WS11. The Inheritance Workshop

Gabriela Arévalo[1], Andrew Black[2], Yania Crespo[3], Michel Dao[4], Erik Ernst[5],
Peter Grogono[6], Marianne Huchard[7], and Markku Sakkinen[8]

[1] Software Composition Group, U. Bern, Neubrückstrasse 10, 3012 Bern, Switzerland
`arevalo@iam.unibe.ch`
[2] OGI School of Sci. & Eng., Oregon Health & Sci. Univ., Beaverton, Oregon, USA
`black@cse.ogi.edu`
[3] Dpto. Informática, U. de Valladolid, Campus M. Delibes, Valladolid 47011, Spain
`yania@infor.uva.es`
[4] France Télécom R&D DAC/OAT, 92794 Issy Moulineaux Cedex 9
`michel.dao@francetelecom.com`
[5] Department of Computer Science, University of Aarhus, Denmark
`eernst@daimi.au.dk`
[6] Dept. of Computer Science, Concordia University, Montreal, Canada H3G 1M8
`grogono@cs.concordia.ca`
[7] LIRMM (CNRS-University Montpellier 2), 161, rue Ada, 34392 Montpellier cedex 5
`huchard@lirmm.fr`
[8] Information Technology Research Institute, University of Jyväskylä, Finland
`sakkinenm@acm.org`

1 Introduction

The Inheritance Workshop at ECOOP 2002, which took place on Tuesday, 11 June, was the first ECOOP workshop focusing on inheritance after the successful workshops in 1991 [41] and 1992 [48]. The workshop was intended as a forum for designers and implementers of object-oriented languages, and for software developers with an interest in inheritance. It was organized by Andrew P. Black, Erik Ernst, Peter Grogono, and Markku Sakkinen.

Because of the size and diversity of the field, it is hard to come up with a litmus test for "object orientation", but one of the most widely accepted ingredients is inheritance. Indeed, in his 1987 characterization of the language design space [58], Wegner made inheritance one of the two defining characteristics of object-orientation.

Nevertheless, inheritance remains an active research area, because of problems like fragile base classes, the so-called inheritance anomaly, and the lack of encapsulation between a class and its subclasses. We believe the abundant activity demonstrates that inheritance is both hard to avoid and hard to get right. The goal of this workshop was to advance the state of the art in the design of inheritance mechanisms, and the judicious use of inheritance.

The number of submissions confirmed the interest in this topic. We accepted 15 short position papers, written by a total of 28 authors from 11 different countries. We had particularly solicited reports from practitioners, but received contributions only from researchers. However, they represent so many different approaches and viewpoints that the workshop became a valuable forum for cross-fertilization of ideas.

The papers can be roughly classified as follows: language design and language constructs [24, 27, 40, 49, 50, 56]; analysis and manipulation of inheritance hierarchies [1, 19, 22, 29]; generalization in UML models [42]; language usage [5]; role models [51]; metaprogramming [16]; partial evaluation [7].

The submitted papers were reviewed by the workshop organizers, although not formally refereed, and the accepted papers published in the workshop proceedings [6] were revised by the authors in the light of these reviews, with a length limit of 7 pages. As real workshop papers, they are mostly less complete and finished than conference papers would be, but we believe that they compensate for this lack of polish by providing access to fresh ideas and ongoing work. We found that every paper had some interesting ideas, and we thank all authors for their contributions.

In addition to these papers, we were happy to have Gilad Bracha (Sun Java Software) as an invited speaker. His talk was entitled "Mixins in Strongtalk" [2]. It was not possible to publish the paper in the proceedings, but copies were available at the workshop.

The website of the workshop is still accessible:
`http://www.cs.auc.dk/ eernst/inhws/`.
Both the papers from the proceedings and the invited paper are available there, or directly at: `http://www.cs.jyu.fi/ sakkinen/inhws/papers/`.

According to the list that was collected at the workshop, there were 27 persons present, 15 of whom were authors of workshop papers. The attendees came from 10 different countries, the largest attendance (5) coming from France. The authors of 4 accepted papers were not able to attend the workshop.

Only 9 papers were selected for oral presentations at the workshop, in order to have more time for discussion. After each paper another workshop participant presented a short comment prepared in advance. These presentations took the first half of the day.

The afternoon sessions started with the invited talk. After that, we spent about two and one half hours in group discussions in three breakout groups. We came together again for a final one hour plenary session in which the groups tried to summarize their findings.

As so often happens, the day appeared to be too short for all the topics that we would like to have discussed. There was a common feeling that an inheritance-related workshop would be welcome also at some future ECOOP, perhaps as soon as 2003 if there are active organizers. We felt that even a somewhat more restrictive topic could attract sufficient participation. There is a mailing list that can be used for such suggestions:
`http://majordomo.cc.jyu.fi/mailman/listinfo/inheritance-ecoop`.

The rest of this report is divided into two parts, namely Sect. 2 which describes the outcome of the discussions in the hierarchy manipulation subworkshop, and Sect. 3 which describes the outcome of the discussions in the mixins subworkshop. The third group discussed *dynamism*, but did not produce written results for this report.

