

# Molding Components using Program Specialization Techniques\*

Gustavo Bobeff and Jacques Noyé

OBASCO Group – École des Mines de Nantes/INRIA  
4, rue Alfred Kastler - 44307 Nantes Cedex 3, France  
{Gustavo.Bobeff, Jacques.Noye}@emn.fr

**Abstract.** To our point of view, adaptability is a key characteristic of components and should be at the heart of any proper component model. However, contrarily to the object *crystal-box* model of reuse, which assumes full access to the object implementation, this adaptability should be strongly guided in order to keep the necessary decoupling between the component producer and its consumers, another key characteristic of components. It turns out that the current component models and infrastructures fall short of conciling these two characteristics and only provide a *superficial* form of adaptation. Taking full advantage of the notion of component requires a deeper form of adaptation, which calls for considering program specialization tools and techniques as key elements of the component-based software engineering toolbox.

## 1 Introduction

The adoption of the component as the unit of design and software construction, instead of an individual class, let us reason in terms of component composition when constructing applications. In this context, an application is conceived as a set of generic *ready to reuse* components connected to each other. But then how do components differ from, let us say, DLLs on the one hand and objects on the other hand? This is mainly a matter of *adaptability*. *Adaptability* refers to the ability of a piece of software to satisfy requirements dedicated to the context in which it is used.

The DLL model of reuse is a *black-box* model of reuse. The implementation of a DLL, provided as binary code, cannot be changed at all: there is no possibility of adapting a DLL to a specific use. On the other side of the spectrum, the reuse model of standard object-oriented languages is a *crystal-box* model of reuse. It relies on a very good visibility of the implementation of the units of reuse (classes). It is very powerful, as adaptability is almost unlimited, but also very fragile [20, 15]. Some approaches, like the use of design patterns [9], make it possible to avoid some of the pitfalls but at the cost of another layer of vocabulary and techniques and therefore additional complexity (and new pitfalls).

A key benefit of introducing a notion of component is to provide a balance between these two extreme visions of reuse. This requires introducing some shade of grey in the initial black-box model in order to provide a very strongly guided form of adaptation. In this paper, we suggest that a general way of doing so is to integrate program specialization techniques to the component development process.

This paper is organized as follows. Sect. 2 gives a basic definition of the notion of component taking adaptation into account and introduces a simple component model in line with this definition. Sect. 3 shows how program specialization can actually be applied in such a setting, without breaking strong encapsulation requirements. Sect. 4 shows how specialization can be performed efficiently through generative programming techniques, by introducing component generators. Sect. 5 concludes the paper.

## 2 Component Models

Starting from a black-box model, we can say that the *implementation* of a component (implementation refers here to program text, whatever the form of this program text, native code, bytecode, source code ...), provided by a component *producer* is essentially a black box, out of reach of the component

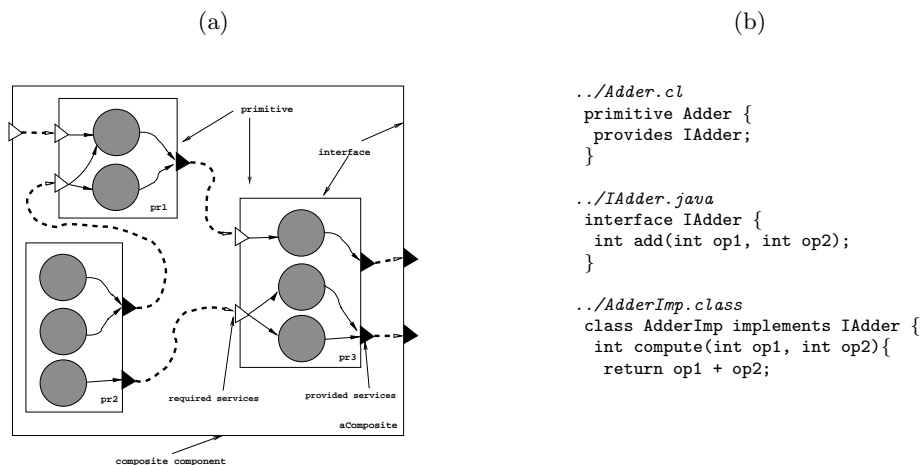
---

\* This work was partially funded by the European Commission in the FET Open Domain of the IST Programme under contract no. IST-1999-14191 (EASYCOMP - Easy Composition in Future Generation Component Systems). Any opinions, findings, and conclusions or recommendations expressed in this material are the authors and not necessarily reflect the view of the sponsors.

