Objects and Classification
A Natural Convergence

Marianne Huchard[1], Robert Godin[2], and Amedeo Napoli[3]

[1] LIRMM, 161 rue Ada, 34392 Montpellier cedex 5, France
marianne.huchard@lirmm.fr
[2] Université du Québec à Montréal, Département d'informatique C.P.8888, Succ.CV, Montréal (Québec), Canada H3C 3P8
godin.robert@uqam.ca
[3] LORIA – UMR 7503 (CNRS – INRIA – Universités de Nancy), B.P. 101, 615 rue du jardin botanique, 54602 Villers-lès-Nancy Cedex, France
amedeo.napoli@loria.fr

Abstract. Classification is a central concept in object-oriented approaches such as object-oriented programming, object-oriented knowledge representation systems (including description logics), object-oriented databases, software engineering and information retrieval.
Nevertheless, research works on classification have often been carried out separately within these different approaches, and they have not always been precisely confronted and connected. The goal of the workshop was to confront these complementary viewpoints on classification, to exhibit and discuss commonalities and differences within these approaches.

1 Introduction

Object-oriented approaches are mainly based on a proximity between real world entities and their computer representation. This leads primarily to the well known "classes"[1], "instances", and specialization relationships implemented by inheritance. More generally, a common trend in object-orientation is to reify all the considered items, and classify them. Unfortunately, research works that deal with classification in object-oriented approaches are spread over different fields such as object-oriented programming, object-oriented knowledge representation systems (including description logics [10, 21]), object-oriented databases, software engineering and information retrieval [15, 6]. Furthermore, they are concerned by various entities like use cases, classes, prototypes, methods, patterns, and they have different goals, like designing, reasoning, indexing and programming. Due to these differencies, many research works are carried out separately, although they have a common concern.

The workshop intended to put together and to confront all these complementary viewpoints on classification, in order to bring on the foreground common questions, and to provide a basis to discuss and share classification techniques.

[1] When ambiguous, denoted by *oo-classes* for "classes in object-oriented systems".

J. Malenfant, S. Moisan, A. Moreira (Eds.): ECOOP 2000 Workshops, LNCS 1964, pp. 123–137, 2000.

This document reports and synthetizes the contributions (position papers and discussions) with regard to the following points:

- main trends of the workshop,
- definitions of classification (as "classification" is polymorphic),
- objectives followed in the classification process,
- classified entities,
- context of the classification (relatively to the life-cycle of the software),
- structure of the classifications produced (partitions, trees, lattices, etc.),
- systems, tools, techniques in general.

2 An Overview of the Contributions

The workshop was appropriately introduced by [w10], which recalled the Aristotle's ideas on classification and how they concern object-oriented approaches: either because they are already present although hidden, or because they could be beneficial.

Apart this first contribution, two main trends were represented.

The first, or "software classification", is concerned with classification problems applied to software engineering. It not only aims at managing classes and hierarchies, but it also extends the notion of classification to all software artifacts: oo-class hierarchy evolution [w9], reuse of persistent instances [w3], tagging software [w5], logic predicates for reasoning on the software [w7] or generating code [w12], use case management [w14].

The second trend is closer to knowledge representation concerns. It focuses more on the notion of oo-class hierarchy, develops specific models (classes described by disjunctions [w8], prototypes [w1], a UML-like model [w4], graphs [w2, w6]), and techniques as instance classification [w8, w4], conceptual clustering [w2, w13, w6], or genetic algorithms [w11].

3 The Meaning of "Classification"

The term "classification" usually covers both the process and the result of:

- class discovery
  - either by dividing a set of entities; classes obtained in this way can be called "extensional classes"; pure extensional classes (sets) are rather uncommon in object-oriented approach;
  - or by associating a description to a set of entities; these classes are "intensional classes"; in this case, the intension can be given either *a priori*, for example by a human actor from his knowledge of the domain, or *a posteriori*, when it is deduced from the analysis of a set of objects. Such intensional classes also have an extensional counterpart: the objects created from or covered by the intension.
- class organization ("class classification") into specialization structures (trees, lattices, etc.),
- associating a class to an entity ("entity/instance classification").