2 Hierarchy Manipulation

An object oriented program is typically organized as a hierarchy of classes. Structurally, the hierarchy may be a tree, a forest, or a directed acyclic graph. Semantically, the hierarchy may be concerned with:

- *Specialization*: the class hierarchy is guided by a classification of concepts of the application domain (close to an ontology);
- *Subtyping*: in a type hierarchy, a type T_1 is a subtype of T_2 if an object of T_1 is always substitutable to an object of T_2 without type error and other semantic constraints (based on assertions, exceptions, etc.);
- *Economy of development*: Inheritance is used to reduce code or structure duplication.

These categories may overlap or be in conflict with one another. Our discussion includes all of these kinds of hierarchy.

At any stage in the software process, the developers may discover that the class hierarchy is inappropriate and should be changed. We refer to such changes as *hierarchy manipulation* and they are the subject of this report. We describe some possible reasons for manipulating hierarchies, some contexts in which the need to manipulate arises, the relation between hierarchy manipulation and refactoring, and finally some specific problems in hierarchy manipulation.

2.1 Why do we Manipulate Hierarchies?

We don't manipulate hierarchies only for fun but with a given objective. The objective may be to try to improve the way that information is structured by providing better factorization or better decomposition [29, 22], or it may be to conform to some programming language constraints as the transformation from multiple to single inheritance [19, 46]. There may be other objectives.

The reasons for manipulating hierarchies (as specialization of manipulating software) can be placed into five main categories [17]. The first four normally occur after development but the fifth occurs during development.

- **evolution:** supporting changes on requirements
- **reuse:** adapting for reusing purposes
- **maintenance:** making corrections
- **qualification:** looking for good characteristics
- **incremental refactoring:** modification of the hierarchy during development

New requirements or modifications of existing requirements may be functional or non-functional. Non-functional requirements such as "improving efficiency" can lead to hierarchy manipulation [55, page 99] which can be categorized as refactoring (see Section 2.3). Satisfying new functional requirements may often enforce deeper changes, but previous refactoring may better prepare the hierarchy for such changes.

The need to manipulate class hierarchies arises in several contexts:

1. Analysis reveals that the hierarchy is deficient in some respect. For example, classes might be redundant, or classes that should be present are not present.
2. A design review shows that classes are too tightly coupled, not cohesive, or have too few or too many methods.
3. The hierarchy is hard to understand and use, due to a non-rational construction — for example, it might be the result of several different development styles.
4. An expert may find that the constructed hierarchy does not match a natural specialization of the application domain.
5. Refactoring often involves changes to the class hierarchy.
6. When a hierarchy has to be extended or reused, it may be necessary to add generalization classes in order to correctly insert new concepts. In the worst case, the hierarchy may have to be entirely reconstructed to benefit from a systematic construction (a process similar to reverse engineering).
7. A hierarchy developed during design may have to be manipulated to match restrictions in the implementation language. For example, a multiple inheritance hierarchy must be mapped to classes with single inheritance and interfaces for Java implementation [19, 46].

Context	1	2	3	4	5	6	7
evolution				●		●	
reuse					●	●	
maintenance	●			●			
qualification		●	●		●		
incremental					●		●

Table 1. The relationship between categories and contexts of hierarchy manipulation

Table 1 shows the relationship between the categories and contexts that we have identified. Clearly, there is considerable overlap between these contexts; in fact, typical situations will involve a blend of several of them. The need to modify the hierarchy may occur more than once during development.

Hierarchy analysis could be manual or automatic, but we are particularly interested in automatic analysis using, for example, concept lattices (see Section 2.2 below) or metrics (e.g., [15]). Crespo's classification [17] of the "Method" of software (hierarchy) manipulation, asks “how does the manipulation start: by inference or by demand?”. Inference means what we are calling here “automatic analysis” and demand, “manual analysis”. There are other automatic analysis techniques (or inference methods), such as program slicing [54], algorithms based in heuristics [14, 39], and algorithms detecting violation of predefined rules (e.g., the Law of Demeter [35]).

In metrics, coupling and cohesion have been intensively studied, but they do not cover all specific aspects of the quality of class hierarchies. Well-known metrics directly connected to inheritance hierarchy measurement include NMO/NMI (Number of Overridden/Inherited Methods), SIX (Specialization Index) [36], PII (Pure Inheritance Index) [38], and MIF/AIF (Method/Attribute Inheritance Factor) [11]. But these metrics do not address issues such as property redundancy measurement or quality of method specialization, as investigated in [20].

2.2 Formal Concept Analysis

Formal concept analysis (FCA) [4, 3, 28] has several applications in the domain of object-oriented software analysis and development.

- ownership-based [30, 23, 59, 32]: concept analysis is based on the relation that associates a class with a property (attribute/method) it owns (mainly declares or inherits)
- behaviour-based [1, 43, 44]: the relation now links a pair (class,selector) to a composite property like “call mode (via `self` *vs.* via `super`)”, concrete *vs.* abstract implementation, etc.
- usage-based [52]: a variable is associated with a property (attribute/method) if the variable makes access to the property
- orthogonal-variability based [44]: analyzing frameworks for improving design, obtaining orthogonal dimensions on variability (hot spots)
- object-reference based [31]: improving class associations analyzing object references
- combine ownership-based and usage-based [43]
- combine ownership-based and object-referenced based [31]
- other applications [1, 57], not necessarily related to hierarchy manipulation

2.3 Hierarchy Manipulation and Refactoring

The word “refactoring” was first used by Opdyke [39], who defined refactoring as a kind of semantics-preserving program transformation that raises program editing to a higher level and is not dependent on the semantics of a program. An alternative definition by Koni-N’Sapu [34] says “Refactoring consists of changing a software system in such a way that is does not alter the external behaviour of the program. It is a disciplined way to clean up code.”.