*consumer*. However, in order for the component consumer to use a component in a composition, an *interface* is provided by the producer. The role of this interface is to guide composition (rejecting, for instance, incorrect compositions) from two main points of view: a structural point of view (a matter of interconnections) and from a behavioral point of view (a matter of interactions) (see, for instance, [19, 8, 23]). But, on top of this syntactic role, this interface has also a semantic role, being transformed at composition time (at least conceptually) into a *wrapper*, encapsulating the *core* of the component (the result of the transformation of the implementation). Whereas the implementation is a black box, the interface is a *white box*. It is not a crystal box as it is neither as transparent nor as fragile, but through parameterization facilities, it is possible to change the component and adapt it to the composition context as proposed in [17].

To keep things simple, we have distinguished here two phases only in the life of a component: its production and its use in a composition. Note that a component can also be *instantiated* (talking about a *component instance* would then be more precise) and finally *executed*. Also, component composition can be incremental and take place either at compile time (the component may then still need to be *deployed*) or at run time. Depending on the details of the model, the interface can be turned into a wrapper at a different time than composition and this transformation could even be incremental.

This general model is actually compatible with all the industrial component infrastructures such as the Enterprise JavaBeans (EJB) or COM+. In particular, it makes it possible to adapt components with respect to predefined technical services (persistence, security ...). What is however disappointing is that, in this model, there is no adaptation action on the implementation of the component. Along the different phases of the component life many pieces of adaptation information are fed into the interface/wrapper, but this has no effect on the implementation/core. It is however possible to go one step further using program specialization.

