MASPEGHI 2004

MechAnisms for SPEcialization, Generalization and inHerItance

Ph. Lahire, G. Arévalo, H. Astudillo, A.P. Black, E. Ernst,
M. Huchard, T. Opluštil, M. Sakkinen, P. Valtchev

(Affiliation and contact information given in Sections 2.1 and 2.2)

Abstract. MASPEGHI 2004 is the third edition of the MASPEGHI workshop. This year the organizers of both the ECOOP 2002 Inheritance Workshop and MASPEGHI 2003 came together to enlarge the scope of the workshop and to address new challenges. We succeeded in gathering a diverse group of researchers and practitioners interested in mechanisms for managing specialization and generalization of programming language components. The workshop contained a series of presentations with discussions as well as group work, and the interplay between the more than 22 highly skilled and inspiring people from many different communities gave rise to fruitful discussions and the potential for continued collaboration.

1 Introduction and Summary of the CFP

The MASPEGHI workshop took place on Tuesday, June 15th, at ECOOP 2004 in Oslo. It was the third edition of MASPEGHI (after OOIS 2002 and ASE 2003), but it was at the same time a follow-on to the *Inheritance Workshop* at ECOOP 2002 in Málaga (see Section 6) — a case of multiple inheritance. The meaning of the acronym MASPEGHI was modified from *MAnaging SPEcialization/Generalization HIerarchies* to *MechAnisms for SPEcialization, Generalization and inHerItance*, thus broadening the scope of the workshop.

MASPEGHI 2004 continued the discussion about mechanisms for managing specialization and generalization of programming language components. The workshop was organized around concepts such as inheritance and reverse inheritance, subclassing, and subtyping, and specialized into variants such as single or multiple inheritance, mixins, and traits.

The workshop was concerned with *(i)* the various *uses* of inheritance, and *(ii)* the difficulties of *implementation and control* of inheritance in practical applications. Several communities were represented, including those dealing with design methods, databases, knowledge representation, data mining, object-oriented programming languages, and modeling: each community addresses these concerns in different ways. Thus, one important goal of this workshop was to bring together a diverse group of participants to compare and contrast the use, implementation and control of inheritance as practiced in their communities.

This report summarizes the workshop. Section 2 lists the organizers, the participants and the written contributions. Section 3 provides an overview of the contributions and debates. Section 4 summarizes the outcome of the three work groups. We end this overview of the workshop with a conclusion and a list of pointers (Sections 5 and 6).

2 People and Contributions

2.1 Organizers

The organizers of this workshop were (in alphabetical order):

– **Gabriela Arévalo**: Software Composition Group, Institut für Informatik und angewandte Mathematatik, Bern, Switzerland. (`arevalo@iam.unibe.ch`)
– **Hernán Astudillo**: Departamento de Informática, Universidad Técnica Federico Santa María Valparaíso, Chile. (`hernan@acm.org`)
– **Andrew P. Black**: Dept. of Computer Science & Engineering, OGI School of Science & Engineering, Oregon Health & Science University (OGI/OHSU), Beaverton, USA. (`black@cse.ogi.edu`)
– **Erik Ernst**: Department of Computer Science, University of Aarhus, Denmark. (`eernst@daimi.au.dk`)
– **Marianne Huchard**: Laboratoire d'Informatique, de Robotique et Microélectronique de Montpellier (LIRMM), CNRS and Université de Montpellier 2, France. (`huchard@lirmm.fr`)
– **Philippe Lahire**: Laboratoire d'Informatique Signaux et Systèmes de Sophia Antipolis (I3S), Université de Nice Sophia antipolis, France. (`Philippe.Lahire@unice.fr`)
– **Markku Sakkinen**: Department of Computer Science and Information Systems, University of Jyväskylä, Finland. (`sakkinen@cs.jyu.fi`)
– **Petko Valtchev**: Dépt. d'Informatique et recherche opérationnelle (DIRO), Université de Montréal, Québec, Canada. (`petko.valtchev@umontreal.ca`)

2.2 Participants and Position Papers

A total of 22 persons participated in the workshop, although some of them only for part of the day. The attendees came from 13 different countries, the largest attendance (4) coming from France. Among them 15 were paper authors (see the table below) and/or members of the organizing committee. Five of the organizers were able to come: A. Black, E. Ernst, M. Huchard, Ph. Lahire and M. Sakkinen. All the papers in the following table are in the proceedings of the workshop [4], which is accessible from the website.