The difference between extensional and intensional classes is pointed out and discussed in [w7]. Even in data analysis [1, 17], which is more interested in the extensions, the classes are described by "numerical structures" (line, hyperplane, etc.) that can be considered as their intension, although interpretating this intension is not obvious.

Intensional and extensional classifications are intimately related. Gather entities in a set to produce an extensional class implies tagging these entities by their membership to that class. The different tags resulting from the different memberships of an entity can be used as a description of this entity. Intensional classes can then be built according to these descriptions, having an extension which may be different from the initial extensional classes, etc.

Let us detail how these different meanings of classification appear in the contributions.

Oo-Classes. Oo-classes are essentially intensional classes. In object-oriented modeling and programming, they are traditionally defined *a priori*, with their extension mostly derived at running stage. This is usually done "manually", but techniques for *a posteriori* class discovery and organization also exist and were represented at the workshop. In the context of programming languages, [w9] deals with local class hierarchy modification by adding "interclasses". [w13] compares two common clustering techniques, similarity-based clustering techniques and the Galois lattice approach. The debate on techniques for a *posteriori* class discovery was broadened out by two contributions [w2, w6] relative to the generalization of classes described by a graph-based language.

Instances. When instances are not created from an oo-class, as for example in object-oriented representation systems, instance classification is a main problem. Interesting variants of instance classification methods have been highlighted during the workshop, in the context of classes described by disjunctions [w8], or in a UML-like model where classes and associations coexist [w4]. The approach of [w11] exploits instance classification to guide emergence of new classes, in a context where instances may evolve independently of classes. Such a method may admit classes defined *a priori* and promotes classes discovered *a posteriori*. Another original problem relative to instance classification is using a set of (persistent) instances with different (although related) class hierarchies [w3]. In this work, accurate descriptions of the relationships between classes are used to guide the loading of persistent instances by an application (this loading has to be understood as instance classification).

Prototypes. It was shown in [w1] that, in the framework of prototype-based languages, "class-less" in essence, there is also a need to define generalizations (abstractions). The ambiguity that exists, in some of these languages, among concrete individual, prototypical individual and kind of classes (trait), results in more flexible language constructions that could help classification processes.

The meaning of classification in prototype-based languages is roughly similar to the corresponding problem in class-based languages, only minor variations were reported and the topic is still to be explored.

Software Artifacts. In "software classification" [w5, w7, w12], two main classifications are mentioned: "manual classification", which consists in putting together software artifacts manually, and creates extensional classes [w7]; "virtual classification" where classes are intensional. The intension can be represented either by a logic predicate [w7, w12], or by tags [w5]. The classes are mainly defined *a priori*, with their extension often computed from the intension.

The possibility of improving software classifications by the organization of software artifact classes has been discussed. Criteria like inclusion of the extensions, or specialization between intensions (implication between the logic predicates associated to the artifact classes, for example) could be used. The techniques mentioned in [w13], that are not restricted to the oo-classes production and organization, could help such a classification process and produce more elaborated artifact classifications.

The table 1 gives a (subjective) overview of the contributions regarding the classification meaning (x+ in the first row corresponds to works where extensional classes are also mentioned or used).

	[w1]	[w2]	[w3]	[w4]	[w5]	[w6]	[w7]	[w8]	[w9]	[w11]	[w12]	[w13]	[w14]
intensional	x+	x	x	x	x	x	x+	x	x	x	x	x	x
a priori			x	x	x		x	x	x	?	x		x
a posteriori	x	x				x				x		x	
class classif.	x	x				x			x	x		x	
instance classif.	x		x	x				x		x			?

Table 1. Use of classification meanings

4 Objectives of Classification

As reported by [w10], classification (one form of "division" for Aristotle) is part of the process of understanding and reasoning. This section shows how, replaced in the framework of object-oriented approaches, these general assertions become more concrete, and how some are more specifically related with software engineering, while others are characteristic of object-oriented knowledge representation.

Design. Design in object-oriented approaches is concerned with producing a model of the domain: oo-classes represent "concepts of the domain" (groups of

objects that share a common behavior and structure); specialization hierarchies more or less reflect complex concept classifications; other entities (states, dynamic diagrams, design patterns, etc.) are concerned with dynamic aspects or high-level modeling. In software engineeering, all design methods give accurate guidelines to build specialization hierarchies, as for example OMT [25] or the UML formalism [5].