Hierarchy manipulation is related but not tied to refactoring. Important evolution and reengineering operations can not be categorized as refactoring because there is no preservation of behaviour. Whereas refactoring preserves semantics, we do not see this as a necessary property of hierarchy manipulation. For example, if the hierarchy is modified to meet new requirements, the semantics of the program will change. Moreover, refactoring may impact aspects of object-oriented systems that do not relate to class hierarchies.

Fowler *et al.* provide a catalog of refactoring transformations [26] but it is not exhaustive. As a first step, however, the catalog could be used to identify hierarchy manipulation operations and try to find out whether they can be inferred and/or automated with FCA. "Inference" here is the key point, because FCA can indicate when and how some transformation must be done. But automation is more than that, because it covers code (or models) analysis and manipulation, parsing techniques, and so on (cf. Section 2.7 below). Bearing in mind that, "if you have a hammer, every problem looks like a nail", we should be careful to avoid missing other analysis techniques.

Crespo [17] proposed a classification for refactoring operations that can be generalized to software manipulation, and can be extended, refined, and with other categories such as optimization techniques [22]. Crespo's classification considers the *reason* for manipulation, as well as the *direction*, *results*, *consequences*, *method*, *human intervention* and *target* of the manipulation, and can be refined and extended either with other categories as defined in [29], or with the classification of other works on hierarchy manipulation such as [22].

Environments that assist refactoring, such as *The Refactoring Browser* [45] should also support hierarchy manipulation.

2.4 Problems in Hierarchy Manipulation

In the following we discuss the problems that we identified in hierarchy manipulation. Each problem is discussed in the framework proposed by the workshop organizers: problem statement, who is affected, forms of solution, and possible approaches.

2.5 Problem: Modelling and Automating Manipulation

Suppose that we wish to improve a hierarchy by analysis based on concept lattices followed by refactoring. This requires solving two problems:

1. How do we formulate a model in terms of concept lattices? The problem is to find the right predicate for the right purpose: a predicate is not intrinsically good or bad, it may or may not be relevant for a given refactoring purpose. What criteria can we use to ensure that the chosen predicate is appropriate?
2. Transforming the current hierarchy to the desired hierarchy by hand is tedious and error-prone. How can we automate the required refactoring?

Who is Affected? Designers working with an iterative process model need criteria and techniques for hierarchy analysis. Implementors performing hierarchy manipulation need software tools to help them.

Forms of Solution. The central problem here is that inference techniques could lead to very complex transformations. We can distinguish atomic and compound refactoring operations, but even compound refactoring operations can be less complex than the required transformation. Perhaps a good combination of that refactoring operations would suffice. The problem, however, is to detect the required combination automatically. We can speak about "refactoring plans" (cf. "population migration plans" in database terminology). It may be possible to formalize refactoring as graph rewriting, because refactoring combination could be very well expressed in terms of graph rewriting. Building refactoring plans to accomplish a given advice (or indication) from inference techniques can be seen as future research direction.

Possible forms of solution include:

- A set of rules or guidelines for assessing the usefulness of the predicate used for concept analysis. Alternatively, Galois lattices (and sub-hierarchies) yield inheritance hierarchies that are proven to satisfy the maximal factorization criterion (among others) for properties among classes.
- An algorithm for refactoring. The algorithm might have two components: the first part would compare the current and desired hierarchies and build a plan of changes; the second part would apply the changes. The solution must also include an implementation of the algorithm, of course.
- Incremental refactoring would manipulate the hierarchy each time it is modified by the designer [23].

Approaches.

- There are many different possible refactoring operations. A first step would be to identify refactoring operations that can be automated by FCA. For example:
    - Attribute/method redundancy can be removed by ownership-based FCA
    - sophisticated ownership analysis can correctly insert abstract methods
    - "Concept pattern 2-case1" [1] of behaviour-based FCA indicates places of possible common code in sibling classes, etc.

  One approach would be to use a catalog such as Fowler's [26] and to analyze for each refactoring operation, which operation can be discovered and/or automated by which kind of FCA — this would probably involve inventing new forms of FCA.
- Think up several predicates and try them out on a variety of hierarchies. If possible, the predicates should be based on well-defined benchmarks and metrics.
- Look for a series of small steps that, taken together, map the current hierarchy to the desired hierarchy. Choose a suitable model or representation of the source code for the implementation of the algorithm (this could be plain text, a linked data structure, or some combination of these).

2.6 Problem: Validating Transformations

Suppose that we have taken a current hierarchy H_c, applied a transformation to it, and obtained the desired hierarchy H_d. How do we validate H_d? This problem has three components:

1. Is the objective of the manipulation fulfilled?
2. Does the structure of the new hierarchy accurately reflect the desired structure of the application?
3. Does the new hierarchy provide the same functionality and performance as the old one?

