**Technical Report No: 2002/09**

# *Beyond Inheritance, Aspects & Roles: A Unified Scheme for Object and Program Composition*

***Henrik S Gedenryd***

*2002*

---

***Department of Computing***
**Faculty of Mathematics and Computing**
**The Open University**
**Walton Hall,**
**Milton Keynes**
**MK7 6AA**
**United Kingdom**

***http://computing.open.ac.uk***

# Beyond Inheritance, Aspects and Roles: a Unified Scheme for Object and Program Composition

Henrik Gedenryd
The Open University
Milton Keynes MK7 6AA, UK
+44–1908–659 542
h.gedenryd@open.ac.uk

## ABSTRACT

The areas of inheritance, aspect-oriented programming and role-based decomposition share the same problem: For all three, the number of candidate schemes is large, all of them different and none of them clearly superior to the rest. Instead of proposing another variation on any of them, this paper presents a simple, unified approach to program composition. The scheme is shown to be compositionally complete, that is, to be sufficient for defining any program composition that is theoretically possible, and therefore forms a superset of all other approaches to program composition. The paper shows how this scheme specifically may supersede inheritance, aspects, and roles. It goes on to show via examples how the scheme can be used as a practical object-oriented language construct. Lastly, it demonstrates how the scheme can be combined with program specialization to yield very good runtime performance. This scheme can make object-oriented languages smaller, yet substantially more powerful and expressive than they currently are.

## 1. MOTIVATION

The topics of inheritance, aspect-oriented programming and role-based design each have a huge literature devoted to them. From the early days, inheritance has been known to be imperfect [24]; its various problems are well documented [9, 14, 37], and a large number of variations [8, 9, 25, 35] and alternatives [18, 24, 32, 40] have been proposed. However, rather than offering a resolution or identifying a winning candidate, the various approaches have diverged so that no clearly superior alternative is available today, especially not if simplicity is desired.

In aspect-oriented programming (AOP) there are also a number of different solutions to similar but not identical problems [11, 15, 27, 28, 31, 39]. And beyond this divergence, there is a lack of well-defined foundations. The central concepts of AOP, such as "aspects" or "crosscutting", have not been well articulated, and the formal basis of AOP seems not to have been addressed: Just what is it that AOP allows us to do that we otherwise cannot do, and what added formal powers do AOP extensions bring to a language? In contrast, general-purpose programming language constructs usually have well-defined semantics and formal foundations. These are the properties whose absence tends to prevent specialist techniques from being adopted as dependable and widely applicable solutions.

Finally, role-based design and problem decomposition are probably as old as object-oriented programming itself, even though the technique has only been gradually articulated over the years [2, 5, 19, 41]. However, roles typically lack language support and their application therefore requires compromises or the use of roundabout techniques [26, 41]. It is also worth noting the great family resemblance of the various techniques surrounding roles, aspects, perspectives [6, 9, 18, 24], subjects [34, 36], and so on. So also here, like in the two previous areas, the divergence of approaches and lack of a clear resolution is holding back widespread adoption of any solution.

### 1.1. A proposed resolution

The object of this paper is not to present yet another variant of any of these schemes. Instead, the key idea behind it is to take a step back to recognize that all of the above techniques are different approaches to composing programs and objects. What if we try to address the problem of program composition in general, once and for all? If we found such a scheme, it would be able to supersede all of the above approaches. The answer to this question is the actual topic of this paper—a simplest, yet fully comprehensive approach to composing objects and programs.

Still, the purpose is not to present a highly sophisticated and complex solution to a very hard problem. On the contrary, the aim is to show that the problem is much less difficult than it might seem. To obtain a general but also simple solution, the chosen strategy was to reduce the necessary ingredients to their essence, and then to allow them to be applied in the most general way possible. A beautiful example of this approach was the demonstration that only two language constructs (*if…then* and *while…do*) are needed for describing any Turing machine [7]. This demonstration was essential to the development of structured programming, and it shows how very simple but sufficiently general elements can be used to generate very complex results. Moreover, it also shows that simplicity itself is the key to achieving this generality. Clearly, this work set a standard that the present work could merely aspire to attain.

The following section introduces the view of composition that is taken here, and shows how inheritance and aspect-oriented programming can be expressed as program composition. It thereby lays the groundwork for the demonstration of compositional completeness that follows it. The next section describes the practical programming approach that results from the theoretical principles, and this is also where role-based design enters the picture. The remainder of the paper gives some example

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*Conference '00*, Month 1-2, 2000, City, State.
Copyright 2000 ACM 1-58113-000-0/00/0000…$5.00.

applications of the resulting approach. The final example addresses an extremely performance-sensitive application. It illustrates how this scheme can be used to obtain very well-structured designs and highly decomposed implementations without sacrificing runtime performance.

# 2. Composing programs by assembling parts into new wholes

The view of objects and programs as compositions is essential for the demonstration of compositional completeness. The aim of the present section is to make the reader accustomed to this point of view before taking on the more theoretical discussion.

## 2.1. The Concept of Composition

Since the term composition is so central to this effort, it deserves to be clarified: The *act of composition* is the building of complex entities by assembling distinct part entities to form new whole entities. The *composition* of an entity is synonymous with the *structure* of an entity, and refers to the part–whole relationships between an entity and its sub-elements. Conversely, *decomposition* refers to breaking a whole down into parts. Compositions are conventionally drawn as box-and-arrow diagrams, where "whole" boxes have arrows pointing to their part boxes, or alternatively as with trees, use plain lines instead of arrows whenever parts are always placed below their wholes (figure 1).

There also ought to be a single name for the elements that are composed; however, all the good names have already been claimed for various purposes. The best candidate would otherwise be *component*, as the meaning of this term is simply "an element in a composition". Instead I will mostly use the term *part*, as it is succinct and clear—you build things out of parts—and not heavily associated with any specific technical meaning.

Whereas composition concerns the structural view of software, the complementary view concerns *content*—without it, any composition is just an empty structure with no capabilities, and conversely, the composition puts together trivial content primitives into more complex, higher-level capabilities. The content primitives of software are state (data) and behavior (code) primitives. The distinction between structure and content is the same as the one between syntax and semantics.

This paper will focus on the composition of objects, which easily generalizes into the composition of programs in general. In pure OO languages the reason is straightforward: since everything is an object, general object composition is sufficient to compose every element of a program, including modules, name spaces, etc. In non-pure OO languages such as Java and C++, the same principles will have to be repeated for the non-object language constructs.

The fundamental form of object composition is that an object refers to other objects via its instance variables, or *slots* following Self terminology [40]. This is how you define part–whole relations between objects, and in this particular respect objects are similar to *records* in non-object-oriented languages. This type of composition uses the most well-known composition principle for software, hierarchical composition, which allows the building of tree structures. Such structures can be described by context-free grammars [30], which consist of a number of rules of the form $Whole \rightarrow Part_1, Part_2, \ldots Part_N$.

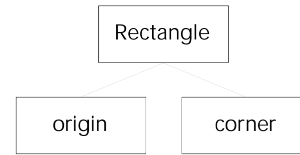


**Figure 1. A basic part–whole composition: an object with two slots.**

## 2.2. Inheritance as Composition

The next step to understanding object composition is to reinterpret inheritance as a form of composition. The contents of any class (state and behavior) are composed from the contents of its superclasses, i.e. the elements in its inheritance graph. However, inheritance relations are not normally drawn as the inheritance graph of a single class, but as class hierarchies, which show how several classes are related to each other by inheritance (figure 2a).From this class hierarchy we need to extract the inheritance graph of a single class. In the case of single inheritance, this is not a tree, but always a linear chain connecting the class and its superclasses. Furthermore, to follow the convention of always having the root of a composition at the top of the diagram, we need to turn this graph on its head (figure 2b). This view implies that in some sense the superclasses should be regarded as parts of the derived subclass. This finds support in a version of the same diagram where the objects' memory layouts are explicitly drawn (figure 2c). This version shows that the *super* slots contain the actual links from subclass to superclass (also, dashes indicate that some parts have been left out).

