity of system configuration, since there are now several realizations for each abstraction. Therefore, it is very important to keep abstractions as independent from execution scenarios as possible.

Aspect-independence is also the main appeal behind *Aspect-Oriented Programming* (AOP) [5]. However, although AOP suggests means to adapt aspect-independent abstractions according to an aspect program, AOP itself does not enforce a design policy that yields aspect-independent abstractions.

2.2. Component Granularity

When talking about efficient adaptation of components, granularity becomes an important matter, since it directly impacts performance, configurability and maintainability. On the one hand, a system made up of a large amount of fine components will certainly achieve better performance than one made up of a couple of coarse components, because each coarse component brings along functionality that will not always be used. The component functionality that is not used often turns into overhead for the applications [6, 2]. On the other hand, a set of fine grain components is harder to configure and to maintain.

We do not propose components to have this or that size, but it is certainly more difficult to efficiently adapt a large component. Besides, extremely large components usually result from poor application domain partitioning during system analysis.

3. Scenario Adapters

Several alternatives to achieve scenario-independence have been proposed by the software engineering community, ranging from simple implementation constructs, to special languages and complete methodologies. Nevertheless, as adaptability is considered one of the "noble" qualities a software can have, paying a high price for it, specially in terms of performance, is usually acceptable. This condition often prevents adaptation techniques from being used in high performance computing system. Next we describe a low-overhead construct that can be used to efficiently adapt a scenario-independent abstraction to a given execution scenario: the SCENARIO ADAPTER.

The basic structure of a SCENARIO ADAPTER is depicted in figure 1. At a first glance, it may resemble the ADAPTER design pattern from [4]. There are, however, important differences. While the ADAPTER pattern suggests a polymorphic implementation, in which `Implementor` and `Scenario` are only bound at run-time, the SCENARIO ADAPTER is designed to be bound at compile-time.

The special care for static binding arises from the fact that many abstractions in a system have a single `Implementor`. In such cases, there is no sense in paying the high

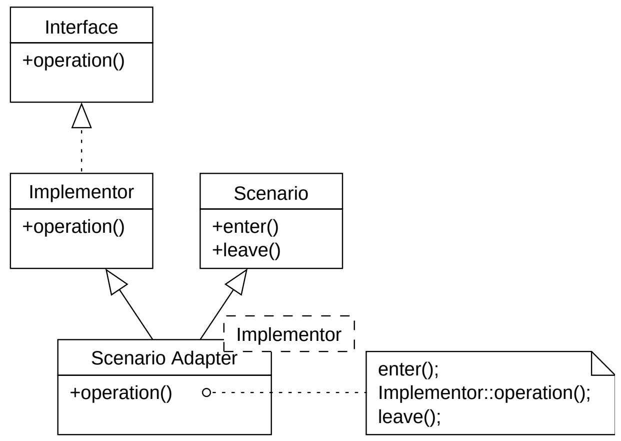

Figure 1: The basic organization of a SCENARIO ADAPTER.

price of polymorphism. Moreover, even if an abstraction have several `Implementors`, many low-level scenarios are mutually exclusive. For example, a well configured `thread` should exist either in the `multiprocessor` or in the `uniprocessor` scenario, since ordinary computers have either one or more than one processors when the system starts up and this situation is not expected to change during execution. In some other cases, abstractions refrain from changing execution scenarios at runtime just for the sake of performance. In this way, statically binding SCENARIO ADAPTERS usually does not imply in flexibility loss and yields quite better performing systems.

Each of the elements in figure 1 will now be described in details. For the sample code, we select the C++ language because it is the most accepted object-oriented language in the system software community. Care has been taken to consider language characteristics and not compiler-dependent aspects. Nevertheless, in order to be sure that the compiler is not misinterpreting some language definitions or generating inefficient code[1], some code generation checks may be convenient. We proceeded such checks for the GNU C++ compiler (egcs-1.1.2) with positive results, i.e., SCENARIO ADAPTERS did not incur in any overhead over scenario-independent abstractions.

Interface: exports the abstraction functionality and defines what an `Implementor` has to implement. Although a true interface declaration is missing in C++, it is possible to achieve a similar effect with a tricky class declaration. First, declaring the constructor `protected` avoids class instantiation and defining it to be empty cancels any influence over the realizations. Second, declaring all member functions to be private forces derived classes to implement them. The realize relationship between `Interface` and `Implementor` is then implemented via inheritance [2].

[1]Some compilers generate code (and the respective calls) for empty methods, including constructors.

An interface declared in this fashion is not as semantically strong as a pure abstract base class, since the compiler will not complain if an `Implementor` fails to implement an operation until it is instantiated. Nevertheless, it has the advantage of avoiding virtual function calls and run-time type information.

Sample code:

```
class Interface {
protected:
    Interface (int) {}
private:
   int operation(int);
};
```

Implementor: realizes the `Interface` in a scenario-independent fashion. There may be several `Implementors` for each `Interface`.

Sample code:

```
class Implementor: public Interface {
public:
   Implementor(int);
   int operation(int);
};
```

Scenario: gathers aspects that are common to all abstractions running in a scenario. Since all abstractions inherit the `Scenario` via the `Adapter`, it can also be used to decorate abstractions with scenario specific constructs. This could be useful, for example, to tag all abstractions with an authentication key in a secure scenario. A scenario has at least two methods: `enter` and `leave`. They are invoked by the `Adapter` respectively before and after each abstraction's method invocation. In the previous example about a secure scenario, `enter` would be responsible for authenticating all operations, while `leave` would probably be empty.