	Contribution	**Presenter** / ***Other authors***
(1)	Object Identity Typing: Bringing Distinction between Object Behavioural Extension and Specialization	D. Janakiram, Indian Institute of Technology Madras, India (`djram@lotus.iitm.ernet.in`) / *C. Babu*
(2)	A Reverse Inheritance Relationship for Improving Reusability and Evolution: the Point of View of Feature Factorization	Philippe Lahire (see above) / *C.-B. Chirila and P. Crescenzo*
(3)	Mathematical Use Cases lead naturally to non-standard Inheritance Relationships: How to make them accessible in a mainstream language	Marc Conrad, University of Luton, UK (`marc.conrad@luton.ac.uk`) / *T. French, C. Maple, and S. Pott*
(4)	Proposals for Multiple to Single Inheritance Transformation	Michel Dao, France Télécom R&D, France (`michel.dao@francetelecom.com`) / *M. Huchard, T. Libourel, A. Pons and J. Villerd*
(5)	The Expression Problem, Scandinavian Style	Erik Ernst (see above)
(6)	The Logic of Inheritance	DeLesley Hutchins, University of Edinburgh, UK (`D.S.Hutchins@sms.ed.ac.uk`)
(7)	An anomaly of subtype relations at component refinement and a generative solution in C++	Zoltán Porkoláb, Eötvös Loránd University, Hungary (`gsd@elte.hu`) / *I. Zólyomi*
(8)	Java with Traits — Improving Opportunities for Reuse	Philip J. Quitslund, Oregon Health and Science University, USA (`philipq@cse.ogi.edu`) / *A. P. Black*
(9)	Merging conceptual hierarchies using concept lattices	Marianne Huchard (see above) / *M. H. Rouane, P. Valtchev, P. and H. Sahraoui*
(10)	Behaviour consistent Inheritance with UML Statecharts	Markus Stumptner, University of South Australia, Australia (`mst@cs.unisa.edu.au`) / *M. Schrefl*

(11)	Domain Modeling in Self Yields Warped Hierarchies	Ellen Van Paesschen, Vrije Universiteit Brussel, Belgium (`evpaessc@vub.ac.be`) / *W. De Meuter and T. D'Hondt*
(12)	Inheritance Decoupled: It's More Than Just Specialization	L. Robert Varney, University of California at Los Angeles, USA (`varney@cs.ucla.edu`) / *D. S. Parker*

Among the other participants were the following (alphabetically):

- Antoine Beugnard, Ecole Nationale Supérieure de Télécommunication, France (`Antoine.Beugnard@enst-bretagne.fr`)
- Kim Bruce, Williams College, Massachusetts, USA (`kim@cs.williams.edu`)
- Sebastián González, Université catholique de Louvain, Belgium (`sgm@acm.org`)
- Håvard Hegna, Norwegian Computing Center, Norway (`hegna@nr.no`)
- Tomáš Opluštil, Charles University in Prague, Czech Republic (`oplustil@nenya.ms.mff.cuni.cz`)
- Wilfried Rupflin, University of Dortmund, Germany (`Wilfried.Rupflin@uni-dortmund.de`).

3 Workshop and Contribution Overview

3.1 Workshop Organization

The organizers prepared for the workshop by a quite lengthy process of characterizing and classifying the papers, based on their main topics. In this process it turned out to be useful to apply techniques from concept analysis, which is a core research area for some of the organizers. Here is an early version of the classification[1]; note that some papers match several topics.

- Contradiction between a desired subtyping or specialization relation and available language mechanisms. What should designers and developers do when a desired subtyping relation cannot be expressed in the particular technology (e.g., programming language) employed to create the software? Papers *P6*, *P7*, *P3*, *P1*, *P11*, *P8*, *P12*, *P5*, *P9*, *P2* deal with this topic. They propose, roughly, to rearrange the desired hierarchy to fit the language, or to design a new language.
- Is class composition worthwhile? One example of a class composition mechanism is multiple inheritance, and it is well-known that multiple inheritance is hard to do well. People who think class composition is worthwhile emphasize that it is powerful, and the more sceptical people emphasize that the resulting software is complex and hard to maintain. Papers *P11*, *P8*, *P12*, *P5*, *P4* are related to this topic.

[1] In this report, papers presented at the workshop are referred to as *Pn* where *n* is the number of the paper in the table in Section 2.2

– Different kinds of subclassing relationships. How many kinds of inheritance relationships are needed? How many kinds does *your* technology have? Papers *P10*, *P12*, *P4*, *P2* deal with this topic.
– Form and Transform, Dealing with evolution. How can methodologies, languages and tools help us to deal with classification, construction and evolution? Papers *P3*, *P4*, *P9*, *P2* deal with this topic.

This process of establishing an overview of the issues and positions represented by the papers continued, and at the workshop we ended up with three sessions:

1. Form and Transform: Dealing with Evolution (papers *P3*, *P2*, *P4*, *P10*).
2. Class composition (papers *P11*, *P8*, *P5*).
3. Contradiction between a desired subtyping or specialization relation and available language mechanisms (papers *P6*, *P7*, *P1*, *P12*).