Since we are interested in the composition of objects, not just classes, figure 3a adds the actual composed object at the top of the composition diagram. It also makes the class field explicit, to show that this slot is no different from other slots from a compositional point of view, even though it is typically treated differently (for good reason). Figure 3c displays the same composition as figure 3c, but no longer in the memory-layout format. The dashed outline marks the most important parts of the object's composition. At this point, the inheritance graph has been "normalized" so that the special semantics of inheritance that is unrelated to composition is ignored, and only the part–whole

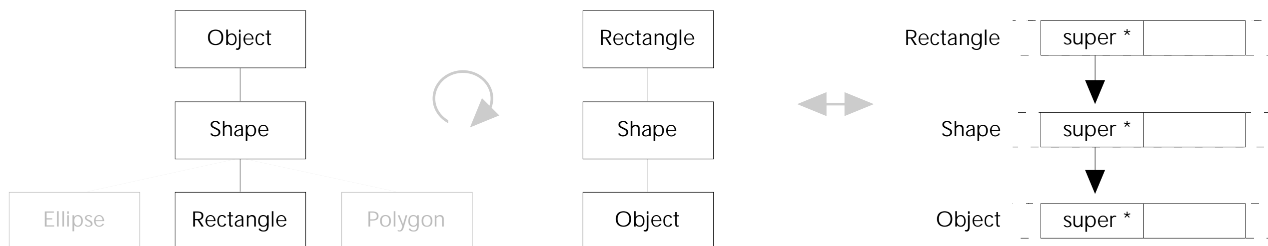


**Figure 2. Inheritance as composition I: (a) Class hierarchy. (b) Inheritance graph. (c) Object format.**

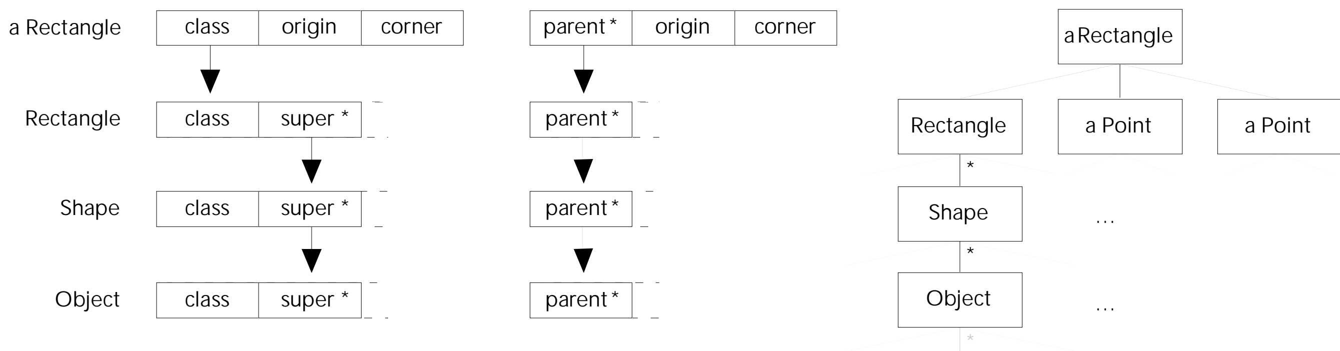


**Figure 3. Inheritance as composition II: (a) Smalltalk object layout. (b) Self-like version of 3a. (c) Composition diagram for the same object. Each figure shows a partial view of the composition graph.**

compositions of object, class and superclasses are expressed in the diagram. That is, the class and superclasses are drawn as any other elements in the composition of the object. As a result, figure 3c represents the object purely in terms of parts and subparts; this is the purely compositional view of the object and its inheritance.

*2.2.1. De-encapsulation*

The remaining element to explain is the asterisks that mark the superclasses in these diagrams. This is the same convention that is used in Self, for designating "parent slots" or "shared parts" [10]. Although the effect is the same, here its meaning will be slightly generalized, to indicate *de-encapsulation* of the marked entity. Since encapsulation is essential to object-oriented composition, the regular part objects (as referenced by instance variables, slots, etc.) do not share their contents with the owning object they are part of—they encapsulate their contents. In Smalltalk, for example, the owner object cannot access the contents of its instance variables except by ordinary message passing. Similarly, in Self the messages defined by a part object cannot be accessed by sending messages to the owning "whole" object, as they are encapsulated. However, a parent designation in Self means that messages defined by the part object will be shared with the whole object. That is, the parent or sharing designation de-encapsulates the messages, so that they will work as though they were part of the whole object itself. (Overriding will be disregarded since it is less relevant here.) Inheritance, as for example in Smalltalk, has a similar de-encapsulating effect [37], firstly because a subclass can directly read and write the variables defined by its superclasses, and secondly because messages defined in superclasses also work as though they were defined in the subclass (again ignoring overriding). So the difference between regular variables and the superclass in Smalltalk is the same as between regular slots and parent slots in Self. From the point of view of object composition, if we would distill the property that distinguishes inheritance/delegation from regular part–whole composition down to its purest form, the remaining element is de-encapsulation. (For example, the principle of overriding is a practical consequence of this principle.) Accordingly, in the above diagrams the *super* slots are marked with an asterisk to indicate de-encapsulation.

*2.2.2. The similarity to Self*

There are now striking similarities between figure 3c and how the inheritance/delegation model of Self is usually represented [10, 40]. If the same organization would be used in Self (which is not how it would be done in true Self style), the resulting diagram would look like figure 3b. That is to say, the composition would be exactly the same, beneath Self's slightly different semantics and terminology. This is no coincidence: Self radically regularized the previously quite disparate mechanisms for object-oriented composition when it unified parents with regular slots, and also enabled any slot to be marked as a parent, not just the *super* slot of Behavior objects, as in Smalltalk. For these reasons, the Self language went further than any other language toward the scheme that is presented in this paper.

In the interest of brevity and simplicity, delegation will be subsumed under the general umbrella of inheritance in what follows, as it concerns the same general type of composition [10, 38].

## 2.3. Aspect-oriented programming as composition

Within aspect-oriented programming, the aspect concept itself has undergone a gradual transition from the early days to the present day. In the beginning, aspects and concerns (as in "separation of concerns") were not thought of in terms of composition or structure, but rather as certain domains of functionality [28, 33]. Firstly, the prototypical examples of aspects were all related to a certain functionality: synchronization, distribution, exception handling, and so on. Secondly, the general approach was to address each such domain with a separate, domain-specific aspect language (ibid.). Thirdly, when the aspect concept was introduced, it was defined in terms of functionality, as "units of system decomposition that are not functional" [28]. Also, a crucial distinction was made between the basic functionality of a program, which could be decomposed by existing language constructs, and "non-functional" concerns that had to be addressed by aspectual decomposition. At that point, the AOP field had not yet, as it were, separated the *composition aspect* of software from the *domain* or *content aspect*.

There is further evidence of this gradual shift from the initial domain-oriented accounts, over the just quoted mixture of functional and compositional terminology, to a more purely compositional view of aspects in recent times, as displayed e.g. in [15]. For example, the earliest versions of AspectJ had domain-specific sub-languages, whereas more recent versions have taken a independent and content-agnostic approach. To follow these developments to their logical conclusion, the position taken here is that aspects are a purely compositional and content-agnostic concept. Or in other words, aspect-oriented programming is all about *aspectual composition*.

Figure 4 shows the composition graph for an example application of AOP, which illustrates the essence of aspectual composition. In this example, the basic functionality of two classes have had an

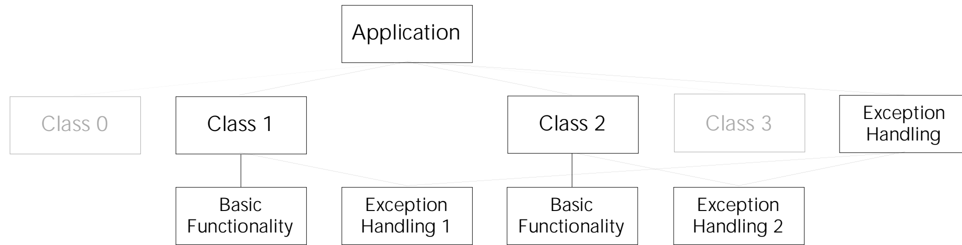


**Figure 4. The composition graph for a simple case of aspectual composition. Note how the composition of the ExceptionHandling aspect literally cross-cuts the composition of the basic classes.**

aspect for exception handling added to them. (To achieve clarity, some less relevant details of the AOP style are not represented with full accuracy in this example.) The graph shows how aspects enter into an implementation on several levels: the exception-handling aspect has subcomponents that are distributed across the implementation, at the levels of both classes and methods (methods not shown).