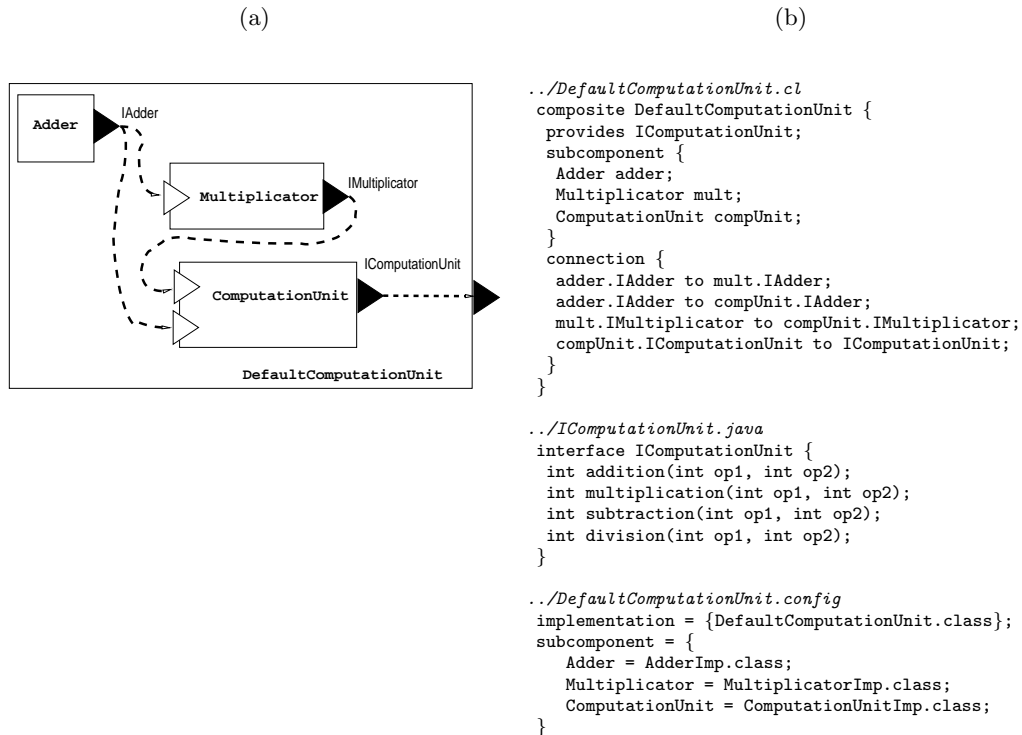


**Fig. 1.** Component Model: Structure.

For the sake of this discussion, let us refine this general model into a simple component model. This simple model presents similarities with a large number of the existing models used by the “connection-oriented” languages [14, 21, 1, 5, 7], but the discussion of these similarities is outside the scope of this paper. We distinguish two main parts in the component definition, the *interface* and the *implementation*. Fig. 1(a) depicts these features representing the component interface as rectangles and the implementation as filled circles inside. The interface comprises *required* (empty triangles) and *provided* (filled triangles) services. The implementation corresponds to the code block associated to each provided service, which may refer to the required services. In the figure, the arrow from a required service to an implementation block means that this block uses this service, whereas the arrow from an implementation block to a provided service means that this block actually represents the implementation of the service. Component composition is expressed through explicit connections from required to provided services in the components (dashed arrows in the figure). The linking expressions belong to the enclosing composite component definition and the enclosed components are called *subcomponents*. We distinguish *primitive* components (e.g. pr1, pr2 or pr3), constructed with an *implementation language*, and *composite* com-

ponents (e.g. `aComposite`), built up by composing both primitive and composite components. In terms of use, a component consumer does not observe any difference between a primitive and a composite component.

In fig. 1(b), we describe the component `Adder` by using a simple component language (CL). In this example, the `Adder` component provides a set of services described through the `IAdder` interface. In CL, an interface permits to classify a set of semantically related service definitions (e.g. `int add(int op1, int op2)`) behind an identifier as it can be found, for instance, in Java. For the sake of simplicity, we use Java as the single implementation language of the primitives components. One possible implementation of the component `Adder` may be `AdderImp` as described in fig. 1(b). The association between a component interface and its implementation is done separately (e.g. through a `.config` file) to decouple the implementation from the interface.



**Fig. 2.** Component Model: Component Composition.

Fig. 2(a) shows the `DefaultComputationUnit` composite component constructed by the composition of the `Adder`, `Multiplier` and `ComputationUnit` primitive components. The files associated to this component application are shown in fig. 2(b). `DefaultComputationUnit.cl` contains the component definition using the component language mentioned above, `IComputationUnit.java` defines the interface provided, and finally, the `.config` file links the component definition and its implementation. We shall use this component definition to exemplify the component specialization process in the following sections.

### 3 Component Specialization

**Specialization Opportunities:** Program specialization techniques make it possible to automatically transform a program into a specialized version, according to an execution context. Let us introduce two well-proven program specialization techniques *partial evaluation* [12] and *program slicing* [24, 16]:

**Partial Evaluation** A partial evaluator is a program that transforms a program by compiling away the computations based on *static* (i.e. known) information and reconstructing the remaining *dynamic* computations to form the specialized program. This technique has been studied mainly in a functional, logical, or imperative setting, but has also lately been applied to the object paradigm [3, 18].

**Program Slicing** It is a method for automatically decomposing programs by analyzing their data and control flow. Starting from a subset of the behavior of a program, slicing reduces the program to a minimal form that still produces the behavior. For instance, this technique can be applied to determine the statements affecting a particular output statement of a program. Then, if this output statement is no more useful, these statements can be eliminated leading to another kind of specialized program. Initially applied to imperative and functional languages, some work has also been started on object-oriented languages [22].