The workshop started with a brief welcome and the introduction of the participants. The three sessions were organized as presentations of the position papers followed by discussion, applying a flexible attitude to timekeeping that prioritized the contents of the discussions rather than adhering rigidly to a schedule. Each presentation lasted about 10 minutes; following that, an *opponent*—who prepared by carefully studying the paper and other related material—initiated the discussion by asking questions, making comments or proposing an alternate point of view. Gradually, the other participants would also ask questions or make comments. Some ingenuity was needed to schedule this activity around lunch and coffee breaks, but the flexible approach to timing worked quite well.

We now turn to the conduct of the three sessions listed above.

3.2 Session 1: Form and Transform: Dealing with Software Evolution

This session dealt with the evolution of designs and of software; papers *P4* and *P10* addressed the design level with UML whereas *P2* and *P3* addressed the programming level. Paper *P4* focused on the use of meta-information, categorizing applications of multiple inheritance according their semantics, and then using this categorization to select a suitable transformation to single inheritance.

Paper *P10* deals with object life-cycles represented with UML statechart diagrams. Inheritance is used to specialize life-cycles, that is, to extend and refine them. The semantics of this kind of inheritance relationship relies on two properties: observation consistency and invocation consistency [42]. Paper *P3* investigates how method renaming, dynamic inheritance and interclassing can be used to strengthen the relationships between mathematical reasoning (algebric structuring) and object-oriented techniques [15]. This led to a discussion about benefits and advantages of introducing these ideas within OO languages. The last paper of the session, *P2*, deals with the introduction of a reverse inheritance relationship to better address the reuse and evolution of hierarchies of classes. This implies the existence of a language that provides both specialization and generalization relationships [20]. The paper introduces a factorization mechanism that enables a programmer to move features up the hierarchy. A discussion ensued about the semantics that should be attached to generalization relationships.

3.3 Session 2: Class Composition

The second session included presentations of three papers that addressed this topic, but based in the culture of three different languages — Self [47,1], Java and gbeta [24].

In *P11* the authors demonstrate that the hierarchies required for proper domain modeling are the reverse of the hierarchies required by the Self programming language for the proper execution of the corresponding code. Self uses a particular kind of prototype object, called a trait object, as a way of sharing behavior. A variation of this idea is explored in paper *P8*, which led to some discussion on this topic. The paper deals with a mechanism for reusing code in Java, based on previous work on traits in Smalltalk [41]. One of the common ideas is that the class is not the best unit of reuse; the authors demonstrate this through a detailed study of code duplication in the Java Swing library.

The third paper, *P5*, is influenced by the expressiveness of the gbeta language and explains how higher-order hierarchies [25] can be used to solve the expression problem [46]. One of the main advantages of gbeta is that it makes it possible to adapt and evolve *whole hierarchies* of classes rather than individual classes. A discussion dealing with other possible solutions to the expression problem, especially reverse inheritance, followed the presentation of the paper.

3.4 Session 3: Subtyping and Specialization

The third session dealt with incompatibilities between the subtyping and specialization relations and the available mechanisms, and involved four papers. In *P6* the author argues that inheritance is fundamentally concerned with the categorization of objects, and that OO languages should thus be founded upon a formalism that supports categorical reasoning. He proposes a formal language called SYM, which is aimed at representing class/object types in a way that avoids classical inheritance problems such as conflict resolution and the dichotomy between subtyping and implementation inheritance. During the discussion we noted that classes in SYM are like traits or mixins [7,26] and that SYM enables the handling of both virtual methods and Beta-style virtual classes.

Paper *P7* describes a limitation of inheritance that the authors call the *chevron shape anomaly*. It is based on the fact that (i) classes in a hierarchy may be extended by inheritance in order to add new functionality and (ii) an application may use several hierarchies and use them at different levels. The authors explain that it implies an increase of complexity and propose a solution based on generative programming [50]. Paper *P1* is concerned with the expressiveness of inheritance in conventional OO languages, which do not make a clear distinction between object behavioral extension (which needs to preserve object identity) and behavioral specialization (where a new object is created). The authors propose to capture this distinction by representing object identity as a type. The paper *P12* pointed out another deficiency of object-oriented languages: that they do not provide sufficient support for interface abstraction and implementation inheritance, thus spreading implementation bias and impairing evolution. To address these issues, the authors propose interface-oriented programming (IOP) [49], which decouples the client of an abstraction from the code that binds it to a specific implementation and provides an interface-oriented form of inheritance that keeps implementation bias in check and is useful for both specialization and adaptation.

3.5 Group Discussions

An important part of the workshop was the group discussions held in the afternoon. Three work groups were formed; the topics of the workgroups reflect the interests of participants. They were largely derived from the session topics: two of them came directly from session topics, whereas the third was formed during the earlier discussions. The topics were as follows:

– Composition of classes
– Subtyping and subclassing
– Inheritance relations applied to components

After one hour of discussion, one representative from each group (Erik Ernst, Andrew Black and Marianne Huchard, respectively) explained to the other participants the perspectives of their groups and the result of the discussion. The next section summarizes these discussions; the summary from each work group is written by its participants, and organized by the group representatives above.