The distinguishing concept of aspectual composition is what is known as *tangling* or *crosscutting* [28]: aspects crosscut or tangle an application in that they are distributed across its composition in such a manner that they cannot be properly separated with the means provided by conventional languages. As figure 4 also shows, crosscutting is directly captured by a composition graph: the part–whole relationships of the aspect crosscut the composition of the rest of the application. The crux of aspectual composition is to allow such crosscutting structures to be defined and dealt with in a convenient manner. Notably, this cannot be handled by rules for hierarchical composition.

# 3. Achieving compositional completeness

Instead of trying to directly address the nature of aspectual composition, we will now address the ability to define and work with any composition that is theoretically possible. Thus we return to the issue of achieving compositional completeness, and this in turn will prove to reveal the precise nature of aspectual composition. While the proof applies to the composition of any system in the sense of systems theory, our concern is the composition of software. The demonstration that follows will not be presented as a formal proof but will instead emphasize explanation.

## 3.1. The compositions

We have so far encountered three composition principles. The simplest one is linear composition, which only allows simple chains. Such compositions are defined by $1 \rightarrow 1$ rules. For example, single inheritance chains are formed by individual subclass/superclass declarations such as “Integer is a subclass of Number”. The next principle in order of power is hierarchical composition, which allows proper trees, and these compositions are defined by $1 \rightarrow N$ rules (one whole, several parts). This is exemplified by instance variable definitions such as “Morph has the instance variables bounds, position, and color”.

The third composition principle was aspectual composition, which so far hasn’t been defined in more precise terms than as being “crosscutting”. But instead of examining this concept more closely and adding a new form of composition, and later possibly another one and another one, the approach taken now is to address the general problem, once and for all as it were. Fortunately this is not as hard as it may appear. Everything required is covered by relatively simple and well-known concepts of computer science.

### *3.1.1. Delineating the class of compositions*

The first step toward a solution is to understand what such a structure may be like, or in other words, to delineate the class of structures that need to be handled. As noted earlier, compositions informally correspond to box-and-arrow diagrams, and hierarchical compositions correspond to the subset of these diagrams that form proper trees. Moreover, we know that composition diagrams always describe part–whole relationships. For this reason, only cycle-free diagrams need to be handled. This is because part–whole relations are directed, one-way relations, which means that an entity cannot be part of itself, neither directly nor indirectly. This is merely a matter of fact, since there is no meaningful interpretation of something being a part of itself. Moreover, it is not a crucial condition for the proof; however, it greatly simplifies the explanation.

As it turns out, this information is all we need. Thus in informal terms, the complete composition mechanism we seek will need to handle all cycle-free box-and-arrow diagrams that can possibly be constructed. In more formal terms, this corresponds to the class of *directed acyclic graphs* (DAGs, cf. figure 4a). Formally, links are one-way because the part–whole relation is asymmetric, and we are thus dealing with directed graphs. Similarly, we are dealing with acyclic graphs because of the restriction to cycle-free relations. Hence, in formal terms, a compositionally complete scheme must be able to compose any directed acyclic graph, where the graph represents part–whole relationships between the composed entities.

### *3.1.2. Covering the class of compositions*

The second step is to identify the format of the rules needed to describe such structures. As previously noted, the theory of syntactic structures states that tree structures can be described by context-free grammars [30]. The same theory also says that *attribute grammars* are required for describing DAGs, and to some extent this defines the rules that are needed. However, this is not a purely theoretical exercise, but should result in a practical technique for design and implementation. Thus, the rules should be expressed in a maximally useful and sensible format. To reach that point the nature of these DAG structures needs to be clarified.

The approach taken here is to regard DAGs as a generalization of hierarchical structures; this is simply the complement of the fact that trees make up a subset of DAGs. Hierarchical decomposition in programming languages is already well-established, it is the foundation of structured programming as we know it, and trees are widely recognized to be rather easy to understand and reason about, as well as practical to deal with. But there is also a special reason for treating DAGs as an extension of tree structures. Very often the additional power of composition afforded by DAGs is not needed, but composition into trees will be sufficient. In these cases the extension will be transparent, and the new scheme will be equivalent to hierarchical composition. And when the

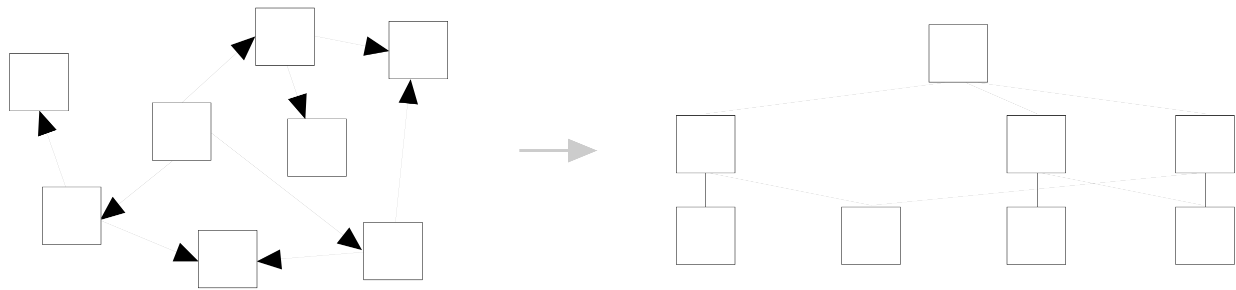

**Figure 4. A directed acyclic graph (a), and (b) the same graph toplologically sorted to place all parents above their children.**

additional power is needed, it will appear as an extension to hierarchical composition, not as an entirely new and different composition scheme. So this means that the new scheme becomes a strict extension of hierarchical composition, which doesn't complicate the composition of more simple structures.

*3.1.3. Making all compositions treelike*

The operation that turns DAGs into treelike structures follows from the fact that these structures are acyclic. This means that DAGs like trees can be drawn so that a parent node always appears above its children (or wholes above their parts). Formally, this corresponds to performing a *topological sort* on the nodes of the graph [1]. (The topological sort operation is what requires the graphs to be directed and acyclic; relaxing these constraints would complicate the principles and make them harder to use, but wouldn't invalidate them.)

The topological sort transform the normally disorderly-looking DAGs into quite a treelike format. The formal remaining difference between DAG.s and trees lies in that a tree node may have any number of outgoing arrows but only one incoming arrow, whereas the nodes of a DAG lack the last restriction, they may have any number of incoming arrows as well. That is, whereas a tree node always has only one parent, a DAG node may have multiple parents.

In a topologically sorted DAG diagram (cf. figure 4b) these multiple parent links can be seen as though they were additional crosscutting links added to a regular tree. Thus, the formal difference between DAGs and trees can be fruitfully characterized as parent nodes being able to "cross-reference" or "hyperlink to" their children. This also shows that the most convenient form of attribute grammars for this domain is *referential attribute grammars* [20]: Trees are described by context-free *Whole* → *Part* rules, and to describe DAGs the only kind of "attribute" we need to add to these rules is *cross-references* to other nodes in the composition tree. In practice, these are not treated as "attributes" but as any other part of a composition.

This additional ability of DAGs to cross-reference nodes is just what is needed—and it is *all* that is needed—for aspectual composition. In the previous example, ordinary *Whole* → *Parts* rules were sufficient for describing the composition of the main program, as well as the aspect as such, but to bind the aspects to their proper locations we need the ability to make cross-references between elements. Moreover, this amounts to the exact and entire difference between hierarchical composition and the ability to compose anything that is theoretically possible. That is, as described here, aspectual composition amounts to this exact difference.

## 3.2. Perspectives

The last step toward practicality is to avoid the overlaps in a topologically sorted DAG by decomposing it into subtrees (figure 5). This operation derives from the fact that any given 1 → N relation does not contain any overlap, and that an N → 1 relation (i.e. several parents referencing the same child) can be divided into N simple 1 → 1 relations. By applying this principle recursively, any topologically sorted DAG can be untangled into a number of simple subtrees. This technique may be considered as taking different *perspectives* on the overall composition DAG: when a child node is cross-referenced by several parents, it may be regarded as being part of more than one perspective, each corresponding to one parent.