**Specialization Scenarios:** Currently, program specializers, particularly partial evaluators [6, 18], require from the user a deep knowledge their internal functioning. For this reason, we follow the proposal of Le Meur et al [13], whereby the definition of specialization opportunities are grouped in a declarative and modular way into *specialization scenarios*. A declarative approach improves component reuse since the specialization constraints are also visible to the component consumer. Besides its interface, the component consumer knows more about the component, emphasizing in this way the gray-box abstraction [4]. A modular approach allows the component producer to keep control of the specialization process performed on the components and it is also possible to reuse specialization scenarios by combining them.

The specialization opportunities within the scenarios are defined by the producer during the construction of the component. In our work, we use an approach based on constraints to define such scenarios. A specialization constraint (constraint for short) will express a specialization opportunity according to one of the two techniques previously described. Such a constraint therefore deals with dependencies between the service input and output parameters. Partial evaluation requires to describe the forward dependencies between the input and the output *binding time* whereas slicing requires to describe the backward dependencies between the output and the input *usefulness*. Basically, the *binding time* of an expression tells whether the expression can be computed at specialization time or not. The expression is said to be *static* in the first case, *dynamic* in the latter. The *usefulness* of an expression tells whether its computation is useful or useless. In the following, we will annotate static, dynamic, useful, and useless expressions as *S*, *D*, *F*, and *L*, respectively.

Both techniques, partial evaluation and slicing, involve code analysis that is not a simple task. It is hard to do manually and error-prone as well. Such an analysis is however necessary in order to identify specialization opportunities in order to create the scenarios. For this reason, we assume that the component producer is provided with a proper tool to perform such an analysis, the *analyzer*.

Then, it is up to the component producer to propose constraints on provided and requires services to begin the construction of the specialization scenarios. This *initial* set of constraints and the component implementation constitute the input of the analyzer. From such an analysis, the component producer acquires information about the *viability* of a constraint and *derived* constraints as well. In the former case, the analyzer should be able to know about the benefits obtained, or not, when the specialization is performed, avoiding some well-known drawback in program specialization (e.g. *over-specialization* and *under-specialization*). In the latter case, extra constraints are computed from the propagation of the initial constraints through the component implementation.

**Partially Evaluating Components:** Without explaining in details the different steps that take place during partial evaluation, we may say that, given a program and the binding-time information of its parameters, it is the analyzer that determines which parts of the program are static (e.g. annotated as *S*), and which are dynamic (e.g. annotated as *D*).

In a component setting, it is possible to guide the component specialization by relating the binding-times information to the provided and required services. Fig. 3(a) shows the specialization scenarios on the **Adder** primitive component. One important issue here is that the specialization opportunities are related to one particular implementation, in this case **AdderImp** (see fig. 1(b)). In our example, all consistent alternatives have been enumerated, but with respect to the implementation of the service **add** it makes sense to consider only the scenario where both parameters and the resulting value are static. In the other cases there is not any specialization opportunity. The syntax we use here for primitive constraints is a service signature where type information has been replaced by binding-time information, and the interface name the service belongs to is included to avoid service name clash in the constraints. This information will be used by the specialization program, the *specializer*, to specialize the service. In the example, we consider a set of values **op1=2** and **op2=3** that represents a concrete usage context for the specialized service **add\$op1\_2\$op2\_2()** as shown in fig. 3(b).

(a)	(b)
<pre> <i>S</i> IAdder.add(<i>S</i>,<i>S</i>); <i>D</i> IAdder.add(<i>D</i>,<i>S</i>); <i>D</i> IAdder.add(<i>D</i>,<i>D</i>); </pre>	<pre> Specialization process: 1) target: AdderImp.class 2) scenario(satisfied): <i>S</i> IAdder.add(<i>S</i>,<i>S</i>); 3) context: op1=2; op2=3; 4) result:    public int add\$op1_2\$op2_2(){return 5;} </pre>

**Fig. 3.** Specialization of AdderImp.

Each component including the `Adder` primitive component as subcomponent, and selecting `AdderImp` as its implementation, should be able to use the specialized version of the services provided by `AdderImp` when its specialization scenario is satisfied.