Many contributions are concerned with design. For [w2, w6, w13] the problem is clearly to organize knowledge by clustering or generalization techniques. A proposal of [w1] is to use prototypes at the design stage, to facilitate the process of abstraction discovery. When the domain description appears to be stabilized, an oo-classes hierarchy can be generated. Software classifications, that help the detection of high-level informations such as object collaborations, components [w5] or even design patterns, also play a role in improving the domain model.

Providing a model of the domain is a concern that goes beyond the design step, especially because the domain, or our understanding may evolve. [w11] approaches this problem, describing an oo-class evolution based on instance evolution, that is, at run-time. The proposal of [w9] is in a same evolution perspective, but limited to the code (note that this code is supposed in use), and it proposes an "interclassing" technique to adjust the class hierarchy to some local evolutions (for example, methods or classes become deprecated —out of date— or the behavior of a class is slightly changed).

Storage, Inspection, and Recovery. A main point in [w5, w7, w14] is to index, browse and query large collections of software artifacts, like oo-classes, methods, use cases, etc.

Another issue is related to persistence and reusability [w3]: persistent objects have to be reusable! Classification may be used to improve these two related aspects of object-oriented programming (organizing and indexing persistent objects).

Programming. In the framework of software engineering, the encoding in a programming language of the oo-class hierarchy gives a basis for programming. Some parts may be generated automatically, for example, most software development tools generate part of the class code directly from design diagrams, although method bodies cannot be generated in the same way. Another approach is, like in [w12], to define a classification of some artifacts that is used afterwards to generate code (for example add a specific code to all classes verifying a predicate). The proposal in [w9] has also to be considered as a help for the programming step. The problem of interclassing is to build a superclass instead of a subclass, and a consequent question is: how interclassing can complement subclassing for code organization? Last but not least, classification appears to be useful in order to improve code factorization [13, 9].

Software Architecture Control. [w7] shows the interest of virtual software classifications in checking the compliance between a software architecture and its implementation, and in keeping detected abstractions on the software synchronized with the code.

Software Running. After encoding in a programming language, some dynamic aspects (as method calls) are supported by the classification (in this context the inheritance hierarchy). In [w3], the problem is really to use a classification mechanism to run an application on persistent instances that may have been generated by another application with different classes.

Reasoning, Predicting. In the software engineering context, the virtual classifications of [w7], as they are expressed in a logic language, are used for reasoning on the software architecture. Apart this aspect, reasoning is more the purpose of object-oriented knowledge representation systems.

With regard to reasoning, problems are to understand how classification can be a basic tool for inferencing, and to enlighten the different relations existing between the different formalisms for representing knowledge and solving problems using classification. object-oriented knowledge representation and description logics are examples of such representation and reasoning formalisms. Instance classification [w4, w8] allows for example to deduce instance features (most of the time the value of an attribute).

5 Entities under Classification

Many entities are candidate for a classification process. They can be divided among:

- a structural viewpoint ("object" in a broad meaning, that is, instances, classes, prototypes; features as attributes and operations, associations, object states),
- a dynamic viewpoint (events, changes of state, actions, interactions).

And as we have seen before, classification may be in concern with:

- "objects" (always in a broad meaning), described by their structural and dynamic features,
- the other entities (operations, associations, state diagrams, collaboration diagrams, use cases, etc.).

5.1 Entities

"Objects". Many contributions deal with "object" classification. Two main models, stemming from cognitive science were discussed: the class-based model and the prototype-based model. For the class-based model, a concept is described

by an "extension" (set of concrete examples) and an "intension" (set of features). Objects may have two different status, instance or class status (an object that has the class status may also be instance of another class, usually called its metaclass). For the prototype-based model, a concept is described by reference to a special object, namely the prototype. The class-based model may be considered as more complex because each object may have two roles (instance or concrete object/class or abstract object), and there are two different abstraction mechanisms (instanciation and specialization). By contrast, in the prototype-based model, there is only one status for the objects, and they are, in a first approximation, related by only one relation, "is-extension-of", that supports the delegation mechanism. Nevertheless this last model suffers from several problems (identity and organization) that are differently solved in languages. An important discussion based on the presentation of [w1] was that it would be worthwhile to look at "hybrid" languages that unify and integrate the benefits of prototype-based and class-based languages in a seamless way. The Agora language [28, 29] is an example of such a hybrid system.