Since perspectives make up the crucial technique for untangling complex compositions, they deserve to be given a special status within the language and tools. Here, a simple diagrammatic convention can be used to represent them, or more precisely, to hide them, as illustrated in figure 6a. Instead of drawing the full structure of the perspective into the diagram, one edge of its parent is emphasized to represent the perspective. The heavier edge can be thought of as a stylized image of a dimension that isn't fully visible from the current point of view (figure 6b).

While perspectives resemble aspects, they are much older as an object-oriented technique [6, 9, 18]. From their origin in knowledge representation, they are also more powerful and general in their application than aspects, for example in their potential to replace inheritance [9, 18, 24]. Here we may give perspectives a precise definition in terms of compositional

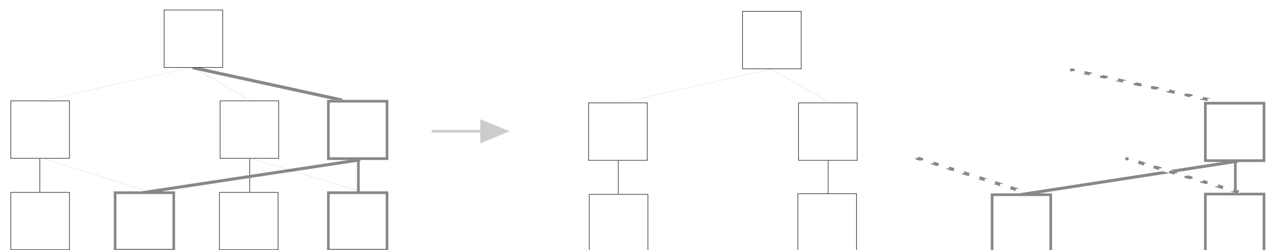

**Figure 5. Untangling a crosscutting DAG into regular trees. On a conceptual level, each tree corresponds to a *perspective*.**

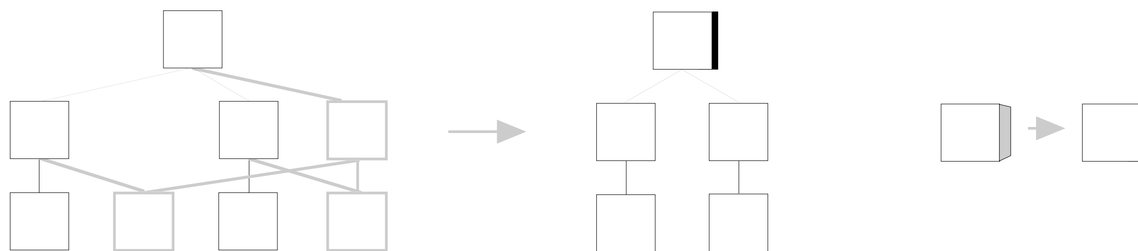

**Figure 6. (a) Hiding a perspective (compare to figure 5). (b) the black edge is a stylized representation of a hidden dimension.**

completeness: A perspective corresponds to a subtree of the composition graph of an object. Since the subtree spans exactly those part nodes that are reachable from the root node, any perspective is uniquely defined by the root part. Strictly, in this definition every part becomes a perspective, and since parts obey encapsulation, each one sees only its own subparts, i.e. "its own point of view" of the whole composition. However, there is an added *capability*-like facility that corresponds to "taking a perspective on" an object *from the outside*. Thus if another object has a reference to this perspective, then any access to the object via this reference will be restricted to the protocol understood by this perspective, rather than to the protocol of the whole object. The protocol of the perspective consists of the methods defined in the root part (or its de-encapsulated subparts). A simple implementation of this facility is to associate a perspective with its own, separate selector name space.

# 4. Applying the scheme in practice

Section 2 illustrated how inheritance and aspect-oriented programming can be reformulated as special cases of generalized composition. The rest of the paper will present the scheme in its own terms, using its own preferred style for performing program composition, rather than to emulate other schemes such as aspects or inheritance.

## 4.1. The nature of the composed parts

So far, little has been said about the units of composition, the parts (or informally the boxes of the diagram), and their contents. On a theoretical level, it is necessary and sufficient that a part may contain other parts plus the basic units of content. These basic units are the primitives of state and behavior that cannot be decomposed into more elementary units. The power of the composition scheme allows the composed units to be reduced to their simplest possible form, since they still can be composed to any level of complexity.

For a practical programming language construct, the corresponding solution is slightly different. The primitive unit of state is a word of memory; in practice this translates to an instance variable holding an object reference, as is normally the case. However, the basic unit of behavior will not be primitive instructions but methods. This is because methods, the provisions that languages already have for composing behavior, are sufficient as they are. In the interest of simplicity they are therefore not replaced. So, in practice a part may hold objects, methods, and other subparts, or rather references to them. Extending Self's concept, we may collectively refer to these—holders of not just state and behavior as in Self, but of all three kinds—as *slots*.

Parts, the basic units of composition, therefore become highly similar to ordinary objects. A part has the same semantics as an object, and composing an object out of parts follows the usual principles for composing objects out of other objects, but for one provision: just as an object need not contain a copy of each of the methods it responds to, an object also doesn't have to contain a copy of each of its parts. Hence, parts belong to an object's abstract composition, but need not be explicitly stored within the object they compose.

### *4.1.1. The advantages of encapsulating parts*

Since parts can contain state but also observe encapsulation, this allows state to be encapsulated by the functional parts to which the state properly belongs. This is not typically possible using inheritance and is not allowed by e.g. Self or Smalltalk. However, here it is not merely possible—it is even the preferred style of use.

This lack may be perceived as an insignificant drawback in current languages. If so, however, this is probably because their inheritance mechanisms place a rather low limit on the number of units (superclasses) an object can be composed from. Thus not being able to keep the state of these parts separated will usually present few problems. Still, whenever the "fragile base class" issue *does* become a significant problem (i.e. when sub- and superclasses have conflicting instance variable names). The fragile base class problem is resolved by the encapsulated parts of the present scheme. Also cf. [37].

However, more importantly, when properly applied the present scheme leads to a much higher degree of complexity in the object compositions. In its most complex application so far, the average objects were composed out of around 30+ subparts; the extreme case was an object with 70+ subparts (while still having no more than three or so instance variables). Moreover, some subparts would occur multiple times within the same object, and in some cases they would even recursively contain a different instance of the same kind. The object with 70+ subparts was in fact composed out of two of the 30+ compositions, plus the necessary connecting elements.

Lastly, the use of multiple, potentially conflicting perspectives within one object also vitally depends on the ability to keep the internals of different perspectives from conflicting with each other. For these reasons, the ability to encapsulate a part's contents—i.e. state, behavior and subparts—becomes crucial and indispensable once one starts to exploit the abilities of this scheme to create more elaborate compositions.

## 4.2. Functional decomposition, a.k.a. role-based design

A scheme that is compositionally complete brings the full freedom to structure a system in any possible way. Moreover, the scheme itself is neutral with respect to what it composes, and therefore any semantical interpretation could be applied to the composed parts. This leads to the following question: When you can choose any principle whatsoever for decomposition, what

principle do you choose? (Inheritance, for example, uses *is-a-kind-of* relations.)

On a theoretical level the approach propose here is *instrumentalism*. Its guiding principle is the analysis of phenomena in terms of their function, purpose, or effect. According to this position, the proof is in the eating of the pudding, as opposed to in the pudding itself. Articulated by John Dewey in the early 20th century [12, 13], instrumentalism is still the state of the art for theories of knowledge, and as far as its ability to scale is concerned, it has provided the basis for e.g. modern physics (including quantum mechanics and other nontrivial domains) for almost a century; this ought to provided some reassurance in this regard.

Thus, the principle proposed here is to decompose problems in terms of function or purpose, and to let each unit of composition, each part, represent one such function. As it happens, this turns out not to be an entirely novel idea. Such a functional part corresponds to a *role* in role-based design [2, 41] and a *responsibility* in CRC terminology [5]. These role-based approaches seem to have reinvented instrumentalism on a practical level, for use as a design and analysis technique. From this point of view, a part might be regarded as a simplest language construct for representing roles or responsibilities. This has otherwise been a difficulty in turning role-based designs into object-oriented implementations [3, 19, 26, 41].