Who is Affected? If development is understood as a seamless transition from analysis to encoding, initial users (clients/experts of the application domain) should recognize and approve validity of software artifacts that directly encode concepts of the application domain. Natural specialization in the application domain (ontologies) should be more or less reflected in software artifacts.

Without validation, the implementors will have to test the new hierarchy extremely thoroughly to ensure that it behaves in exactly the same way as the old hierarchy and meets all of the system requirements.

If the required transformation can be obtained by means of refactoring, there is no problem because refactoring operations preserve behaviour and we could pass the problem to the refactoring definition and implementation. But, when we start to work with combinations of refactoring operations, we must not only be sure that refactoring combination preserve behaviour but we must also be sure we choose the appropriate combination.

Forms of Solution.

- A tool that evaluates a hierarchy according to stated criteria.
- A tool that formally analyzes and/or runs tests on two hierarchies in order to compare their behaviour and performance.

Approaches.

- There is a subjective aspect to the second component of the problem being described (the structure of the hierarchies): perhaps human judgment would be required to assess the appropriateness of the new hierarchy. However, there are two ways in which the assessment might be partly automated:
  - design metrics and use them to compare the two hierarchies
  - use AI techniques, such as a rule-based expert system, to assess the hierarchies
- It should be possible to establish functional equivalence by formal techniques: for example, by showing that all calls in the new hierarchy have the same effect as equivalent calls in the old hierarchy. However, it is hard to assess performance by formal techniques.

- A more promising approach would be to construct a test suite automatically. Benchmarking, as used in the parallel and high-performance computing community, might be a suitable approach.

2.7 Problem: Separation of Concerns

How can we separate language-dependent and language-independent issues in hierarchy manipulation?

Who is Affected? Without this separation, we would have to build a complete set of tools for each programming or modelling language. Separating out the language-independent issues would enable us to build tools that could do part of the work of hierarchy manipulation for any programming language, or even for multi-language systems.

Forms of Solution. A complete solution would consist of a list of language-dependent issues in hierarchy manipulation, and a list of language-independent issues.

Approaches. Build metamodels for languages. Group languages with similar metamodels into families. Hierarchy manipulations expressed at the metamodel level would apply to all languages in the corresponding family and would, to that extent, be language independent. Manipulations that could be applied to all metamodels would be fully language independent.

In addition to defining the metamodel, we have to define "instantiation of the meta-model": for applying a transformation to a C++ (for example) hierarchy, first we have to interpret C++ artifacts as instances of the meta-model (this can be difficult, and it may be necessary to omit aspects such as access control), secondly; after application of the transformation, we have to re-generate correct C++ code. Huchard *et al.* defined in their research [33]:

- a general meta-model and a ownership-based FCA construction tool using this meta-model;
- a tool for extract from Java classes informations about their interface that match the meta-model
- a tool that uses result of the FCA construction algorithm for generate Java code of an interface hierarchy (that compiles and can be linked to classes).

Crespo *et al.* defined a metamodel for a certain family of languages [17, 18]. A metamodel instantiation for Eiffel has been defined and a Java instantiation is almost complete. The approach is via framework construction. The language-independent part is encoded into the kernel of the framework, and the language-dependent part is encoded as framework hot-spot instantiations. Similar work is being done by the Software Composition Group at the University of Bern [53].

Working at the analysis/design level might help tackling the language dependency problem. UML is an object-oriented meta-model, so a possible solution might be to use UML as much as possible. Other possibilities includes enriching UML and using other analysis design formalisms, e.g., to express specialization between properties—attributes or methods [21].

Some language dependent aspects might even be transformed into this language independent level. Producing a list of OO languages artifacts and their specific implementation in different languages along with the possible transformations of one into another might be of great help. This of course may rely on one or several metamodels.

2.8 Hierarchy Manipulation — Conclusion

The discussion demonstrated that hierarchy manipulation is a rich area in which much research remains to be done. The members of this group feel that a Hierarchy Manipulation Study Group should be established and intend to take steps to form such a group.

3 Mixins

The traditional notion of inheritance binds each subclass very tightly to its superclass(es). The concept of *mixins* can be used to make this connection more flexible.

The concept was first introduced as *mixin classes*, a programming convention in languages such as Flavors [13] and CLOS [8]. A mixin class is an ordinary class that is by convention used in a special manner, namely as one of several superclasses. The idea is that the mixin class adds certain facilities to some of its fellow superclasses, possibly using other facilities of those fellow superclasses. Hence, a mixin class may use features not available in the class itself, because these features are expected to be provided by other classes. It is possible to write a mixin class in Flavors and in CLOS because the LISP family of languages is not statically type checked; but it is also possible to produce run-time type errors (‘message not understood’), if the mixin class uses a feature that should be—but is not—provided by any of its fellow superclasses.

To make the mixin concept more robust it was necessary to develop it as a separate concept, a step taken by Bracha and Cook in 1990 [10]. The mixin as a concept and a language construct has been further developed and refined many times since then, e.g., in [25, 9, 37].