(a)	(b)
<pre> ../IMultiplier.java interface IMultiplier {   int compute(int op1, int op2); }  ../MultiplierImp.java Class MultiplierImp implements IMultiplier {   ...   public int multiply(int op1, int op2) {     int result = 0;     if (op1 &gt;= 0) {       for (int i = 1; i &lt;= op1; i++)         result = adder.add(result, op2);     } else if (op2 &gt;= 0) {       for (int i = 1; i &lt;= op2; i++)         result = adder.add(result, op1);     } else {       int _op1 = Math.abs(op1); int _op2 = Math.abs(op2);       for (int i = 1; i &lt;= _op1; i++)         result = adder.add(result, _op2);     }     return result;   } } </pre>	<pre> <i>S</i> IMultiplier.multiply(<i>S</i>,<i>S</i>); <i>D</i> IMultiplier.multiply(<i>S</i>,<i>D</i>); <i>D</i> IMultiplier.multiply(<i>D</i>,<i>S</i>); </pre>
	(c)
	<pre> Specialization process: 1) target: MultiplierImp.java 2) scenario(satisfied): <i>D</i> compute(<i>S</i>,<i>D</i>); 3) context: op1=3; 4) result:    public int multiply\$op1_3(int op2){      int result = 0;      /* unfolded loop      result_1 = adder.add(result, op2);      result_2 = adder.add(result_1, op2);      result_3 = adder.add(result_2, op2);      return result_3;    } </pre>

**Fig. 4.** Specialization of MultiplierImp.

Now, let us take a look at the implementation of `MultiplierImp`, associated to the component `Multiplier`, as shown in fig. 4(a). It seems reasonable to define some specialization scenarios for the `multiply` service when `op1` and `op2` are bound to a static and a dynamic value, respectively. Based on this information the specializer eliminates the conditional structure, unfolds the loop, etc. The set of scenarios is listed in fig. 4(b). Fig. 4(c) shows the specialization of the `multiply` service in the `Multiplier` component implementation.

Unlike the `Adder` component, `Multiplier` requires `IAdder` services. The producer of `MultiplierImp` has not access to the implementation which will provide these services (in our case `AdderImp`) in a determined component composition, for example, as shown in fig. 2. But, even if it is not possible to reason in terms of a concrete specialization scenario defined for `AdderImp`, it is possible to make hypotheses about the specialization scenario associated to the required services. For example, let us consider the following hypothetical rules:

```

S IAdder.add(S,S) -> S IMultiplier.multiply(S,S)
D IAdder.add(S,S) -> D IMultiplier.multiply(S,S)

```

The first rule says that if the component providing the service `IAdder.add` includes the constraint `S IAdder.add(S,S)`, then the service `IMultiplier.multiply` can be specialized according to the

constraint  $S$  `IMultiplier.multiply(S, S)`. Otherwise, if nothing is known about the return type of `IAdder.add`, then the service `IMultiplier.multiply` can be specialized according to the constraint  $D$  `IMultiplier.multiply(S, S)`, as it is expressed in the second rule. Both rules are added to the initial set of constraints described in fig. 4(b).

**Slicing Components:** In many cases, a generic component provides more services than required by a given composition. Then, the unused services can be sliced away according to the corresponding constraints. Let us consider the `ComputationUnit` primitive component included as a subcomponent of the `DefaultCalculationUnit` composite component shown in fig. 2(a). One possible implementation of this primitive component is given in fig. 5(a). An analysis of the dependencies between the provided and required services in `ComputationUnitImpl` gives the possible specialization opportunities. A dependency graph is depicted in fig. 5(b), where an arrow means that the source *depends* on the target. Since no service depends on the `division` service, this service can be removed. The following constraint expresses such an opportunity: `L IComputationUnit.division(L, L)`.