Thus, the present scheme is fortunate to draw on how role-based analysis has established itself as a reliable, practically proven approach through years of OO practice. The analysis of Model–View–Controller in [5] can illustrate how role-based analysis translates into the current scheme. This example states the View object in MVC as having the responsibilities *Render the model* and *Transform coordinates*, and the Model object as having the responsibilities *Maintain probem-related info* and *Broadcast change notification*. Here, each of the responsibilities would correspond to one functional part, and to give an object the ability to broadcast change, one would add this functional part to the object. In informal terms, this corresponds to "giving" it this ability or functionality, which is the essence of what being a Model is about. Currently you instead inherit this functionality via Object, which with multiple inheritance would be from Model. Recognizing that an object has this functional part, rather than it being a subclass of Model, arguably provides a clearer and conceptually more direct explanation of what the object does. This clarity follows from basing the analysis on what an object does rather than what it is; in other words, the instrumental point of view.

## 4.3. An example application

The next example comes from using this scheme to solve a research problem within Ubiquitous Computing [16]. It was the third application domain that this scheme has been applied to. It will here serve to illustrate two points: Firstly, how this scheme may obtain a much better separation of concerns than inheritance, and secondly, how the scheme is used for aspect-like compositions. The research problem is based on allowing physical computing devices and virtual, digital objects in ubiquitous computing environments to interoperate in various configurations, in a manner that is smooth and transparent to the user, and without requiring any configuration effort to make the interoperations work. The technique involved allows a user to specify (or "combine") various physical or virtual objects using a magic wand-like combination device [21]. The wand is currently implemented as a handheld computer that uses infrared signals for object selection and wireless networking for inter-device communication. Similarly the devices involved have infrared receivers to pick up the selection signal. In a representative scenario, the user would for example select a document on a tablet computer and then select a wall-based display, by pointing at each of them in turn.

The task of the combination algorithm is to identify the possible meaningful actions that could result from combining the selected objects, to allow these actions to be presented as options on the wand's display. The user may then carry out an operation by simply selecting it. The prototypical operation to result from the above combination would be to show the document on the wall display. Some advantages with this technique, besides transparency of use, are that a user can come into an environment and immediately start using the available technology without knowing the device addresses, or knowing the available functionalities, or even the commands required to use a certain functionality.

The wall display and the document are given as arguments to the combination algorithm. The arguments are provided as objects, having been retrieved in a prior stage. If these domain objects are modeled using inheritance, the usual problems of inheritance present themselves, if perhaps to an unusually high degree: It is hard to isolate interactions and combinations, so a great deal of code duplication is necessary, and hand-coding of individual combinations is required to handle each case correctly. As a result, the combination algorithm is distributed across every class in the domain model, with a great deal of redundancy and brittle cross-dependencies. If there is such a thing as "code smell", then in this case there is something truly rotten in the state of Denmark, due to the limitations of inheritance. These problems are more fully described in [16].

When the present composition scheme is used instead of inheritance, the various objects are decomposed into their functional parts. Here, the relevant functions are that the document is visually Renderable, and the wall display is a Renderer, capable of displaying anything that is renderable. To render the document on the display, the Renderable and Renderer parts in the respective objects enter into a classical collaboration pattern. The use of functional composition in itself guarantees that combinable properties now may always be represented once and only once, instead of having to distribute the combination code across all the various entities that could participate in a rendering operation. (From the above principles it even follows that "once and only once", i.e. redundancy-free composition, can be attained for any composition problem. This point is however beyond the scope of theis paper.)

It bears noting that the solution architecture so far is not specifically tailored to serve combination finding, but uses the conventional approach (functional decomposition), which thus proves highly suitable for implementing device capabilities in a useful way. In contrast, inheritance would say that a Document is a kind of VirtualObject, a PDA a kind of Computer, and so forth, but this wouldn't provide much help for either the problem of representing device capabilities, or to finding combinations. However, because the analysis in terms of function is not restricted to one point of view, but can handle multiple functionalities in parallel, the combination functionality can now be added in a manner that meets two important conditions: it is noninvasive and it suits the problem of finding combinations. The resulting solution is based on adding combination functionality as a separable perspective (roughly corresponding to and aspect)

The basic device functionality is provided as ordinary methods; The Renderable part contains methods for rendering the document, and the Renderer part has methods and state that implement the rendering of renderable entities on its surface. In actual fact both of these functional parts would themselves have rather elaborately decomposed implementations.

The additional information needed should annotate the specific operations that would be made available; for Renderer the message *render: aRenderable* would be indicated along with a description of the operation, "Display <the renderable> on <the renderer>", which would be used to present the operation in a menu. To avoid adding this information to the Renderable part itself, it is instead provided non-invasively by adding a Combinable part to the Renderable part, and to any other parts that should be involved in the combination algorithm. The Combinable parts all belong to the Combination perspective, and this is where they are specified for combinable objects, by cross-reference to these basic objects.

The composition of a Document object would now look similar to figure 7. Here there are some special compositional features worth noting. Firstly, the basic functional parts are de-encapsulated, since the wall display object itself should "acquire" the basic functionalities, that is be able to respond to for example the *render:* message. However, the contents of the Combinable parts in turn are encapsulated, and so a Document does acquire the abilities of being Renderable and being Authored, but not those of being Combinable. Secondly, the diagram convention of leaving out this perspective is shown in AC3; this corresponds to browsing the system from a different point of view.

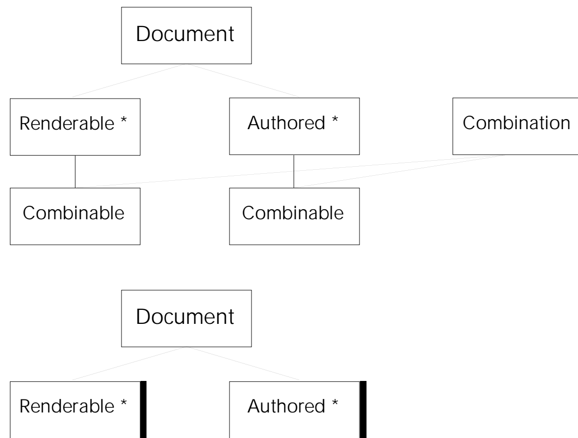


**Figure 7. (a) A Document with Renderable and Authored parts, each having a part belonging to the Combinable perspective. (b) The perspective left out of the composition graph.**

The inheritance version of the combination algorithm would have handwritten code in the classes of every potential combinee, where these would send messages to each other to negotiate whether they make a valid combination, and if so, what operations would apply. Here, a valid combination can be specified simply by indicating for each Combinable what its corresponding combinee is; one indicates simply that a Renderer part can be combined with any Renderable, and so on. This information is provided in a noninvasive manner in the various Combinable parts (figure 7).

This version of the search algorithm exploits the fact that functional parts are first-class objects, and therefore uses reflection to traverse the composition graphs of the two potential combinee objects, to single out those basic functional parts that have Combinable subparts (cf. figure 8) added to them. The eligible Combinable parts in the two objects are then exhaustively searched for matching pairs. In this case, Renderer in the WallDisplay object specifies Renderable as its partner, and such a matching part is also found in the Document object (figure 8).

In this manner the information can be provided in a structured fashion that suits the problem, and so the combination search becomes drastically more simple, and it does not require any combination code to infiltrate the basic implementation. Instead the search algorithm can be specified in pure form, once and only once and in its own functional part; the domain objects need only contain annotation information that the search algorithm can use.

## 4.4. Composing behavior

Since behavior has a more complex structure than static content, the composition of behavior requires special attention. *Aspect weaving* is a central concept in aspect-oriented programming [28]. It is the problem of integrating aspect and non-aspect behavior by taking the appropriate code from the aspects involved, and integrating it with the regular code of the program, "weaving" these pieces into a single piece of executable code. In the more general-purpose scheme presented here, aspect weaving corresponds to the general problem or composing methods out of separate pieces of code, as provided by multiple parts.