4 Summary of Group Discussions

In the following subsections we describe the working groups held during the afternoon.

4.1 Composition of Classes

The members of this group were Marc Conrad, Erik Ernst, Philippe Lahire, Philip Quitslund, and Markku Sakkinen. It quickly became clear that nobody in the group was vehemently against class composition, even though they acknowledged what Alan Snyder said many years ago: "multiple inheritance is good but there is no good way to do it" (reported by Steve Cook [16]).

Consequently, we implicitly responded to the question of whether class composition is worthwhile with a 'Yes!', qualified by the realization that there will probably always be wrinkles in the design of each concrete class composition mechanism, and then continued to explore the similarities and differences between our approaches to it.

One line of exploration was to find features of each approach that other approaches could not readily match. For traits, represented by Philip, the feature we selected was the symmetry of trait composition: two traits may both import and export from each other, thus satisfying the requirements of both of them. In contrast, with mixins the dependency is strictly unidirectional. Symmetric dependencies enable the creation of composite entities, e.g., classes created by composing traits, in a more flexible manner than is possible with strict unidirectional dependencies.

The selected feature of gbeta, represented by Erik, was that of composing nested entities, e.g., families of classes or even families of families of classes, and having the composition propagate recursively into the structure. This enables disciplined and well-defined composition of many classes in parallel with a very concise syntax. In contrast, a single class composition mechanism is generally more error-prone and typically lacks the ability to ensure compatibility among many classes.

Reverse inheritance, which is a main topic in the paper by Philippe and also the subject of earlier work by Markku under the name exheritance, is unique in that it allows for non-intrusive modification of existing classes (i.e., changing their meaning without editing them). The precise scope of this kind of modification depends very much on the details of the mechanism, but generally it enables addition of new supertypes to existing classes even in a type system based on name equivalence, and some kinds of inverse inheritance or exheritance allows for semantically significant changes, too, such as overriding an inherited method or even adding state to the specified subclasses. Non-intrusive modification improves on the flexibility in system development, especially where large amounts of existing source code must be modified, but cannot be edited.

It is difficult to evaluate the quality of programming language mechanisms because this would ideally require that we look at *all* programs that could ever be written using the mechanism and evaluate whether those programs would be of higher quality without the use of the mechanism, or using an alternative version of it. Obviously, no such thing could ever be done or even approximated. Hence, evaluation of language mechanisms tends to be informal. However, as Marc pointed out, one might use things like design patterns [27] in an evaluation, because patterns represent well-known and hard problems in software design at the level of a few classes—which is often the level where language mechanisms for class composition are most relevant.

Finally, we discussed an inversion scenario for the unfolding of software as a vehicle to gain insight into the real nature of class composition and other abstraction mechanisms. Software is unfolded in the following sense: designers and developers create abstractions such as classes, subclasses, type parameterized classes or methods, etc. We may consider inheritance as a short-hand for repeating the declarations inherited from superclasses, and similarly for type parameterization, so the most sophisticated abstractions could be 'unfolded away', leaving us with a simple, flat universe of classes with no inheritance or type parameters, etc. In fact, this would typically yield a correct description of actual objects at run-time and their behavior.

Now imagine that we start from the other end, with a running system of objects and behavior (with no classes or other abstractions defined *a priori*, as in Self [47]). We could then examine which objects and behaviors are similar, and construct classes and methods to describe them; next we could explore similarities between classes and use them to build inheritance hierarchies, etc.—which is one of the things that Marianne Huchard and others are doing with concept analysis. We would then reconstruct the abstractions from the run-time environment, as opposed to constructing the run-time entities from the abstractions. The latter is an unfolding process, whereas the former is a folding or 'compression' process.

The intriguing insight is that the run-time world can be considered as the primary artifact, with the abstractions as derived entities—just the opposite of typical thinking for class based languages, especially statically typed ones. The very thought that abstractions may be constructed automatically may help to make class composition and other mechanisms more lightweight and less intimidating, and similarly refreshing is the idea that manipulations of a "program" could sometimes take place at the concrete level, with new abstractions arising by subsequent (more or less automatic) analysis of the concrete level.

Of course, this thesis about the wonders of concreteness is immediately followed by an antithesis: abstraction is one of our most powerful tools and hence abstractions should not be reduced to mere implementation details produced by a programming tool. However, abstractions might be more manageable if the top-down unfolding point of view is supplemented with the bottom-up folding point of view.

4.2 Subtyping and Subclassing

The members of this working group were Andrew P. Black, Kim Bruce, DeLesley Hutchins, D. Janakiram, Zoltán Porkoláb, Markus Stumptner and L. Robert Varney. The group focused on the problem of what to do when a programmer finds that it is convenient to subclass an existing class, not to capture a specialization relationship in the domain being modeled, but just to share implementation. It can be argued that this is a bad programming practice, but often the only practical alternative is the wholesale use of copy and paste, which is surely a worse programming practice. William Cook's examination of the Smalltalk Collection Classes [18] shows that the practice is common.