Generally, a mixin is a building block for classes. A mixin M can be applied to a class C, thereby producing a subclass C' of C. With a suitable interpretation of classes and $\oplus$, this could be formalized as $C' = C \oplus M$. Flatt et al. [25] formalize mixins as functions from classes to classes, but there is no deep conflict in these points of view because the function would simply be $\lambda C \,.\, C \oplus M$.

A mixin such as M can be reused with several classes. For example, M may also be applied to D, producing a subclass D'. Using traditional inheritance, we would need two identical copies of the text corresponding to M, in order to create C' from C as well as D' from D. This textual redundancy demonstrates the inferior support for reuse with traditional inheritance, and it introduces a potential for inconsistencies. Moreover, C' and D' will be unrelated with traditional inheritance and name based type equivalence, whereas they would have a common element M when using mixins. It may be possible to write polymorphic code that is capable of working on instances of either C' or D' using features from M; with traditional inheritance it would again be necessary to create two textually identical copies, one working on C' and another working on D'. Since this is concerned with client code, the duplication of code could penetrate deeply into the rest of any system using C' and D'.

To summarize: mixins can be used to open the doors to a number of new abstraction and reuse opportunities. However, the introduction of mixins does not only solve problems, it also raises new problems. We identified three core problems at the workshop which are described below.

3.1 Problem: Mixing Things From Different Sources

When mixins are used it will often be the case that mixin composition ($\oplus$) is used to combine entities written in different contexts. Indeed, it seems to be one of the important benefits of mixins that they could be used to combine a class C from one vendor, V_a, with a mixin M from another vendor, V_b. After all, it may well be that C is better for the given purpose than any class delivered by V_b, but M is better than any mixin delivered by V_a.

However, it is not enough that C has exactly the right semantics for the desired superclass, and M provides exactly the right semantic adjustment for the desired mixin. The two must also agree on a number of more mundane properties associated with the *expression* of the class C and the mixin M. In other words, classes and mixins are not abstract semantic entities, they depend on such seemingly accidental details as the choice of names, access or visibility specifications, `const`, `final`, and other modifiers, and more.

Who is Affected? This problem affects programmers working on complex, real-life projects.

Possible Solutions.

Encapsulation. It may be possible to use encapsulation to make both classes and mixins more abstract. In particular, it may be possible to hide the difference between stored and computed results, at least in some cases. This would, e.g., make it possible for an instance variable of type T in M to (dynamically!) override a method in C returning a value of type T, as is possible in ordinary inheritance in Eiffel. Overriding an instance variable v in C by two methods in M, having signatures similar to a ‘getter’ and a ‘setter’ method for v, might also be feasible in some languages. Since there is no general approach that allows us to use a method (or two) where an object is expected, or vice versa, it might be necessary to depart more radically from main-stream semantics, in order to make stored and computed state freely interchangeable.

In the same vein, it might be useful to let a method in M override two methods in C, or vice versa. This introduces the question of naming, which is discussed below in the last problem.

Disambiguation by origin. If the problem is a name clash in superclasses, i.e., among mixins used to build the superclasses, then it may be possible to solve the problem by explicitly selecting a feature from a particular mixin. This could be similar to the `SomeClass::SomeFeature` syntax in C++. Note that the name clash would have to be resolved at mixin *application*, unless the language allows some knowledge about the actual superclass to be made available at the mixin declaration.

Since a naive semantics for this mechanism would imply that late binding of method implementations is disabled, there is a need to define more sophisticated semantics of such an explicit selection by origin, such as the 'titles' suggested for C++ in [47]. This is all the more important because the superclass from which the feature must be selected is not statically known inside the mixin definition.

Disambiguation by type. With the same the problem, i.e., a name clash in superclasses, it may be possible to use disambiguation by type as a solution. This means that exactly one of the available definitions is chosen, because it matches a given type better than all the others. This probably implies that the usage context (what we called M earlier) must contain a specification of the type of the feature, such that the comparison between this requested type and all the available types (in what we called C) can be based on a visible criterion.

In many languages it would actually be possible to *infer* the type of a named entity from the expression(s) in which it is used, but this seems to be a rather error-prone basis to build on, because the programmer might never realize that there was a name clash, and because seemingly benign changes of the program may change the semantics drastically.

3.2 Problem: How to Specify the Requirements of a Mixin

When composing a class and a mixin it is important that the class satisfies the requirements of the mixin—otherwise they should not be composed. Such requirements may take many forms.

There are the automatically checkable requirements, such as 'any class with which this mixin is composed must define an instance variable named `x` of type `int`', or 'it must define a method `foo` conforming to [a specific signature]'. The reason we might want to make such simple requirements explicit is that we may not know exactly what class C and mixin M are being composed at a given mixin application site. Being explicit about requirements will make it possible to ensure that these simple requirements are satisfied—like an ordinary type system keeping track of the consistency of types of values without actually keeping track of the values themselves. This amounts to giving classes and mixins *types* with respect to mixin application, and checking the types at mixin application. Note that such type checking may require explicit type declarations, and possibly a more verbose mixin composition language.