The parts of a method, the messages etc., are ordered, they may be nested into blocks, and so on. For this reason, some mechanism must allow this additional information to be provided when different pieces of behavior should be assembled into new, composed behavior. The simplest solution is to simply write a method that invokes the parts' methods in the desired order:

```
printOn: aStream

    a printOn: aStream.
    b printOn: aStream.
    c printOn: aStream.
```

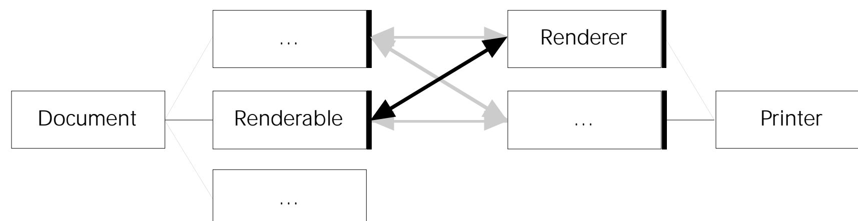


**Figure 8. Schematic of the combination search: All Combinable parts in each combinee are matched against those in the other. Here the Renderer–Renderable combination forms a matching pair.**

However, often the same pattern would be repeated for many or all methods, so a way of specifying a general pattern for all methods is convenient. A common approach both in multiple inheritance and aspect-oriented programming is to use directives such as *before*, *after* and *around* [27, 29]. However, such directives provides an unnecessarily restricted range of options. Instead, one may turn the above concrete method into a general *composition method* that has been generalized to work for any message. Thus to invoke the implementations in part a, b and c you would use the following composition method:

```
methodFor: aMessage

    aMessage sentTo: a.
    aMessage sentTo: b.
    aMessage sentTo: c.
```

A part that writes tracing statements and then passes on the message to *otherPart* could thus be written in the following way:

```
methodFor: aMessage

    Transcript show: 'Entering ...'.
    aMessage sentTo: otherPart.
    Transcript show: '... leaving'; cr.
```

This technique allows the full range of the language to be used, including more advanced forms of message composition using e.g. blocks. For example, the basic functionality of one part could be wrapped in exception-handling code provided by another part, using block closures to compose the respective parts:

```
methodFor: aMessage

    [aMessage sentTo: computationPart]
      onExceptionDo:
         [aMessage sentTo: exceptionHandlerPart]
```

This composition method sends the message to *computationPart*, wrapping it in a hypothetical error-handling message *onExceptionDo:*, and if an exception occurs it will send the same message to the *exceptionHandlerPart*. In this way an implementation of *divideBy:* could be separated into the respective parts:

```
Computation>>divideBy: aNumber

    ^self primitiveDivideBy: aNumber
```

and

```
ExceptionHandling>>divideBy: aNumber

    ^self primitiveDivideByZeroError
```

## 4.5. Static composition using program specialization

In this way, ordinary message passing works perfectly well for composing code from different parts into new whole methods. The only provision is that this technique may be rather slow, especially since the proper use of the present decomposition scheme leads to much higher levels of decomposition, which then requires more messages for gluing the individual pieces of code together. The result would be poorer performance than with existing composition schemes. However, by using program specialization, also known as partial evaluation, [23] this can be turned into highly efficient runtime behavior. Program specialization is a program transformation technique that can be regarded as the most general form of code optimization, subsuming several techniques of more limited scope, such as inlining and constant folding. You specialize a program by providing values for some of its input parameters. Program specialization then identifies those of the program's operations that can be computed ahead of runtime because all their input data is known at the time of specialization. It then outputs a specialized version of the program where it has replaced all those operations with their results. A trivial application would be to transform the expression $3 * 2 + a$ to $6 + a$. Similarly, if the expression $3 * b + a$ is specialized for $b = 2$, the result will be identical.

The code that results is highly efficient by any measure. The effect is that the runtime expense of a computation becomes no higher than it absolutely needs to be. This principle may be expressed as "when the cost is zero, the price is nothing": if e.g. the result of a certain composition can be computed once and for all, then it shouldn't be computed every time it is invoked.

The specialization technique of inlining messages is particularly relevant here. Where runtime message passing composes behavior dynamically, inlining will do the same thing statically. It does not merely early-bind the message receiver, but in effect eliminates the entire message, "pasting" the code of the receiving method into the sending method, while performing parameter substitution and whatever else is necessary to produce an equivalent program. The effect is that the message that links the pieces of calling and called code will vanish, merging both pieces of code into one. If for example the earlier composition method

```
methodFor: aMessage

    [aMessage sentTo: computationPart]
      onExceptionDo:
         [aMessage sentTo: exceptionHandlerPart]
```

is specialized for a message with the selector *divideBy:* the result would be as follows:

```
divideBy: aNumber

    [self primitiveDivideBy: aNumber]
      onExceptionDo:
         [self primitiveDivideByZeroError]
```

Here, the messages that invoke the two *divideBy:* methods above have been completely removed, and only the cores of those methods remain, one inside each block.

## 4.6. A second example: BitBlt

The following example will illustrate just how efficient the resulting code can be when it has been translated from a high-level design in terms of functional parts, using program specialization, and inlining in particular. BitBlt is the original Bit BLock Transfer operation of Smalltalk-80, used to transfer bitmap graphics e.g. to the high-resolution display [17]. As pixels may be smaller than memory words, this involves not merely simple memory moving operations, but potentially also shifting the bits to handle sub-word positioning. And with the advent of color graphics, the operation has been extended to handle the various formats that pixels may come in, e.g. 2, 4, or 8 bit color map indices, or direct RGB values with alpha channels in 16 or 32 bits per pixel. In its general formulation, the operation also includes a transfer function, in which case it computes the function from the source and destination pixels, and overwrites the destination with the resulting value.

To achieve self-simulation capability and platform independence, the Squeak system implements BitBlt in a subset of Smalltalk [22]. This code can either be run as is, which is highly useful during development but far too slow for real-time graphics, or it can be translated into C, from which highly efficient machine code may be generated. The need for speed naturally places tight constraints on the implementation. This applies in particular to the most speed-sensitive inner loop that performs the actual pixel transfer. In contrast to normal Smalltalk code, which typically comes as a number of small, well-factored high-level methods for maximum clarity, the inner loop is hand-optimized to maximize its speed and therefore looks much the opposite.

This BitBlt inner loop is a large monolithic method, over a hundred lines long, and with several highly similar chunks of code repeated with subtle variations. It could in other words easily be refactored, to yield a much better organized but also slower version. Moreover, several variants of this inner loop are provided, which exploit various special conditions to make some important cases as fast as possible. A mere glance shows that there are great similarities also between these variants. They differ merely in the optimizations, which have typically been applied by hand; any non-optimization differences could only be counted as oversights. In all, it is evident how the concessions that have been made to achieve maximum performance come at the expense of most everything we consider good programming practice, and this trade-off would not turn out more favorably if coding directly in C, or even assembly language. It has often been said that object-oriented design is not suitable for certain applications, since it prevents an reasonable implementation, for example because it limits performance. BitBlt may be the ultimate illustration of this point.

### *4.6.1. A functionally decomposed BitBlt*

It seems appropriate to have the architecture follow the high-level description of the BitBlt operation, as a general pixel-by-pixel transformation that takes the source and destination pixel maps as its inputs, and the destination as output. From this, the source and destination may be defined as PixelMaps (figure 9a). As a result, many aspects of the algorithm can be expressed as properties of the pixels: this includes the pixel sizes in bits, and whether they are indexed or direct colors, a possible RGB(A) format of the color, and so on.

Also, much of the low-level processing can be encapsulated as higher-level operations on the pixel maps, thereby hiding dozens of parameters like the respective width and height of the pixel maps, and much internal state that is kept track of during the operation, like positions of the current pixel in each map. This is an example of how functional parts can be effectively used to encapsulate state.

A PixelTransform can be used to organize the remaining aspects of the computation: besides the transfer function, various auxiliary operations like pixel alignment, halftoning, and others (figure 9b). To handle pixel alignment, bit rotation and masking can be encapsulated in a Skew part.

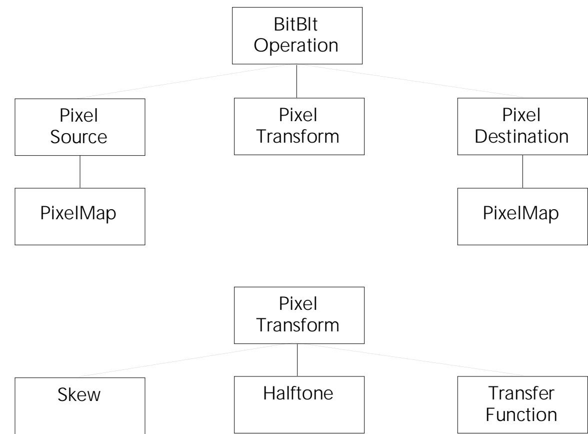


**Figure 9. Schematic of the composition of BitBlt. (a) The overall operation is a transformation that applies a transfer function to a source and a destination pixel map. (b) The transfer computation involves additional steps to the transfer function itself.**

This description of the BitBlt composition has intentionally been kept brief and undetailed, so as not to distract from the main point of this example, which is that a highly structured implementation need not incur any performance loss. However, the description ought to have shown that the resulting composition closely matches a high-level explanation of how the operation works.