However, this activity can lead to problems. The sub*typing* relation that does capture the specialization relationship of the domain is at best obscured, and at worst destroyed. For example, in Smalltalk it is obscured: two classes A and B that happen to be defined in completely different parts of the subclass hierarchy, but with sets of methods such that B is a subtype of A, have the property that aB can be substituted for anA. But this property is obscure: it is not expressed explicitly in the program. In Java, the interface construct and the implements keyword let the programmer say explicitly that A and B both implement a common interface, let us call it the "I" interface: this makes the programmer's intention explicit. But Java has its shortcomings too, because Java insists that any subclass is a subtype of its superclass, whether or not this makes sense in the context of the application domain. Moreover, if A was defined by someone else, perhaps someone working for another company, *without* also stating that it implements the I interface, then when another programmer comes along and tries to define and use aB in place of anA, the program won't compile. In order to substitute aB, not only must A implement I, but all declarations of parameters and variables must use I rather than A.

The goal of the group was to consider this problem and possible solutions. There was agreement that this is fundamentally a problem of language design: a single mechanism (inheritance) is made to play too many roles, with the result that programs are harder to understand. That is, the reason for the use of an inheritance relationship is not explicit in the program text, and the reader must try to infer it.

The problem (outlined above) with Java programs that do not use interfaces could be solved in a backwards-compatible way by a small modification to the language semantics. The idea is to create for each class C an interface with the same name, and to interpret variable and parameter declarations involving C as referring to the interface name rather than to the class name. It would then be possible for a maintenance programmer to define a new class that **implements** C, and instances of this new class could then be used in places that require a C. However, interfaces do have a run-time cost in Java, and this patch would impose that cost on every program. A non-compatible change to Java would involve separating the subtyping (interface) and subclassing hierarchies entirely; this would also make it possible to allow non-subtype-compatible changes in a subclass, such as canceling methods.

An alternative approach is to find another mechanism for code reuse, thus freeing inheritance to be used solely for domain modeling, which several of the participants at the workshop had argued is the primary role of inheritance (see for example papers *P3* and *P6*). The trait concept described in *P8* is one candidate for such a mechanism. As currently implemented, traits do not subsume inheritance because they do not allow the declaration of instance variables. However, it seems that an extended trait mechanism without this restriction would provide all of the reuse opportunities offered today by inheritance, as well as others, but without implying any conceptual classification that might not be intended. An implicit or explicit subtyping system could then be used for classification.

4.3 Inheritance Relations Applied to Components

This group consisted of Michel Dao, Marianne Huchard, Ellen Van Paesschen, Antoine Beugnard, Sebastián González, and Tomáš Oplučtil. It was established because, although the (mainly academic) research in the field of software architecture and component systems has become mature, not much attention has been devoted to the study of emerged high-level abstractions from the point of view of inheritance relations. This becomes even more important in the context of model transformations (see, e.g., OMG MDA [3]). Therefore the long-term goal of this group is to set up a basis for research on inheritance in architecture description languages (ADLs) and component systems.

The discussion was initiated by Tomáš Opluštil who presented some ongoing research aimed at introducing inheritance in SOFA CDL [35,36]. This initial discussion resulted in the following list of key long-term goals (of which only the first two were discussed because of time limitations).

– Selecting/defining abstractions in component models and ADLs to which inheritance or specialization can be applied.
– Defining the terms subtyping, specialization and inheritance in the context of components and other higher-level abstractions.
– Proposing purposes for which inheritance should be used in component models and ADLs.
– Proposing corresponding relations in implementation languages (into which inheritance in higher-level abstractions should be mapped).

The main abstractions in component models, which are thus *a priori* potential candidates for specialization, reuse inheritance or subtyping, are the following: a *component* is a unit of computation (often both a design-time and runtime entity); an *interface* is roughly a set of operations; a *component type* is a set of component interfaces; a *connector* manages communication between components; an *architectural configuration* is a set of components and connections. Components have *ports* or *component interfaces*, which usually characterize the type of the component. A principle distinction is between *client* and *provided* component interfaces which draw required and provided services; additional classifications can be by contingency (optional/mandatory) and cardinality (singleton/collection) as in Fractal [11].

We started with common definitions and uses of subtyping, inheritance and specialization in object-oriented programming and modeling languages, and initiated a discussion about their interpretation in the case of component models.