Software Artifacts. Software classification [w5, w7, w12, w14] is potentially concerned by all entities (classes, methods, etc.), and gives views on the software not limited to a specialization (or inheritance in the programming case) hierarchy on oo-classes. Actually, this trend, that has some links with aspect-oriented programming, generalizes the impact of classification in object-oriented approaches. In particular, conceptual clustering techniques applied to code abstractions could lead to the automatic discovery of "aspects".

Other useful references on software classification are [24, 23, 18].

5.2 Description Language

We inspect now in more detail the "language" used to describe entities under classification, and the language used to describe classes if it is different. It is useful to warn the reader that this notion of "description language" is not always well codifyed.

Objects/Prototypes/Oo-Classes. They are mainly described by a conjunction of features, among which there are always attributes, sometimes with a type —a range— (mainly in oo-classes) or a value (mainly in concrete objects [w13]). The presence of methods among features is more usual in the programming approaches. Methods are in the center of the problem for [w9], being the cause or the solution of the modification; they are present in the model of [w3] and may be used for the loading of an instance (if we consider that a method can be called in the invariant of a class); they are mentioned, but not discussed in [w11]. They are also considered in class-construction algorithms [13, 9] which were mentioned during the discussions.

In a more prospective approach, [w8] has shown that it can be sometimes convenient to introduce disjunctions in the class description to describe the variability of objects within the same classes (individuals and natural kinds). This

is actually the case in the so-called semi-structured classes and semi-structured objects. Moreover, links can be made with polythetic classes where a class is defined with respect to a set of attributes that are shared by the objects [27, 30]: the attributes are not necessarily all possessed by the members of the class, and a class is no more defined by a defined set of attributes acting as a set of necessary and sufficient conditions for an object to be a member of the class.

Another trend, close to instance and class diagrams in the UML formalism, is to represent objects provided with their features *and* associations. First, this leads to a model like in [w4], second this tends to consider objects described by graphs. Methods using such a representation have been proposed in [w2, w6], and [w6] has tried to fill the gap between theoretical aspects and pratice, searching for a graph model appropriate in the object-oriented context.

An interesting point was to make a distinction about the nature (significance) of object features. [w10] reports the Aristotle's division among properties and accidents. In [w11], there is a notion of "fundamental genotype", that covers the set of "minimal and fundamental features inherent to a population", by contrast to features specific to classes in a population.

Software Artifacts. A first proposal for software artifacts is to attach to them a collection of tags [w5] in order to index and query the artifacts. These tags are included by the developpers in the source code. A tag has a name and a value, that can be a string or a number.

In another approach [w7, w12], software artifacts are described by their code, and classification is done by logic predicates on the code. These logic predicates are abstractions on the software and can be interpreted as a kind of declarative approach to code manipulation.

For use cases, the description is a tricky problem [w14], knowing that they are given in a somewhat unstructured manner based on natural language, that is ambiguous by nature. The use case management can be enhanced when an ontology of the domain related to the use cases is available, i.e. a kind of knowledge-based approach to use case management.

Returning to UML diagrams, as they are in essence labelled graphs, a question is to exploit results of [w2, w6] to have a way of generalizing them, producing patterns of event sequence, collaboration, state change, etc.

6 Classification Context

The previous sections made clear that classification is related to all the main steps of the software life-cycle:

- Analysis, for example with use cases [w14].
- Design, design of oo-class hierarchies [w1], or design of artifacts in general [w5, w7].
- Implementation, when classification serves as a guide for encoding [w9, w12].

– Exploitation (run-time), where the computation [w3] or the reasoning [w8, w4] are based on instances.

It appears also that a main common topic of interest is evolution. This is for example the case with "automatic class hierarchy reorganization" [w1], "exploiting classification for software evolution" [w5], using conformance checking for managing co-evolution between an architecture and the software code to deal with architectural drift and software erosion [w7], hierarchy evolution [w9], "Object Structures Emergence" [w11], etc. A difference between the approaches is that the artifact-based software classification approach deals more with design-time evolution, while the object-oriented classification problematics is present in all steps.