### *4.6.2. The resulting code*

By using this design, the resulting source code for the general (non-inlined) inner loop is straightforward to understand and like regular Smalltalk approaches the level of pseudo-code. It looks roughly like this:

```
copyLoop

    self setupVerticalLoop.
      1 to: sourceMap height do: [:line |
          self prepareLine: line.
          self resultPixel: self transformedPixel
               edge: skew leftEdge.
          self nWords > 1 ifTrue: [
              self copyHorizontalLine.
              self resultPixel: self transformedPixel
                   edge: skew rightEdge].
          self nextLine]
```

In this version, the pixel transformation is expressed directly in its general, most high-level form:

```
transformedPixel

    ^transferFunction
      source: source pixelValue
      destination: destination pixelValue
```

Spelled out, this means “return the result of applying the transfer function to the source and destination pixels”. In Squeak’s hand-written version the code for the same operation looks as follows:

```
…
thisWord _ self srcLongAt: sourceIndex. "pick up next
word"
sourceIndex _ sourceIndex + hInc.
skewWord _
  ((prevWord bitAnd: notSkewMask) bitShift: unskew)
      bitOr:    "32-bit rotate"
  ((thisWord bitAnd: skewMask) bitShift: skew).
prevWord _ thisWord.
mergeWord _
  self mergeFn: (skewWord bitAnd: halftoneWord)
      with: (self dstLongAt: destIndex).
self dstLongAt: destIndex put: mergeWord.  "write  result"
destIndex _ destIndex + hInc
…
```

The code that results from applying specialization to the high-level version is identical to the code shown here, except that intermediate variables are used only when necessary. A first version was written to mimic the hand-written code exactly, but the later version captures the intention behind the code much more directly, by specifying what should be computed without saying how to do it (i.e. whether to use intermediate assignments). It relies on the ability of the specialization engine to insert local variables when necessary, i.e. when the result of a computation is used more than once.

And instead of writing alternative versions by hand to optimize them for certain conditions, different variants can be generated by altering the parameters given to the specializer. Then, whenever possible, optimizations are applied automatically during specialization. For example, one of the most important special cases is when no source bitmap is used (e.g. when simply filling the destination with a solid color). To obtain a version of the inner loop that is optimized for this case, you merely specify a no-op-style part as the SourceMap, which generates empty operations for every source-related part of the code. The resulting inner loop is instruction-by-instruction equivalent to the hand-optimized version of the same method.

*4.6.3. Optimal performance*

Table 1 compares the speed of hand-written and re-engineered versions of the BitBlt inner loop, where these have been translated into C. A deeper investigation traced the discrepancies to the used C compiler's varying ability to optimize code that performs the same computation but with or without explicitly assigning intermediate values to local variables.

**Table 1. A comparison of benchmark execution times in ms (smaller is better) of hand-optimized and mechanically specialized high-level versions of BitBlt. *Paint* and *over* are two common BitBlt transfer functions.**

| | depth | hand-optimized | re-engineered | ratio |
|---|---|---|---|---|
| paint | 1 | 63 | 62 | 98% |
| | 8 | 190 | 179 | 94% |
| | 32 | 608 | 611 | 100% |
| over | 1 | 10 | 11 | 110% |
| | 8 | 78 | 84 | 108% |
| | 32 | 323 | 350 | 108% |

As the comparison shows, the performance cost of reshaping BitBlt as a high-level composition is practically none, even compared to a hand-optimized version of this highly speed-sensitive algorithm. And whereas the hand-written code trades a great deal of clarity and conciseness for optimal performance, the re-engineered version has been given a high-level architecture that lies very close to the most convenient conceptual description of the operation. As seen in figure 9b, the functional part responsible for the PixelTransform is itself decomposed into successive functional steps in a data-flow pattern. This means that this version could be extended in a well-structured manner by adding additional parts to the PixelTransform, where each new part would add one new ability, for example to perform byte-order conversion or to exploit special hardware.

# 5. Related Work

## 5.1. Software composition

There is already a vast literature on software composition [4, 11], where the approaches have had varying degrees of success; it would no doubt seem rash to claim to address such a formidable problem in the general case. In a nearby case, the problem of compositional semantics arose in relation to subject-oriented composition rules [34]. The reason why this scheme works whereas others do not could be that it carefully separates the concern of compositional "syntax" from the concern of compositional semantics. It separates the concern of composing software elements from the concern of the contents of these elements, by taking care to ensure that the contents will not be affected by the composition that is performed.

This separation can be illuminated by how program specialization works, because it too carefully avoids the issue of program semantics, by taking care not to perform any transformation that may change the semantics of the program. It will for example transform *3 + 2* into *5* because the semantics of this computation will not be affected by whether it is performed at runtime or ahead of runtime. Thereby the semantic correctness of a transformed program is left unaffected, and will only depend on the correctness of what the programmer has written. In fact, program specialization has been used in the present scheme precisely because it has this property: It can therefore be used to compose behavior without affecting the semantics of the composed elements.

That is, the present approach has not solved the problem of compositional semantics. On the contrary, by recognizing that composition is a purely structural concept, this scheme can address that specific problem in the general case, by taking care to *circumvent* the problem of compositional semantics.

## 5.2. Inheritance

Various problems related to method lookup in multiple inheritance (MI) have been well documented [9, 14, 37]. The way in which the present scheme resolves these problems is highly similar to the one described in [9]. However, since that approach is based on inheritance, the definition of a point of view becomes complex and non-intuitive. Here there is no need for introducing a special construct for this purpose, instead functional parts (and perspectives) already handle this problem. In fact, if the examples from [9] are recast as composition graphs, the results resemble perspectives very closely, with the infamous "multiple inheritance diamond" corresponding to multiple perspectives cross-referencing the same part or parts.

The ability to form general DAGs is neither unique nor new; MI has always had this ability. The critical difference indeed appears

to be that inheritance always enforces de-encapsulation, cf. above and [37]. The present scheme fully exploits this fact by eschewing inheritance for composition. It builds new entities out of functional parts with precisely carved abilities, instead of deriving new entities from other complete entities, in an all-or-nothing fashion (i.e. deriving new classes from other fully capable classes). This difference, composition from small, precise parts instead of derivation from large, fully capable entities indeed seems to be the crucial factor that makes a great difference.

In this, there is a distinct similarity between functional parts and mixins [8]. However, in comparison to the gradual and rather ad hoc evolution (and divergence) of inheritance into MI, mixins, points of view, and so forth, it should be clear that the present composition scheme results from a clean-sheet design and so rests on a conceptually simple and clear foundation. On this point, the present scheme stands in contrast to Self, which held on the notion of inheritance, resulting in an odd mix of concepts from inheritance and composition, as in "parents are shared parts of objects", "traits objects", and so on [10].

## 5.3. Aspect-oriented programming

Aspect weaving corresponds to a special form of behavior composition, as discussed above. Aspect weaving is usually considered to be performed ahead of runtime, even though nothing prevents it from being performed dynamically, at runtime; the use of program specialization for this purpose was discussed above. And even though it addresses a less general problem, the resulting solution introduces more additional complexity into the base language than has been done here. The simplicity obtained here is due to the general but conceptually simple design of the composition mechanism, which draws on the existing mechanism for composing behavior, namely message passing, and this is all the machinery needed for the general case.

Composition methods were introduced as a means for allowing the same composition pattern to be used for several methods in the same part. In contrast, AspectJ introduces three types of construct, *join points*, *pointcut designators*, and *advice declarations* [27]. Each of the latter categories have multiple members, which in relation to existing language constructs range from the familiar to the quite unfamiliar. Thus, the concepts introduced by the present scheme are far fewer and less foreign to the base language than those introduced by AspectJ. The main reason for these differences is that AspectJ is specifically tailored for one new category of software composition, namely aspectual composition. Such a specialized scheme would not work here, as the present scheme needs to work for program composition in general.