There are also precisely specifiable requirements based on correctness criteria that cannot be automatically checked, e.g.: 'this mixin method may call the method `lock` once and then `select` or `update` some number of times, and then `unlock` once, and that must be an appropriate usage of these methods from the class with which this mixin is composed'. Whether such a *method protocol* is actually respected by a piece of code is of course undecidable, though it can be checked at run-time. It is even further away from decidability—and it cannot be checked at run-time—whether it is application-correct to treat the superclass methods `lock`, `select`, `update`, and `unlock` as described. Nevertheless, programmers may be allowed to *specify* such requirements explicitly, and it might then be possible to check the consistency of these annotations, e.g., that there exists a method protocol that satisfies all the requirements.

Finally, the requirements of a mixin on its superclass may have to be described in natural language, and it is then up to programmers to check that mixin applications do not violate these requirements. There may be tool-support for *presenting* such requirements to programmers when they write the mixin application expression.

Who is Affected? This problem affects anybody who wants to reuse a given mixin with a given class: A reuser of code needs concise and explicit specifications of constraints on the usage, because (s)he cannot be expected to know how the reused code works in great detail.

Possible solutions.

Specify the requirements. An explicit requirements specification implies more work at mixin definition time, but it also serves as documentation of the exact intentions in this area. If it turns out that the requirements are not satisfied in some case where they 'should' be satisfied, the programmer will have to think about the requirements specification once more. After changing the specification, (s)he should reconsider whether the implementation of the mixin actually fits the new requirements, or—in the case of automatically checkable requirements—(s)he should let the language processing system re-check the requirements.

Infer the requirements. As opposed to the explicitly specified requirements, inferred ones are very easy on programmers at definition time. Programmers can just write the code with some functionality, and both the painstaking derivation of requirements, the tedious typing of them, and the reading-unfriendly verbosity of the resulting code is avoided. Language processing tools may give the programmer the opportunity to inspect the requirements and see if they conform to his wishes, but they do not force the programmer to do so. On the other hand, purely inferred requirements could never include such things as constraints on method protocols and other, more complex issues.

Intermediate solutions. It would be possible to give an explicit requirements specification that is to be treated as an upper bound on the actual demands of any future version of the mixin. Similarly, a class might be annotated with a specification that is to be considered a lower bound on what any future version of the class will provide. This kind of approximate requirements specification will provide some support for safer code evolution. Moreover, it might be possible to combine such incomplete specifications with inferred specifications, giving rise to warnings from compilers and other tools when there is a conflict.

3.3 Problem: Dependence on Names

One particularly thorny issue is the choice of names for features. Each name is chosen by a programmer at some point in the development of a given piece of software, when the future usage contexts are unknown. In particular, code that is intended to be highly reusable might be used in many unforseen contexts, and ironically it is in exactly this kind of code that the right choice of name is most important. Since a mixin generally performs white-box reuse of the class with which it is composed, the mixin depends on a wider set of names and properties in the superclass than client code does. In Java terminology, the mixin would have access to the `protected` interface, rather than being restricted to the `public` interface.

Compared to traditional inheritance, a given mixin is much more vulnerable to name mismatches than an ordinary subclass. The traditional subclass will always be written using exactly the name space that is actually available in its superclass. The mixin may turn out to be very useful with superclasses with different name spaces, except that it can only be applied to superclasses whose features happen to have exactly those names that the mixin expects.

Note, however, that a subclass and a mixin are equally vulnerable to name mismatches arising from *evolution* of the (actual) superclasses. Change a name in a class, and typically both subclasses and applied mixins will break. This illustrates that the dependence on names is a problem with a wide scope.

Who is Affected? This problem also affects anybody who wants to reuse a given mixin with a given class: reuse may be possible or impossible depending on the chosen names for features in the class and in the mixin, rather than on the inherent semantics of the class and the mixin.

Possible solutions.

Explicit renaming. It is possible to use a mechanism such as Eiffel feature renaming to adapt a given mixin M to a given class C: as a subclass C' is being created by applying M to C, each feature of the mixin that needs to have a different name according to the requirements of the class is first renamed. In some cases, features of C could be renamed instead.

Coloring. Coloring is a way of resolving name conflicts. If there are two methods `foo` that conflict, and we need to access them both, then we color one as "the green `foo`" and the other as "the blue `foo`" and now we can talk about them both. Scope rules may be manipulated to direct all usages of names in a given area of source code to prefer the "blue" names, etc. This might also be combined with renaming, so the green `foo` might be renamed and exported as `grass_foo` while the blue `foo` might be renamed as `sky_foo`.

Call-by-declaration. In [24], the concept of 'call-by-declaration' is introduced. It is named according to the traditional phrases used to describe parameter transfer mechanisms for procedures and methods, because the mechanism is similar to such parameter transfers in several ways. However, it is a mechanism that introduces explicit parameterization of a mixin with the declarations upon which it depends. It is then possible to bind these formal declarations to actual declarations in the actual superclass at mixin application time. Call by declaration provides support for feature renaming at mixin application, without affecting the declarations of the class or of the mixin.

Explicit parameterization. It is possible to use a broader notion of explicit parameterization than the one inherent in the call-by-declaration approach. It might for instance be possible to parameterize the mixin with the methods it should provide: If a given method `foo` is declared in the mixin definition but not chosen at parameterization (configuration) time, the method `foo` would simply not be included. As a consequence, requirements on the superclass derived from the implementation of `foo` would vanish. However, the mixin would still have to be consistent, so if some other method `bar` in the mixin calls `foo` then `bar` must also be excluded, or some other implementation of `bar` that does not use `foo` must be provided as a parameter.