7 Forms of Classifications

7.1 Classification Structures

The simplest form of classification structure is an unrelated collection of sets. This is the case in artifact-based software classification [w5, w7]. As said before, an issue was to see if it could be useful to compare and organize these sets simply by inclusion, or to apply conceptual clustering techniques to these sets.

However, most of the contributions deal with hierarchies, whose structure is a tree [w4], a lattice [w2, w6, w13], or any partial order [w1, w3, w8]. This variety is reflected in languages, some of them admitting multiple inheritance (C++, Eiffel), others only single inheritance (Smalltalk). Java has a special policy concerning this point: it admits two kinds of concepts, classes and interfaces, with single inheritance for classes and multiple inheritance for interfaces.

The viewpoint of Aristotle's is the following [w10]. The division must be exhaustive, with parts mutually exclusive, and an undirect consequence of Aristotle's principles is that only leaves of the hierarchy should have instances. Furthermore, the divisions must be based on a common concern (the *discriminator* in UML). This is not always the case in usual programming practice. Multiple inheritance, for example, is contradictory with the assumption of mutually exclusive parts, and instances may in general be directly created from all (non-abstract) classes. Direct subclasses of a class can be derived according to different needs (for example [25], a class Employee can be specialized at the next level by classes that distinguish the status —first discriminator— mixed with classes that divide employees into pensionable or not-pensionable —second discriminator). How are these principles relevant to class hierarchies ? This still is an open and somewhat controversial question.

7.2 Links between Classes and Instances

Traditional object-oriented programming languages assume that instances are created with respect to a class, and remain linked to this class all their life-time.

The rule also is that an instance belongs to a single class (mono-instanciation), even if less common models with multi-instanciation exist [7, 4].

By contrast, in object-oriented representation systems and databases, having instances more independent of classes is an important need [w4, w8, 4]. This characteristic extends to prospective models like [w3, w11, 22] or [8] that allows objects to change their classes depending on property-based classification that augments the explicit type-based classification of ordinary classes.

8 Techniques and Strategies

Inductive vs. Deductive Classification. Classification can be used in ascendant (inductive) and descendant (deductive) ways. In the work of [w9], for example, interclassing is a form of inductive classification. When a class is derived to obtain a new class (subclass), as in the usual programming practice, the classification is rather descendant.

Inductive discovery of a class hierarchy is usually called conceptual clustering [20, w2, w6, w13] in the machine learning field while the term classification (or hierarchical classification) is used to refer to the deductive process of finding the best place for a new instance within a fixed class hierarchy [3, 19, w8, w4]. Concept formation is used to characterize the fact that the class hierarchy is discovered incrementally [12]. Conceptual clustering and concept formation fall under the category of unsupervised learning because the classes are not known in advance.

Conceptual Clustering in Question. In [w13], a comparison has been carried on the relations existing between similarity-based classification and Galois lattice-based classification (also known as formal concept analysis [31]): similarity-based classification is flexible and easy, while Galois lattice-based classification is a more complete classification method. There are a number of situations where similarity-based classification can complement Galois lattice-based classification and conversely. This contribution opens a promising research way, as such clustering approaches are more and more used for software class hierarchies construction [13, 9, 2, 14, 26, 16].

The complexity of the classification operation was hightlighted in [w2, w6]: classification is a NP-hard problem in the general case and this must be taken into account for realistic and large problems involving classification [11]. A kind of anytime classification may be studied for real time problems, as shown in [w2]. In the case of graph-based description languages, [w6] proposes to use rather models for which the complexity of generalization is polynomial, as for example rooted labeled locally injective graphs.

9 Conclusion

Many topics have been addressed and discussed during the workshop. Classification is very ubiquitous and appears to be a central tool both for object-oriented

representation and object-oriented programming. The workshop has provided a number of criteria to compare and appreciate classification techniques or results, the main references on that issue are [w7, w13].

Many directions have still to be studied and developed. Classifiers as used in object-oriented representation systems could be used with profit for class design in object-oriented programming, i.e. building and reorganizing class hierarchies. As well, aspects of object-oriented programming such as the design and the management of code artifacts, persistence, could be considered also with profit in the field of object-oriented representation systems.