Subtyping. Static type systems are a way to limit runtime errors in programming languages, mainly preventing inappropriate operations to be called on entities. Subtyping is usually based on the substitutability notion [21]: “a type t_2 conforms to a type t_1 iff any expression of type t_2 can be substituted for (bound to) any variable of type t_1 without any runtime error”. Substitutability on classes is ensured by invariant redefinition of attributes and redefinition of methods with covariant (more specific in the subtype) return type and contravariant (more general in the subtype) parameter types. Most of the abstractions listed above that are involved in component systems can be also handled as types, and thus candidates for subtyping. Our attention was primarily focused on interfaces and components. With interfaces, which are usually sets of operation signatures, the same policy that applies to subtyping for classes in class-based OOPLs can apply to interfaces in component systems. For component types (in the Fractal terminology: sets of provided and client component interfaces), substitutability can be understood as the possibility of replacing a component by another without changing the environment (components and bindings) [11]. Errors that we would like to avoid during this substitution may include plugging-binding errors (when a component interface is missing), or invocation of non-existent services and services with bad signatures. Subtyping is usually guided by the idea of providing more services and requiring less services; covariant policy should apply to provided services and a contravariant policy to required services [11,19]. However, some researchers argue (in as yet unpublished papers) that this notion of subtyping may fail in some special cases, in which a substitution may lead to halting communication in the component system. Therefore alternative approaches to the definitions of subtyping are being introduced; a promising one is based on behavior protocols [38].

Specialization. In object-oriented modeling, class specialization is defined by inclusion of the instance sets (or extensions). Specialization hierarchies should reflect usual domain classifications. When a class C_2 specializes another class C_1, a consequence is that properties of C_1 are inherited by C_2, with possible refinements [21]. The fact that components are intrinsically generalizable elements can be demonstrated by the case of the UML meta-model [34]: metaclass Component is a subclass of Class in UML 2.0 meta-model.

Inheritance. In the object-oriented context, inheritance is a mechanism that allows a class to own (inherit) properties (mainly methods and attributes) of another class. The subclass can specialize, redefine and even cancel the inherited properties. Unlike OOPLs, current proposals for components do not provide much room for inheritance in their design, e.g., in ComponentJ [19] the language designers argue that in the presence of inheritance the result of method invocation (with dynamic binding) is dependent on the class hierarchy, making it difficult to define well-encapsulated pieces of software (which should be independent deployment units). As a result, emphasis is put on aggregation and component sharing rather than on inheritance. However, inheritance is in fact really useful for creating new component definitions by extending or merging existing ones as proposed in Fractal [10] and SOFA [36].

In the case of object-oriented programming and modeling languages, subtyping, inheritance and specialization are unfortunately mixed: inheritance is used to reflect domain specialization for clarity's sake in programs; types are often identified with classes; and subclassing is constrained by type safety [21]. Invariance policy in property redefinition, although too restrictive, is still the usual rule. We can hope that component models will be more careful in their interpretation of such notions: the best choice would be to define these relations independently avoiding all confusion.

We concluded that the adaptation of relations and techniques used in object-oriented languages to the context of component systems provides a lot of room for further research — new, higher-level abstractions, introduced in these systems, can bring on new uses for and give new meanings to the “old” well-explored relations of the object-oriented world.

5 Conclusion

The contents of both the written contributions and the debates described in this document showed that during this workshop we addressed number of interesting topics. Of course we were not able to go into the details of all of them here, but nevertheless feel that this report captures the atmosphere and scope of the workshop. Both modeling and language levels were covered, and the issues of evolution, adaptation and transformation, as well as reusability and type-safety, were given especial emphasis.

The contributions of the participants, and in particular the lively discussion that pervaded the workshop, convince us that workshops where the paper is only the starting point for the exchange of ideas are more profitable than mini-conferences that emphasize presentations of papers. Participants have now enough knowledge on the work of others to think about some collaboration. The working groups which had been set after the three sessions (even though only the subject had been set at this time) already provide some early information on the kind of collaboration that is starting.

A mailing list and a website will be maintained to ensure continuous discussion and visibility even after the end of the workshop.

– **Mailing list**: `maspeghi-ecoop2004@i3s.unice.fr`
– **Website**: `http://www.i3s.unice.fr/maspeghi2004`

6 Pointers to Related Work

The reader wishing to delve more deeply into the topic of this workshop might do well to start with the proceedings of the previous workshops dedicated to inheritance [37,40,5], to specialization or generalization [9,48] and to object classification [31,30]. Several Ph.D theses have also been written on these topics, including references [6,17,24,32,44]; the books by Gamma [27] and Meyer [33] are also a useful starting point. Papers of particular interest include references [45,23,28,29,2,39,22,43,8,13,14,12].