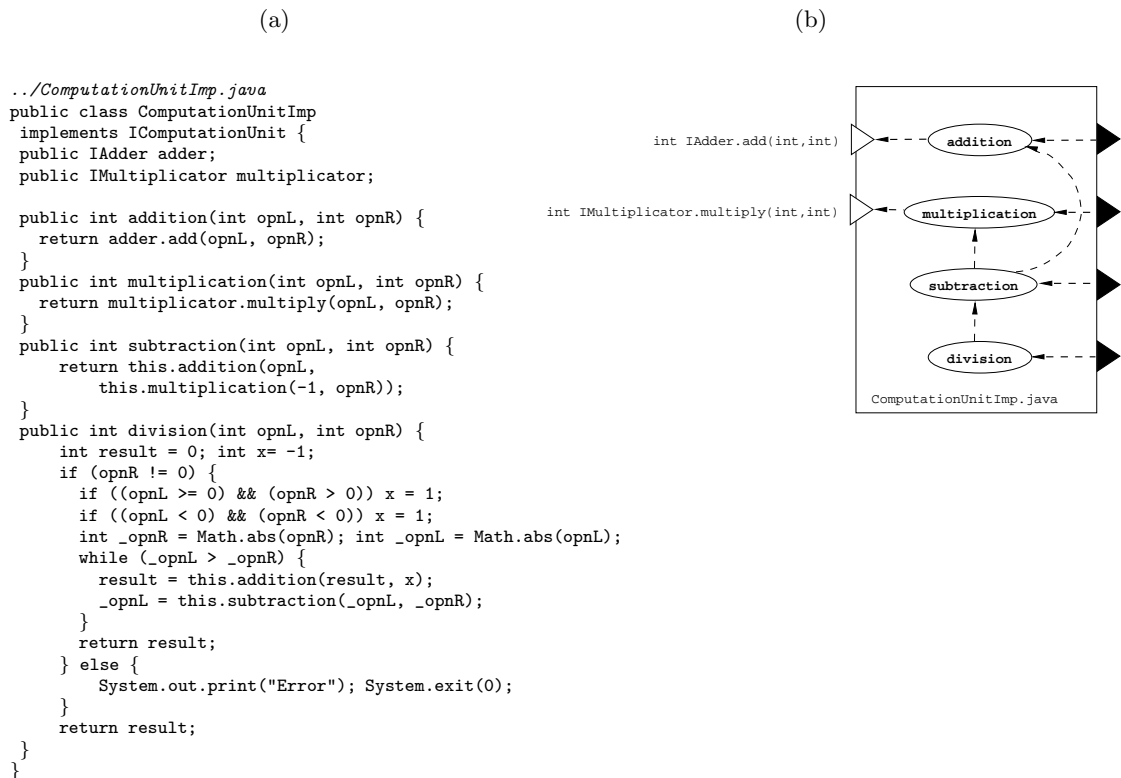


Fig. 5. Service dependencies in `ComputationUnitImp`.

Now, let us consider the component `ImageFilter` depicted in fig. 6(a). Fig. 6(b) shows `IFConvolutionImp`, an implementation of the `ImageFilter` primitive component which applies a mask to each point of an image. A look at the code of `applyMask(int x, int y)` shows that only the `addition` and `multiplication` services provided by `DefaultComputationUnit` are used. Therefore, the following set of constraints can be asserted:

```

L IComputationUnit.division(L, L)
L IComputationUnit.subtraction(L, L)

```

As with partial evaluation, the specialization constraints can be propagated, in our example from `IFConvolutionImp (ImageFilter)` to `ComputationUnitImpl (ComputationUnit)`, so that the uselessness of required services in `IFConvolutionImp` entails the specialization of `ComputationUnitImpl`.

Of course, slicing may also affect only part of a service.