Like multiple inheritance, also AOP schemes have the theoretical capacity to describe general DAGs. For example, hyperspaces [39] provide this capability. However, beside the fact that HyperJ much like AspectJ introduces a substantial amount of new language machinery, it also requires the full composition graphs to be specified from top to bottom, as it were, using absolute references. To obtain good encapsulation, it is essential that the composition scheme allows cross-references to be both encapsulated and relative. For example, the compositions of 30+ and even 70+ parts mentioned above were all generated from one single top-level rule each, with the equivalent of five parameters whose values cascade into other parameterized rules, to a total of thirty or seventy nodes. Thus, relative and encapsulated composition rules are essential to allow the compositions to scale in a controlled manner.

# 6. Conclusions

This paper has sought to show that the aim of achieving compositional completeness, i.e. the ability to decompose any given system in any manner that is theoretically possible, is not as hard to reach as intuition would suggest: Any composition can be described by a directed acyclic graph, and any such graph can be fully untangled by decomposing it into multiple "perspectives", each of which is an ordinary non-tangled tree. In this manner, any system can be described in a structured and manageable way.

This work was originally inspired by the sheer multiplicity of approaches to inheritance and aspect-oriented programming, and a feeling that they all had some shared core. Compositional completeness is that core. A scheme that has this property is also the superset of all other software composition schemes, for example any variants of inheritance and aspect-oriented programming. Therefore a language based on this core would be smaller, yet substantially more expressive and powerful than current languages.

Moreover, this paper has tried to show by example that the practical application of this scheme leads to designs and implementations that are both problem-oriented and highly structured. The scheme brings the freedom to structure a system in any possible way, and I have proposed that this freedom is best utilized by decomposing problems in terms of function or purpose, in a manner that corresponds to role-based design. The concluding BitBlt example showed that this approach can be applied even to highly performance-critical applications, and then without incurring any efficiency loss, even when compared to hand-crafted, optimized C code. My intention has been to show that this scheme is not merely a theoretical construct, but that it also works well when put into practice.

# 7. REFERENCES


[1] Aho, A.V., J.E. Hopcroft, and J.D. Ullman, The design and analysis of computer algorithms. 1974, Reading, MA: Addison-Wesley.

[2] Andersen, E.P. and T. Reenskaug. System Design by Composing Structures of Interacting Objects. in European Conference on Object-Oriented Programming (ECOOP'92). 1992.

[3] Bækdal, K.K. and B.B. Kristensen. Perspectives and complex aggregates. in Proceedings of OOIS 2000. 2000.

[4] Batory, D. and B.J. Geraci, Composition Validation and Subjectivity in GenVoca Generators. ACM Transactions on Software Engineering, 1997. 23(2): p. 67-82.

[5] Beck, K. and W. Cunningham. A Laboratory For Teaching Object-Oriented Thinking. in Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'89). 1989.

[6] Bobrow, D.G. and T. Winograd, An overview of KRL, a knowledge representation language. Cognitive Science, 1977. 1(1): p. 3-46.

[7] Böhm, C. and G. Jacopini, Flow diagrams, Turing machines and languages with only two formation rules. Communications of the ACM, 1966. 9(May): p. 366-371.

[8] Bracha, G. and W. Cooke. Mixin-based inheritance. in Proceedings of the 1990 ACM Conference on Object-Oriented Programming Systems, Languages and Applications. 1990.

[9] Carré, B. and J.-M. Geib. The Point of View notion for

multipe inheritance. in Proceedings of ECOOP/OOPSLA’90. 1990.

[10] Chambers, C., D. Ungar, B.-W. Chang, and U. Hölzle, Parents are shared parts of objects: inheritance and encapsulation in Self. Lisp and Symbolic Computation, 1991. 4(3): p. 207-222.

[11] Czarnecki, K., Aspect-Oriented Decomposition and Composition, in Generative Programming: Methods, Techniques, and Applications, K. Czarnecki and U. Eisenecker, Editors. 1999, Addison-Wesley: Reading, MA.

[12] Dewey, J., The Quest for Certainty: a study of the relation of knowledge and action. Gifford lectures ; 1929. 1929, New York: Minton Balch. 318.

[13] Dewey, J., Logic: the Theory of Inquiry. 1938, New York, NY: H. Holt and Company.

[14] Ducournau, R., M. Habib, M. Huchard, and M.-L. Mugnier. Proposal for a Monotonic Multiple Inheritance Linearization. in Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA’94). 1994.

[15] Elrad, T., R.E. Filman, and A. Bader, Introduction to aspect-oriented programming special issue. Communications of the ACM, 2001. 44(10): p. 29-32.

[16] Gedenryd, H., S. Holland, and D.R. Morse. Meeting the software engineering challenges of interacting with dynamic and ad-hoc computing environments. 2002. Submitted.

[17] Goldberg, A. and D. Robson, Smalltalk-80: The Lanaguge and its Implementation. 1983, Reading, MA: Addision-Wesley.

[18] Goldstein, I.P. and D.G. Bobrow. Extending Object Oriented Programming in Smalltalk. in Proceedings of the First Lisp Conference. 1980.

[19] Gottlob, G., M. Schrefl, and B. Röck, Extending object-oriented systems with roles. ACM Transactions on Information Systems, 1996. 14(3): p. 268-296.

[20] Hedin, G. Reference Attributed Grammars. in Second Workshop on Attribute Grammars and their Applications–WAGA99. 1999.

[21] Holland, S., D.R. Morse, and H. Gedenryd. Ambient Combination: a New User Interaction Principle for Mobile and Ubiquitous HCI. 2002. Submitted for review.

[22] Ingalls, D., T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. in Proceedings of OOPSLA’95. 1995.

[23] Jones, N.D., An Introduction to Partial Evaluation. ACM Computing Surveys, 1996. 28(3): p. 480-504.

[24] Kay, A., The Early History of Smalltalk, in History of Programming Languages–II, T.J. Bergin and R.G. Gibson, Editors. 1996, ACM Press: New York. p. 511-578.

[25] Kempf, J., W. Harris, R. D’Souza, and A. Snyder. Experience with CommonLoops. in Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87). 1987.

[26] Kendall, E.A. Role model designs and implementations with aspect-oriented programming. in Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA’99). 1999.

[27] Kiczales, G., et al., Getting started with AspectJ. Communications of the ACM, 2001. 44(10): p. 59-65.

[28] Kiczales, G., et al. Aspect-Oriented Programming. in Proceedings of the European Conference on Object-Proented Programming ECOOP’97. 1997: Springer Verlag.

[29] Kiczales, G., J.d. Riviéres, and D.G. Bobrow, The Art of the Metaobject Protocol. 1991, Cambridge, MA: MIT Press.

[30] Knuth, D.E., Semantics of context-free languages. Mathematical Systems Theory, 1968. 2: p. 127-145.

[31] Lieberherr, K., D. Orleans, and J. Ovlinger, Aspect-oriented programming with adaptive methods. Communications of the ACM, 2001. 44(10): p. 39-41.

[32] Lieberman, H., Using prototypical objects to implement shared behavior in object–oriented systems. SIGPLAN Notices, 1986. 21(11): p. 214-223.

[33] Lopes, C.V. and W.L. Hursch, Separation of Concerns, . 1995, College of Computer Science, Northeastern University, Boston, MA.

[34] Ossher, H., M. Kaplan, W. Harrison, A. Katz, and V. Kruskal, Subject-oriented Composition Rules. Proceedings of OOPSLA’95, ACM SIGPLAN Notices, 1995. 30(10): p. 235-250.

[35] Ostermann, K. and M. Mezini. Object-Oriented Composition Untangled. in Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA’2001). 2001.

[36] Smith, R.B. and D. Ungar, A Simple and Unifying Approach to Subjective Objects. Theory and Practive of Object Systems, 1996. 2(3): p. 161-178.

[37] Snyder, A. Encapsulation and inheritance in object-oriented programming languages. in Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA’86). 1986.

[38] Stein, A. Delegation is inheritance. in Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87). 1987.

[39] Tarr, P., H. Ossher, W. Harrison, and S.M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. in Proceedings of the International Conference on Software Engineering (ICSE'99). 1999.

[40] Ungar, D. and R.B. Smith, Self: the power of simplicity. Lisp and Symbolic Computation: An International Journal, 1991. 4(3): p. 45-55.

[41] VanHilst, M. and D. Notkin. Using role components to implement collabration-based designs. in Proceedings of OOPSLA’96. 1996.