Related workshops
(workshop name, associated conference, number of participants and Web site):

– Maspeghi 2002 - OOIS 2002 - 15 persons.
  `http://www.lirmm.fr/~huchard/MASPEGHI/`
– Maspeghi 2003 - ASE 2003 - 14 persons.
  `http://www.iro.umontreal.ca/~maspeghi/`
– Inheritance 2002 - ECOOP 2002 - 27 persons from ten countries (15 were authors or coauthors of an accepted paper).
  `http://www.cs.auc.dk/~eernst/inhws/`

References

1. O. Agesen, L. Bak, C. Chambers, , B.-W. Chang, U. Hölzle, J. Maloney, R. B. Smith, D. Ungar, and M. Wolczko. *The Self 4.0 Programmer's Reference Manual.* Sun Microsystems, Inc., Mountain View, CA, 1995.
2. O. Agesen, J. Palsberg, and M. I. Schwartzbach. Type inference of SELF: Analysis of objects with dynamic and multiple inheritance. In O. Nierstrasz, editor, *Proceedings of ECOOP'93*, LNCS(707), pages 247–267, Kaiserslautern, Germany, July 1993. Springer-Verlag.
3. Architecture Board ORMSC1. *Model Driven Architecture (MDA), document number ormsc/01-07-01.* Object Management Group, July 2001. `http://www.omg.org/docs/ormsc/01-07-01.pdf`.
4. G. Arévalo, H. Astudillo, A. P. Black, E. Ernst, M. Huchard, P. Lahire, M. Sakkinen, and P. Valtchev, editors. *Proceedings of the 3rd International Workshop on "Mechanisms for Specialization, Generalization and Inheritance" (MASPEGHI'04) at ECOOP'04.* University of Nice - Sophia Antipolis, Oslo, Norway, June 2004.
5. A. P. Black, E. Ernst, P. Grogono, and M. Sakkinen, editors. *Proceedings of the Workshop "Inheritance" at ECOOP'02.* Number 12 in Publications of Information Technology Research Institute. University of Jyväskylä, Málaga, Spain, June 2002.
6. G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* Ph.D. thesis, Dept. of Computer Science, University of Utah, Mar. 1992.
7. G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of OOPSLA/ECOOP'90*, volume 25(10) of *ACM SIGPLAN Notices*, pages 303–311, Ottawa, Canada, October 1990. ACM press.
8. G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, pages 282–290, Washington, DC, Apr. 1992. IEEE Computer Society.
9. J.-M. Bruel and Z. Bellahsène, editors. *Advances in Object-Oriented Information Systems: OOIS 2002 Workshops.* LNCS(2426). Springer Verlag, Montpellier, France, September 2002.
10. E. Bruneton. *Fractal ADL tutorial 1.2.* France Telecom R&D, March 2004. http://fractal.objectweb.org.
11. E. Bruneton, T. Coupaye, and J.-B. Stefani. The Fractal component model. Specification. Draft, France Telecom R&D, February 2004. http://fractal.objectweb.org.
12. L. Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types, International Symposium Sophia-Antipolis Proceedings*, LNCS(173), pages 51–67. Springer-Verlag, June 1984.
13. L. Cardelli. Structural subtyping and the notion of power type. In *POPL '88. Proceedings of the conference on Principles of programming languages*, pages 70–79, San Diego, CA, USA, January 1988. ACM Press.
14. L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):480–521, Dec. 1985.
15. M. Conrad and T. French. Exploring the synergies between the object-oriented paradigm and mathematics: a Java led approach. *International Journal on Education Sciences and Technology*, 2004. to appear.
16. S. Cook. OOPSLA '87 Panel P2: Varieties of inheritance. In *OOPSLA '87 Addendum To The Proceedings*, volume 23(5) of *ACM SIGPLAN Notices*, pages 35–40, Orlando, FL, USA, October 1987. ACM Press.