A main issue is ensuring the consistency between the different classification operations made at different steps in the life-cycle of a software (discovery and organization of classes in design, classes in programming, dynamic aspects, patterns, etc.). In object-oriented programming, for example, an instance is created from a class, but nothing ensures that it is the best fitting class for that instance. Classes can be organized into single inheritance hierarchies at the design step, and into multiple inheritance hierarchies in the code (or the opposite): which tools may guarantee a systematic transition between the two models?

Software classification opens a very promising way that could be extended and applied to modeling artefacts, such as UML diagrams, maybe with other forms of description languages.

We think that "classification", in all its meanings, is something that structures very well the view one can have on the object-oriented field.

Acknowledgments. The authors would like to thank Tom Mens who contributed by fruitful notes to this document.

10 List of Participants

Daniel Bardou
Action ROMANS - INRIA Rhône Alpes 615 avenue de l'Europe, F-38330 Montbonnot Saint-Martin, France
email: Daniel.Bardou@inrialpes.fr
http://www.inrialpes.fr/romans/people/bardou

Isabelle Bournaud
LRI, Bat. 490 Université Paris XI - Orsay 91 405 Orsay Cedex, France
email: Isabelle.Bournaud@lri.fr

Cécile Capponi
Laboratoire d'Informatique de Marseille , Université de Provence UFR M.I.M., Technopôle de Château-Gombert – 39 rue Joliot Curie 13453 Marseille Cedex 13, France
email: Cecile.Capponi@lim.univ-mrs.fr

Pierre Crescenzo
Laboratoire Informatique Signaux et Systèmes de Sophia Antipolis (I3S), UPRES-A 6070 du C.N.R.S., Les Algorithmes/Bâtiment Euclide, 2000 route

des Lucioles, BP 121, 06903 Sophia-Antipolis Cedex - France
email: Pierre.Crescenzo@unice.fr

Koen De Hondt
MediaGeniX N.V., Gossetlaan 54, B-1702 Groot-Bijgaarden, Belgium
email: Koen.DeHondt@MediaGeniX.com
http://www.mediagenix.com

Dirk Derrider
Programming Technology Lab Departement Informatica, Faculteit Wetenschappen, Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussel, Belgium
email: dirk.deridder@vub.ac.be
http://progwww.vub.ac.be/

Roland Ducournau
LIRMM, 161 rue Ada, 34392 Montpellier cedex 5, France
email: ducour@lirmm.fr
http://www.lirmm.fr/~ducour

Reginald Froli
CUseeMe Networks
email: rfroli@wpine.com

Jérôme Gensel (Action ROMANS - INRIA Rhône Alpes)
email: Jerome.Gensel@inrialpes.fr
http://www.inrialpes.fr/romans

Robert Godin
Université du Québec à Montréal, Département d'informatique C.P.8888, Succ.CV, Montréal (Québec), Canada H3C 3P8
email: Godin.Robert@uqam.ca
http://saturne.info.uqam.ca/~godin/

Giovanna Guerrini
DISI, Università di Genova via Dodecaneso, 35 16146 Genova, ITALY
email: guerrini@disi.unige.it
http://www.disi.unige.it/person/GuerriniG

Marianne Huchard (LIRMM)
email: huchard@lirmm.fr
http://www.lirmm.fr/~huchard

Philippe Lahire (I3S)
email: lahire@unice.fr

Kim Mens (Programming Technology Lab)
email: kimmens@vub.ac.be
http://progwww.vub.ac.be/~kimmens

Tom Mens (Programming Technology Lab)
email: tommens@vub.ac.be
http://progwww.vub.ac.be/~tommens

Amedeo Napoli
LORIA – UMR 7503 (CNRS – INRIA – Universités de Nancy), B.P. 101, 615 rue du jardin botanique, 54602 Villers-lès-Nancy Cedex, France
email:amedeo.napoli@loria.fr

Pascal Rapicault
I3S - ESSI, 930, route des Colles, 06902 Sophia Antipolis, France
email: rapicaul@essi.fr
http://www.essi.fr/~rainbow

Derek Rayside
Electrical & Computer Engineering University of Waterloo, Waterloo, Canada
email: drayside@swen.uwaterloo.ca