3.4 Mixins — Conclusion

The discussions about mixins illustrated that there are several deep problems yet to be solved, and also that the participants in this subworkshop are working actively on the problems, along with other researchers.

References

[1] G. Arevalo and T. Mens. Analysing object-oriented application frameworks using concept analysis. In Black et al. [6], pages 3–9.

[2] Lars Bak, Gilad Bracha, Steffen Grarup, Robert Griesemer, David Griswold, and Urs Hölzle. Mixins in Strongtalk. Technical report, Sun Microsystems, Inc., 2002.

[3] M. Barbut and B. Monjardet. *Ordre et classification.* Hachette, 1970.

[4] G Birkhoff. *Lattice Theory.* AMS Colloquium Publication. American Mathematical Society, third edition, 1967.

[5] Andrew P. Black. A use for inheritance. In Black et al. [6], pages 10–15.

[6] Andrew P. Black, Erik Ernst, Peter Grogono, and Markku Sakkinen, editors. *Proceedings of the Inheritance Workshop at ECOOP 2002.* Number 12 in Publications of Information Technology Research Institute. University of Jyväskylä, Málaga, Spain, June 2002.

[7] Gustavo Bobeff and Jacques Noyé. On the interaction of partial evaluation and inheritance. In Black et al. [6], pages 16–22.

[8] B. Bobrow, D. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. *Common Lisp Object System Specification.* Document 88-002R. X3J13, June 1988.

[9] Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In Rachid Guerraoui, editor, *ECOOP '99 — Object-Oriented Programming 13th European Conference, Lisbon Portugal*, volume 1628 of *Lecture Notes in Computer Science*, pages 43–66. Springer-Verlag, Berlin - Heidelberg, June 1999.

[10] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices*, volume 25, 10, pages 303–311, October 1990.

[11] F. Brito e Abreu and W. Melo. Evaluating the impact of object-oriented design on software quality. In *Proc. METRICS 96.* IEEE Computer Society, 1996.

[12] F. Brito e Abreu, Mario Piattini, Geert Poels, and Houari A. Sahraoui, editors. *Proceedings of the 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2002).* Springer, 2002.

[13] Howard I. Cannon. Flavors: A non-hierarchical approach to object-oriented programming. Symbolics Inc., 1982.

[14] E. Casais. *Managing Evolution in Object Oriented Environments: An Algorithmic Approach.* PhD thesis, Centre Universitaire d'Informatique, University of Geneva, May 1991.

[15] H.S. Chae. Restructuring of classes and inheritance hierarchy in object-oriented systems. Master's thesis, Software Engineering Laboratory, Computer Science Department, Korea Advanced Institute of Science and Technology (KAIST), 1996.

[16] Pierre Crescenzo and Philippe Lahire. Customisation of inheritance. In Black et al. [6], pages 23–29.

[17] Y. Crespo. *Increasing software reuse potential by refactoring.* PhD thesis, Departamento de Informatica, Universidad de Valladolid, 2000. In Spanish.

[18] Y. Crespo, V. Cardenoso, and J.M. Marques. A model language for refactoring definition and analysis. In *Actas PROLE'01, Almagro, Espana*, November 2001. In Spanish.

[19] Yania Crespo, Joseé Manuel Marqués, and Juan José Rodríguez. On the translation of multiple inheritance hierarchies into single inheritance hierarchies. In Black et al. [6], pages 30–37.

[20] M. Dao, M. Huchard, H. Leblanc, T. Libourel, and C. Roume. A new approach to factorization - introducing metrics. In *Proc. Metrics 2002: 8th International Software Metrics Symposium*, 2002.

[21] M. Dao, M. Huchard, T. Libourel, and C. Roume. Spécification de la prise en compte plus détaillée des éléments du modèle objet UML. Technical report, Projet MACAO. Réseau RNTL, 2001. Sous-projet 4, activité 2.

[22] Michel Dao, Marianne Huchard, Thérèse Libourel, and Cyril Roume. Evaluating and optimizing factorization in inheritance hierarchies. In Black et al. [6], pages 38–43.

[23] H. Dicky, C. Dony, M. Huchard, and T. Libourel. On automatic class insertion with overloading. *Special issue of Sigplan Notices - Proceedings of ACM OOPSLA'96.*, 31(10):251–267, 1996.

[24] Erik Ernst. Call by declaration. In Black et al. [6], pages 44–50.

[25] Matthew Flatt, Shriram Krishnamurthi, and Matthias 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 January 1998.

[26] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999. Object Technologies Series.

[27] Peter H. Fröhlich. Inheritance decomposed. In Black et al. [6], pages 51–57.

[28] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations.* Springer, 1999.

[29] R. Godin, M. Huchard, C. Roume, and P. Valtchev. Inheritance and automation: Where are we now? In Black et al. [6], pages 58–64.

[30] R. Godin and H. Mili. Building and maintaining analysis-level class hierarchies using Galois lattices. In *Conference Proceedings Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, pages 394–410, 1993. Published as special issue of SIGPLAN Notices, **28**(10).