17. W. R. Cook. *A Denotational Semantics of Inheritance.* PhD thesis, Brown University, 1989.
18. W. R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *Proceedings of OOPSLA'92*, volume 27(10) of *ACM SIGPLAN Notices*, pages 1–15, Vancouver, Canada, October 1992. ACM Press.
19. J. Costa Seco and L. Caires. A basic model of typed components. In E. Bertino, editor, *Proceedings of ECOOP'00*, LNCS(1850), pages 108–128, Cannes - Sophia Antipolis, France, June 2000. Springer Verlag.
20. P. Crescenzo and P. Lahire. Using both specialisation and generalisation in a programming language: Why and how? In Bruel and Bellahsène [9], pages 64–73.
21. R. Ducournau. "Real World" as an argument for covariant specialization in programming and modeling. In Bruel and Bellahsène [9], pages 3–12.
22. R. Ducournau, M. Habib, M. Huchard, and M.-L. Mugnier. Monotonic conflict resolution mechanisms for inheritance. In *Proceedings of OOPSLA'92*, volume 27(10) of *ACM SIGPLAN Notices*, pages 16–24, Vancouver, Canada, October 1992. ACM press.
23. R. Ducournau, M. Habib, M. Huchard, and M.-L. Mugnier. Proposal for a monotonic multiple inheritance linearization. In *Proceedings of OOPSLA'94*, volume 29(10) of *ACM SIGPLAN Notices*, pages 164–175, Portland, Oregon, USA, October 1994. ACM press.
24. E. Ernst. *gbeta – A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance.* PhD thesis, Devise, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 1999.
25. E. Ernst. Higher-order hierarchies. In L. Cardelli, editor, *Proceedings of ECOOP'03*, LNCS(2743), pages 303–329, Darmstadt, Germany, July 2003. Springer-Verlag.
26. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, San Diego, California, 19–21 Jan. 1998.
27. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, USA, 1995.
28. R. Godin and H. Mili. Building and maintaining analysis-level class hierarchies using Galois lattices. In *Proceedings OOPSLA'93*, volume 28(10) of *ACM SIGPLAN Notices*, pages 394–410, Washington, DC, USA, 1993. ACM press.
29. F. J. Hauck. Inheritance modeled with explicit bindings: An approach to typed inheritance. *ACM SIGPLAN Notices*, 28(10):231–239, Oct. 1993.
30. M. Huchard, R. Godin, and A. Napoli. Objects and classification. In J. Malenfant, S. Moisan, and A. Moreira, editors, *ECOOP'00 Workshop reader*, LNCS(1964), pages 123–137, Cannes - Sophia Antipolis, France, 2000. Springer-Verlag.
31. M. Huchard, R. Godin, and A. Napoli, editors. *Proceedings of the workshop "Objects and Classification: a Natural Convergence" at ECOOP'00.* Loria, University of Nancy, Sophia-Antipolis, France, June 2000.
32. G. Kniesel. *Dynamic Object-Based Inheritance with Subtyping.* PhD thesis, Computer Science Department III, University of Bonn, July 2000.
33. B. Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, N.Y., second edition, 1997.
34. OMG. *Unified Modeling Language (UML) Superstructure - Final Adopted specification.* Object Management Group, August 2003. Version 2.0.
35. T. Opluštil. Inheritance in SOFA components. Master thesis, Faculty of Informatics, Masaryk University, Brno, Czech Republic, 2002.

36. T. Opluštil. Inheritance in architecture description languages. In J.Šafránková, editor, *Proceedings of the Week of Doctoral Students conference (WDS 2003)*, pages 118–123, Prague, Czech Republic, 2003. Charles University, Matfyzpress.
37. J. Palsberg and M. I. Schwartzbach, editors. *Proceedings of the Workshop "Types, Inheritance and Assignments" at ECOOP'91*, DAIMI PB-357, Geneva, Switzerland, July 1991. Computer Science Department, Aarhus University.
38. F. Plášil and S. Višňovský. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11), November 2002.
39. M. Sakkinen. A critique of the inheritance principles of C++. *Computing Systems*, 5(1):69 – 110, Winter 1992.
40. M. Sakkinen, editor. *Proceedings of the Workshop "Multiple Inheritance and Multiple Subtyping" at ECOOP'92*, Working Paper WP-23, Utrecht, the Netherlands, June 1992. Department of Computer Science and Information Systems, University of Jyväskylä.
41. N. Schaerli, S. Ducasse, O. Niestrasz, and A. P. Black. Traits: composable units of behaviour. In *Proceedings of ECOOP'03*, LNCS(2743), pages 248–274, Darmstadt, Germany, June 2003. Springer-Verlag.
42. M. Schrefl and M. Stumptner. Behavior consistent specialization of object life cycles. *ACM Transactions on Software Engineering and Methodology*, 11(1):92–148, 2002.
43. C. A. Szyperski. Import is not inheritance - why we need both: Modules and classes. In O. L. Madsen, editor, *Proceedings of ECOOP'92*, LNCS(615), pages 19–32, Utrecht, The Netherlands, June 1992. Springer Verlag.
44. A. Taivalsaari. *A Critical View of Inheritance and Reusability in Object-Oriented Programming*. PhD thesis, University of Jyväskylä, 1993.
45. F. Tip and P. F. Sweeney. Class hierarchy specialization. In *Proceedings of OOPSLA'97*, volume 32(10) of *ACM SIGPLAN Notices*, pages 271–285, Atlanta, Georgia, USA, 1997. ACM press.
46. M. Torgersen. The expression problem revisited. In M. Odersky, editor, *Proceedings of ECOOP'04*, LNCS(3086), pages 123–143, Oslo, Norway, June 2004. Springer-Verlag.
47. D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings of OOPSLA'87*, volume 22(12) of *ACM SIGPLAN Notices*, pages 227–242, Orlando, FL, USA, Oct. 1987. ACM press.
48. P. Valtchev, H. Astudillo, and M. Huchard, editors. *Proceedings of the workshop "Managing Specialization/Generalization Hierarchies" at ASE 2003*. DIRO, University of Montreal, Montreal, Quebec, Canada, October 2003.
49. L. R. Varney. Interface-oriented programming. Technical Report TR-040016, UCLA, Department of computer science, 2004.
50. I. Zólyomi, Z. Pórkoláb, and T. Kozsik. An extension to the subtype relationship in C++. In *Proceedings of GPCE'03*, LNCS(2830), pages 209–227, Erfurt, Germany, July 2003. Springer-Verlag.