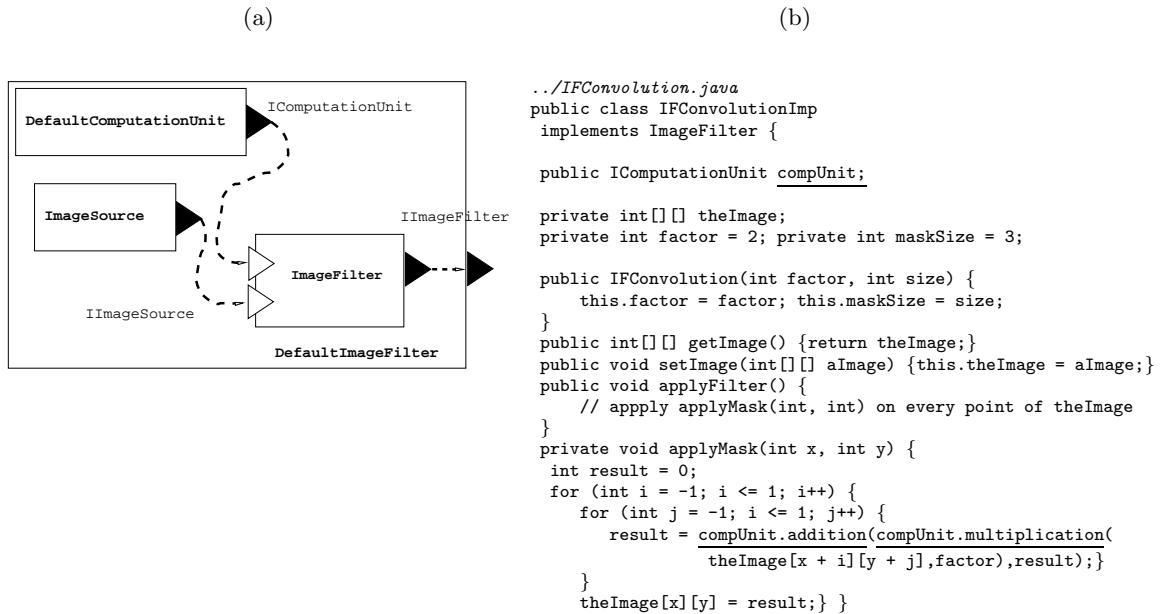


Fig. 6. Component Specialization by slicing services.

## 4 From Components to Generators

Packaging components with valid specialization scenarios is key to making component specialization compatible with a black-box model.

Let us first assume a simple case where all the components are implemented using the same language, delivered as source code (high-level bytecode could also do). We also assume that each developer has access to the same standard analyzer and specializer. Then, the assembly can be analyzed and specialized as a whole, with the analysis guided by the specialization scenarios. The main point here is that the component consumer does not need to care about the details of the analysis, which simply replays the analysis performed at production time to build and validate the specialization scenario. Of course, some assembly could be rejected because of some scenario incompatibilities, but this would relate to component interfaces available to the component consumer.

This first scenario can be refined and its hypotheses relaxed in the following ways.

As a first step, each packaged component could include one (or several) annotated version(s) of the implementation, corresponding to the specialization scenarios (the analyzer is no longer needed) as well as a specific specializer. The problem is that this makes packaged components heavy-weight.

As a second step, a component could be packaged as a *component generator*. Similarly to the *generating extension* technique, developed in the partial evaluation community [12, 2], the structure of the component generator would express the specialization opportunities present into the initial component definition. It can be seen as a specialization of the generic specializer mentioned above with respect to the concrete component implementation and the specialization scenarios. One issue could be here, in the case of a large number of scenarios, to find ways of properly structuring the generator using some forms of polymorphism. Resorting to a *polymorphic binding-time analysis* [10] could be of help here.

Note that, with component generators, components can be packaged as binary code. Also, component assembly is turned into component generator assembly. This is the collaboration of these generators which builds the specialized composition.

## 5 Conclusion

We have shown that adaptability is a key component feature and proposed a basic component definition taking this feature into account. This has hopefully made clear that component adaptation brings many optimization opportunities but that taking advantage of the opportunities associated to the component core calls for sophisticated program specialization techniques. We have further illustrated these ideas on

a simple component model, considering dual specialization techniques, partial evaluation and slicing. A key to not breaking encapsulation is then to use specialization scenarios, as introduced, in the context of C, by Le Meur et al. [13]. This can be complemented with the use of component generators in order to further decouple component production and use. We are currently working on a component model, language, and infrastructure based on these ideas.