It is also usual for a `Scenario` to redefine common system methods with scenario-optimized versions, so that `Abstractions` can transparently access them. For example, a `Scenario` may redefine the `operator new` in order to optimize the memory allocation according to

[2]Note that implementing a realize relationship via inheritance, for all the compilers we had access to, has the undesirable side-effect of enlarging the resulting object by the size of an integer. This side-effect results from the C++ language definition that two pointers to two distinct objects may never have the same value, what is usually granted by having the compiler to assign a minimum size of 1 to any object. However, as the instantiation of the class `Interface` is prevented by declaring its constructor `protected`, having a zero size assigned to it would never break any language definition.

the current execution conditions.

Sample code:

```
class Scenario {
public:
   Scenario ();
   ~Scenario ();
   void enter();
   void leave();
};
```

Adapter: adapts an `Implementor` to to join a `Scenario`. It is implemented as a parametrized class (`template`) that inherits from the `Implementor` given as parameter. It defines all member functions declared in the `Interface` in such a way that `Implementor` operations are wrapped between the `enter`/`leave` pair. The `Adapter` is also the ideal place to carry out operations such as tracing, profiling, and *Remote Object Invocation*.

Sample code:

```
template <class Imp> class Adapter
 : public Scenario, public Imp {
public:
   Adapter(int i)
      : Scenario (), Imp(i) {}
   int operation(int i) {
      enter ();
      int ret = Imp:: operation (i );
      leave ();
      return ret;
   }
};
```

Abstraction: is the construct that will be instantiated by the system clients. We say that an abstraction has an `Interface` and one or more `Implementors`, and that it is adapted to a `Scenario` via an `Adapter`. It is implemented by the instantiation of the parametrized `Adapter` with an `Implementor`.

Sample code:

```
typedef Adapter<Implementor> Abstraction;
```

4. Discussion

The SCENARIO ADAPTER mechanism described above has proved to produce very efficient code. If the operations in `Scenario` and `Adapter` are declared `inline`, the result of an abstraction method invocation will be a direct call to the corresponding `Implementor`'s operation surrounded by the `enter` and `leave` primitives. Therefore, not only virtual function calls are avoided, but function calls at all. Another advantage of these statically metaprogrammed scenario adapters is that they can easily be optimized by the compiler: segments of the metaprogram that are not used are not included in the output.

A collection of SCENARIO ADAPTERS can be arranged to form a statically metaprogrammed framework [8]. Such a framework would define scenario-independent relationships among abstractions, letting open "holes" where IMPLEMENTORS can be plugged in.

5. The EPOS System

The project EPOS [3] aims at the construction of highly adaptable run-time system to support parallel computing on distributed memory machines, particularly clusters of workstations. In order to deliver each application a tailored operating system, EPOS takes on PURE [7] building blocks to implement a set of scenario-independent system abstractions that can be adapted to a given execution scenario with the aid of SCENARIO ADAPTERS. These abstractions are collected in a repository and are exported to the application programmers via INFLATED INTERFACES. This strategy, besides drastically reducing the number of exported abstractions, enables programmers to easily express their application's requirements regarding the operating system.

An application designed and implemented according to the strategy proposed by EPOS can be submitted to a tool that will proceed syntactical analysis to extract a blueprint for the operating system to be generated. The blueprint is then refined by dependency analysis against information about the execution scenario acquired from the user via visual tools. The outcome of this process is a set of keys that will support the compilation of an application-oriented operating system.

SCENARIO ADAPTERS are used in EPOS to define a metaprogrammed framework. System abstractions such as tasks, threads, communicators, synchronizers, memory, and peripherals have their interrelations defined in the framework, and can be arranged in systems for the following scenarios: kernel, library, local, remote, protected, single/multi-task, single/multi-thread. Several application-oriented systems have already been generated and evaluated, corroborating the efficiency of SCENARIO ADAPTERS.

6. Conclusion

In this paper we considered the utilization of component-based software engineering for the development of adaptable systems compromised with performance. The SCENARIO ADAPTER construct was proposed as an effective alternative to achieve this goal. The adoption of SCENARIO ADAPTERS in the Project EPOS helped to demonstrate the potentiality of this construct to support parallel applications running on cluster of workstations. Its use in deeply embedded system is now being considered.

The successful use of SCENARIO ADAPTERS, however, is a small achievement when one considers the amount of problems that are still to be solved in order to make component-based software engineering a leading software development strategy.

References

[1] James O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, 1999.

[2] Jörg Cordsen and Wolfgang Schröder-Preikschat. Object-Oriented Operating System Design and the Revival of Program Families. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, pages 24–28, Palo Alto, USA, October 1991.

[3] Antônio A. Fröhlich and Wolfgang Schröder-Preikschat. High Performance Application-Oriented Operating Systems – the EPOS Aproach. In *Proceedings of the 11th Symposium on Computer Architecture and High Performance Computing*, pages 3–9, Natal, Brazil, September 1999.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of ECOOP'97, Lecture Notes in Computer Science*, pages 220–242, Springer-Verlag, 1997.

[6] D. L. Parnas. On the Design and Development of Program Families. Technical Report BS I 75/2, TH Darmstadt, 1975.

[7] F. Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. Design Rationale of the PURE Object-Oriented Embedded Operating System. In *Proceedings of the International IFIP WG 10.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems*, Paderborn, Germany, October 1998.

[8] Todd Veldhuizen. Using C++ Template Metaprograms. *C++ Report*, 7(4):36–43, 1995.