Dalila Tamzalit
Equipe Modèles par Objets, Université de Nantes - Faculté des Sciences IRIN, 2, rue de la Houssinière BP 92208 44322 Nantes cedex 03, France
email: Dalila.Tamzalit@irin.univ-nantes.fr
http://www.sciences.univ-nantes.fr/irin/Theme_Objets

Tom Tourwe (Programming Technology Lab)
email: Tom.Tourwe@vub.ac.be
http://progwww.vub.ac.be/

Petko Valtchev (UQAM)
email: valtchev@info.uqam.ca

Bart Wouters (Programming Technology Lab)
email: bart.wouters@vub.ac.be
http://progwww.vub.ac.be/

Roel Wuyts (Programming Technology Lab)
email: rwuyts@vub.ac.be
http://prog.vub.ac.be/~rwuyts

11 Position Papers

All position papers can be seen on the web site of the workshop:
http://www.lirmm.fr/~huchard/WorkshopClassif.html

They are also available in:
Contributions of the ECOOP'00 Workshop, "Objects and Classification, A natural convergence", Research Report LIRMM n.00095, Marianne Huchard, Robert Godin, Amedeo Napoli, (Eds)

[w1] D. Bardou. *"Inheritance Hierarchy Automatic (Re)organization and Prototype-based languages"*
[w2] I. Bournaud. *"Automatic objects organization"*

[w3] A. Capouillez, R. Chignoli, P. Crescenzo, P. Lahire. *"Towards a More Suitable Class Hierarchy for Persistent Object Management"*
[w4] C. Capponi, J. Gensel. *"Classifications among classes and associations: the AROM's approach"*
[w5] K. De Hondt, P. Steyaert. *"Exploiting classification for software evolution"*
[w6] M. Liquière. *"A machine learning model for generalization problem in object-oriented approaches"*
[w7] K. Mens, T. Mens. *"Codifying High-Level software Abstractions as virtual classifications"*
[w8] A. Napoli. *"Classification and Disjunction in object-based representation systems"*
[w9] P. Rapicault. *"A pragmatic approach to hierarchy evolution"*
[w10] D. Rayside, G.T. Campbell. *"An Aristotelian Introduction to Classification"*
[w11] D. Tamzalit, M. Oussalah. *"A model for Object Structures Emergence"*
[w12] T. Tourwe, K. De Volder. *"Using software classifications to drive code generation"*
[w13] P. Valtchev, R. Missaoui. *"Similarity-based Clustering versus Galois lattice building: Strengths and Weaknesses"*
[w14] B. Wouters, D. Deridder, E. Van Paesschen. *"The use of Ontologies as a backbone for use case management"*

References

[1] P. Arabie, L.J. Hubert, and G. De Soete, editors. *Clustering and Classification*. World Scientific Publishers, River Edge, NJ (USA), 1996.
[2] H. Astudillo. Maximizing object reuse with a biological metaphor. *Theory and Practice of Object Systems (TAPOS)*, 1997.
[3] F. Baader, B. Hollunder, B. Nebel, H.-J. Profitlich, and E. Franconi. An Empirical Analysis of Optimization Techniques for Terminological Representation Systems. *Journal of Applied Intelligence*, 4(2):109–132, 1994.
[4] E. Bertino and G. Guerrini. Objects with multiple most specific classes. In *Proceedings of ECOOP'95*, pages 102–126, 1995.
[5] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
[6] C. Carpineto and G. Romano. A Lattice Conceptual Clustering System and Its Application to Browsing Retrieval. *Machine Learning*, 24(2):95–122, 1996.
[7] B. Carré and J. Geib. The Point of View notion for Multiple Inheritance. *Special issue of Sigplan Notice - Proceedings of ACM ECOOP/OOPSLA'90*, 25(10):312–321, 1990.
[8] Craig Chambers. Predicate classes. In *Proceedings of ECOOP '93*, Lecture Notes in Computer Science 707, pages 268–296. Springer-Verlag, Berlin, 1993.
[9] H. Dicky, C. Dony, M. Huchard, and T. Libourel. On automatic class insertion with overloading. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'96*, 31(10):251–267, 1996.
[10] F.-M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. Reasoning in description logics. In G. Brewka, editor, *Principles of Knowledge Representation*, pages 191–236. CSLI Publications, Stanford (CA), USA, 1996.
[11] F.M. Donini, M. Lenzerini, D. Nardi, and W. Nutt. The complexity of concept languages. *Information and Computation*, 134(1):1–58, 1997.
[12] D. H. Fisher, M. J. Pazzani, and P. Langley. *Concept Formation: Knowledge and Experience in Unsupervised Learning*. Morgan Kaufmann, San Mateo, CA, 1991.