## References

1. J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting software architecture to implementation. In ICSE2002 [11].
2. Lars Birkedal and Morten Welinder. Hand-writing program generator generators. In Manuel V. Hermenegildo and Jaan Penjam, editors, *PLILP*, volume 844 of *LNCS*, pages 198–214. Springer, 1994.
3. M. Braux and J. Noyé. Towards partially evaluating reflection in Java. In *2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'00)*, pages 2–11, Boston, MA, USA, January 2000. ACM Press.
4. M. Büchi and W. Weck. The greybox approach: When blackbox specifications hide too much. Technical Report 297a, Turku Centre for Computer Science, Turku, August 1999.
5. R. Cardone, D. Batory, and C. Lin. Java layers: Extending Java to support component-based programming. Technical Report 00-11, Department of Computing Science, University of Texas at Austin, June 2000.
6. C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, and E.N. Volanschi. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys*, 30(3), September 1998.
7. J. Costa Seco and L. Caires. Parametric typed components. In *Proc. of the Fifth Intl. Workshop on Component-Oriented Programming ECOOP'00*, Sophia Antipolis and Cannes, France, June 2000. Springer-Verlag.
8. B. Councill and G.T. Heineman. Definition of a software component and its elements. In G.T. Heineman and W.T. Councill, editors, *Component-Based Software Engineering – Putting the Pieces Together*, pages 5–19. Addison-Wesley, 2001.
9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
10. F. Henglein and C. Mossin. Polymorphic binding-time analysis. In D. Sannella, editor, *Programming Languages and Systems - ESOP'94 - Fifth European Symposium on Programming*, volume 788 of *LNCS*, pages 287–301, Edinburgh, UK, April 1994. Springer-Verlag.
11. *Proc. of the 24th Intl. Conf. on Software Engineering*, Orlando, FL, USA, May 2002. ACM Press.
12. N.D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, September 1996.
13. A. Le Meur, C. Consel, and B. Escrig. An environment for building customizable software components. In *IFIP/ACM Working Conference - Component Deployment*, pages 1–14, Berlin, Germany, June 2002. Springer-Verlag.
14. S. McDirmid, M. Flatt, and W.C. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *OOPSLA '01, Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 2001.
15. L. Mikhajlov and E. Sekerinski. A study of the fragile base class. In E. Jul, editor, *ECOOP'98 - Object-Oriented Programming - 12th European Conference*, volume 1445 of *LNCS*, pages 355–382, Brussels, Belgium, July 1998.
16. T. Reps and T. Turnidge. Program specialization via program slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, volume 1110 of *LNCS*, pages 409–429. Springer-Verlag, February 1996.
17. Ralf H. Reussner. The use of parameterised contracts for architecting systems with software components. In J. Bosch, C. Szyperski, and W. Weck, editors, *Eighth International Workshop on Component-Oriented Programming*, Darmstadt, Germany, July 2003. In conjunction with ECOOP 2003.
18. U.P. Schultz. *Object-Oriented Software Engineering Using Partial Evaluation*. PhD thesis, Université de Rennes I, December 2000.
19. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.
20. A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 38–45, Portland, OR, USA, 1986. ACM Press.
21. V. Sreedhar. Mixin'up components. In ICSE2002 [11], pages 198–207.
22. C. Steindl. Intermodular slicing of object-oriented programs. In *Proceedings of the 8th Workshop for PhD Students in Object-Oriented Systems ECOOP'98*, Brussels, Belgium, July 1998. Springer-Verlag.
23. C. Szyperski. *Component Software*. Addison-Wesley, 2002. 2nd edition.
24. F. Tip. A survey of program slicing techniques. Technical Report CS-R9438, CWI, 1994.