[31] C. Hernandez, F. Prieto, M.A. Laguna, and Y. Crespo. Formal concept analysis support for conceptual abstraction in database reengineering. In *Proceedings of the Database Management and Reengineering Workshop at ICSM 2002*, 2002.

[32] M. Huchard, H. Dicky, and H. Leblanc. Galois lattice as a framework to specify algorithms building class hierarchies. *Theoretical Informatics and Applications*, 34:521–548, January 2000.

[33] M. Huchard and H. Leblanc. Computing interfaces in Java. In *Proc. IEE International conference on Automated Software Engineering (ASE'2000)*, pages 317–320, 11-15 September, Grenoble, France, 2000.

[34] G. G. Koni-N'Sapu. A scenario based approach for refactoring duplicated code in object oriented systems. Diploma Thesis of the Faculty of Sciences University of Bern, 2001.

[35] Karl Lieberherr and Ian Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.

[36] M. Lorentz and J. Kidd. *Object-Oriented Software Metrics, a Practical Guide.* Prentice Hall, 1994.

[37] Carine Lucas and Patrick Steyaert. Modular Inheritance of Objects Through Mixin-Methods. In Peter Schulthess, editor, *Advances in Modular Languages*, pages 273–282. Universitatsverlag Ulm GmbH, 1994. Proceedings of the Joint Modular Languages Conference, University of Ulm, Germany, 28-30 September 1994.

[38] B. K. Miller, P. Hsia, and C. Kung. Object-oriented architecture measures. In *32nd Annual Hawaii International Conference on Systems Sciences.* IEEE Computer Society, 1999.

[39] W.F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992. also Technical Report UIUCDCS-R-92-1759.

[40] Klaus Ostermann and Mira Mezini. Blurring the borders between object composition, inheritance, and delegation. In Black et al. [6], pages 65–68.

[41] Jens Palsberg and Michael I. Schwartzbach, editors. *Types, Inheritance and Assignments A collection of Position papers of the ECOOP'91 workshop W5*, DAIMI

PB-357, Geneva, Switzerland, July 1991. Computer Science Department, Aarhus University.

[42] Claudia Pons. Generalization relation in UML model elements. In Black et al. [6], pages 69–75.

[43] F. Prieto, Y. Crespo, J.M. Marques, and M.L. Laguna. Mecanos and formal concept analysis as support for framework construction. In *Actas de las V Jornadas de Ingenieria de Software y Bases de Datos (JISBD'2000)*, pages 163–175, November 2000. In Spanish.

[44] F. Prieto, Y. Crespo, J.M. Marques, and M.L. Laguna. Formal concept analysis as support for domain framework evolution. In *Taller de Evolucion de Software. VI Jornadas de Ingenieria de Software y Bases de Datos (JISBD'2001)*, November 2001. In Spanish.

[45] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[46] Cyril Roume. Going from multiple to single inheritance with metrics. In *[12]*, pages 30–37, 2002.

[47] Markku Sakkinen. A critique of the inheritance principles of C++. *Computing Systems*, 5(1):69 – 110, Winter 1992.

[48] Markku Sakkinen, editor. *Multiple Inheritance and Multiple Subtyping, Position papers of the ECOOP 1992 Workshop W1*, Working Paper WP-23, Utrecht, the Netherlands, June 1992. Department of Computer Science and Information Systems, University of Jyväskylä.

[49] Markku Sakkinen. Exheritance - class generalisation revived. In Black et al. [6], pages 76 – 81.

[50] Nathanael Schaerli, Stéphane Ducasse, and Oscar Nierstrasz. Classes = traits + states + glue. In Black et al. [6], pages 82–88.

[51] K. Chandra Sekharaiah and D. Janaki Ram. Object schizophrenia problem in modeling is-role-of inheritance. In Black et al. [6], pages 88–94.

[52] G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. *Proceedings of the ACM SIGSOFT Sixth International Symposium on the Foundations of Software Engineering, ACM SIGSOFT Software Engineering Notes*, 23(6):99–110, November 1998.

[53] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings ISPSE 2000*, pages 154–164. IEEE, 2000.

[54] F. Tip, J-D. Choi, J. Field, and G. Ramalingam. Slicing class hierarchies in C++. In *Proceedings of the Conference ACM SIGPLAN OOPSLA'96*, pages 179–197. Special issue of ACM SIGPLAN Notices 31(10), ACM Press, October 1996.

[55] F. Tip and P.F. Sweeney. Class hierarchy specialization. In *Proceedings of the Conference ACM SIGPLAN OOPSLA'97*, pages 271–285, 1997.

[56] Mads Torgersen. Inheritance is specialisation. In Black et al. [6], pages 95–101.

[57] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *21st International Conference on Software Engineering, ICSE-99*, pages 246–255. ACM, 1999.

[58] Peter Wegner. Dimensions of object-based language design. In *Proceedings Second ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 168–182, Orlando, Florida, 1987. ACM Press.

[59] A. Yahia, L. Lakhal, and J.P. Bordat. Designing class hierarchies of object database schemas. In *Actes 13èmes journées Bases de Données Avancées*, pages 371–390, Grenoble, France, September 1997.