[13] R. Godin and H. Mili. Building and Maintaining Analysis-Level Class Hierarchies Using Galois Lattices. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'93*, 28(10):394–410, 1993.
[14] R. Godin, H. Mili, G. W. Mineau, R. Missaoui, A. Arfi, and T.-T. Chau. Design of Class Hierarchies Based on Concept (Galois) Lattices. *Theory And Practice of Object Systems*, 4(2), 1998.
[15] R. Godin, R. Missaoui, and A. April. Experimental Comparison of Navigation in a Galois Lattice with Conventional Information Retrieval Methods. *International Journal of Man-Machine Studies*, pages 747–767.
[16] M. Huchard and H. Leblanc. Computing Interfaces in Java (short paper). In *proceedings of Automated Software Engineering (ASE'2000)*, 11-15 September, Grenoble, France.
[17] N. Jardine and R. Sibson. *Mathematical Taxonomy.* John Wiley & Sons Ltd, London, 1971.
[18] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, and K. Nygaard. Classification of Actions or Inheritance also for Methods. In *Proceedings of* ECOOP *'87, Paris, Special issue of Bigre 54, Lecture Notes in Computer Science 276*, pages 109–118, 1987.
[19] R.M. MacGregor and D. Brill. Recognition Algorithms for the Loom Classifier. In *Proceedings of AAAI'92, San Jose, California*, pages 774–779, 1992.
[20] R. Michalski and R. E. Stepp. Learning from Observation: Conceptual Clustering. In R. Michalski, J. Carbonell, and T. Mitchell, editors, *Machine Learning : an Artificial Intelligence Approach.* 1978.
[21] B. Nebel. *Reasoning and Revision in Hybrid Representation Systems.* Lecture Notes in Artificial Intelligence 422. Springer Verlag, Berlin, West Germany, 1990.
[22] G.T. Nguyen and D. Rieu. Schema evolution in object-oriented database systems. *Data and Knowledge Engineering*, 4:43–67, 1989.
[23] R. Prieto-Diaz. Implementing Faceted Classification for Software Reuse. *Communications of the ACM*, 34(5):88–97, 1991.
[24] R. Prieto-Diaz and P. Freeman. Classifying Software for Reusability. *IEEE Software*, 4(1):6–16, 1987.
[25] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modeling and Design.* Prentice Hall Inc. Englewood Cliffs, 1991.
[26] G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In *Proceedings of the ACM SIGSOFT 6th International Symposium on the Foundations of Software Engineering (FSE-98)*, pages 99–110, Lake Buena Vista, FL, USA, 1998.
[27] R.R. Sokal and P.H.A. Sneath. *Principles of Numerical Taxonomy.* Freeman, San Francisco (CA), USA, 1963.
[28] P. Steyaert, W. Codenie, T. D'Hondt, K. De Hondt, C. Lucas, and M. Van Limberghen. Nested mixin methods in Agora. In *Proceedings of ECOOP'93*, Lecture Notes in Computer Science 707, pages 197–219. Springer-Verlag, Berlin, 1993.
[29] P. Steyaert and W. De Meuter. A marriage of class- and object-based inheritance without unwanted children. In *Proceedings of ECOOP'95*, Lecture Notes in Computer Science 952. Springer-Verlag, Berlin, 1995.
[30] J.P. Sutcliffe. Concept, Class, and Category in the Tradition of Aristotle. In I. Van Mechelen, J. Hampton, R.S. Michalski, and P. Theuns, editors, *Categories and Concepts. Theoretical Views and Inductive Data Analysis*, pages 35–65. Academic Press, London, 1993.
[31] R. Wille. Concept lattices and conceptual knowledge systems. *Computers Math. Applic*, 23:493–513, 1992.