INSTANCE ORIENTATION: A PROGRAMMING METHODOLOGY

Thomas Schöbel-Theuer, Universität Stuttgart, Germany, email: schoebel@informatik.uni-stuttgart.de

ABSTRACT

Instance orientation is an approach for designing and programming software systems. It addresses a limitation of current software architectures: it allows multiple higher-level views *in parallel* on a running software system by non-destructive (virtual) *transformations* between them. Different views on the system and its references alleviate modularity, stability, protection, and flexibility. Examples for multiple views are network transparent views, fault tolerant views, security-enforced views, and more. Self-control (e.g. organic computing) can be achieved by specialized configurations. Instance orientation may be combined with other programming methodologies, in particular OO, to compensate for some of their deficiencies. We explain how to simulate the central idea of Aspect Oriented Programming by instance-oriented transformers.

KEY WORDS

Software design and development, programming methodology, programming discipline, software architecture.

1 Introduction

Current software design methods tend to describe systems and the relations of their objects / components / instances in a single way or in a single architectural model. While objects and their relations are statically modelled in a well-disciplined way, the *control* over *runtine* relations between instances is *itself* not structured in an *independent* way from class or object structuring, but often spread over the whole system in the same way as the classes or objects are spread. Control over runtime relations has often no other internal structuring than given by the relations at class or object level. A rather fixed correspondence between object relations and control over object relations leads to a single view at the system structure (although different granularities of reasoning on that model are possible). This is a *general limitation* which does not seem to have been recognized until now.

By removing this limitation, we open the door to easier modelling of complex systems, to more flexible control over system structures, to easier introduction of multiple views at the system in parallel, and in effect to *lower engineering costs* when compared to achieving that functionality with conventional methods not capable of modelling that explicitly.

To resolve the limitation, we propose a programming methodology called ”instance orientation”. The key idea is (1) to *separate runtime control* over *references* or *relations* between instances *and then* (2) creating *multiple views* on that reference structure by means of non-destructive *transformers*. While the prerequisite principle (1) has been already in use (e.g. in coordination languages, or in pipe-and-filters style in software architecture [1]), the central principle (2) does not seem to have been widely used until now.

Instances in the sense of this paper may be objects in OO sense, component instances, or any other type of instantiated systems to which a rudimentary form of *identity* can be attributed.

(1) Separation of runtime control over references means that an instance does not know the identity of related instances (anonymity of relation), and thus cannot directly influence its relations. Although the instance may be active (currently doing work), it does not know for whom it is working, and cannot gain influence on that fact. This and the separation of runtime control over references leads to increased modularity, stability, protection, reliability, resilience, testability, and even flexibility at the level of application logic (e.g. compositorical genericity [2]).

(2) *Multiple views* on the objects and their reference relations may exist in parallel. In particular, non-destructive *transformations* between different views may be employed to create *virtual views*. For example, there may be views on instance networks abstracting away from concrete instance locations in a network of computers. Other views may hide technical details such as connector objects; others may provide different programming interfaces to the system (masquerading). Virtual views may even create totally different higher-level pictures of the system having ”virtual” properties. For example, multiple computer systems may be integrated with each other to form a single virtual system providing virtual services. Each view may be created and maintained by a different transformer instance. Parallel and sequential composition of transformers is the key to increased modularity, testability, stability and flexibility.

While classical modularity relations are defined on the *static program code* level, our relations between instances are defined on a potentially unbounded number of individual instances at *runtime*, where relations may be *dynamically altered* (similar to pointer assignment, but caused by a *separate* control instance, not by the objects themselves). In contrast, traditional OO systems usually mix up constructors and ordinary methods which are responsible for maintaining references, and they spread them often over dozens or even hundreds of classes. Instance orientation requires the *principle of locality* for object relations *in their totality*. On that basis, non-destructive transformers are operating on the object relations in parallel,

creating and maintainig different *views* of the system.

Section 2 shows up some locality problems in current OO systems. Section 3 shows some examples for instance orientation from a novel operating system design. In section 4, some hints are given for applications of instance orientation to OO systems and designs; further research will be required for a full adaptation of current software engineering methods to instance orientation. Section 5 discusses related work, and section 6 concludes with open problems.

2 Some Problems with OO

Object orientation (OO) is a rather old and well-understood discipline in computer science. Most OO design and programming methods focus on a *class structure* and on relations on it, like "is-a" and "has-a" relations. Reasoning is done almost always on class level, but rarely on instance level. There are some exceptions, e.g. newer UML versions may also describe relations between instances, but this is neither the common case, nor is it generally applicable to situations where concrete instances are not known at design stage.

While nearly all OO methods spend a lot of effort on reasoning about relations at class level as well as controlling and restricting relations at that level, many of them do nearly nothing at the instance level. Object instances usually contain *pointers* or *references*, which point to other instances[1]. Pointers or references are usually *typed*. Type systems bear some useful information (e.g. the superclasses of a named object instance may be determined). However, there is usually no restriction on the concrete instance (identity) which is named by a reference; it may in fact name *any* instance of the designated type (or a subtype). References have the *purpose* of naming other instances.

First notice a trivial fact: reference values are values like any other values such as number values or string values. Values are usually *computed* by a program which is formulated in a Turing-complete programming language. As a consequence, the values computed at runtime are in general *undecidable* for a given piece of code. This also applies to reference values in general.

OO systems tend to produce large networks of interconnected object instances at runtime. Since reference values are kept inside object instances as part of their state information, it is in general unpredictable from a theoretical point of view, which instance will point to which other instance at runtime. Although programmers have learned to write code with rather "predictable" behavior, the theoretical concept of undecidabiliy shows up in the presence of *bugs*. Finding bugs may be hard. Chances for finding bugs increase with the *simplicity* of a software structure. But is it really simple in a larger OO system to trace down why a particular reference has a particular (unexpected) value? In many cases, the answer will be "No".

The reason is rather simple: the *responsibility* for correctness of reference values is usually spread across *many* classes in OO systems. Although each class is usually responsible for maintaining its own references, the side effects between instances at runtime may be much harder to understand. At runtime, the code of one class calls the code of another class with parameters that are computed on the fly (cf. "law of demeter" [3]). It is extremely hard to understand in detail what is going on in a complex OO system. As a result, the "unpredictability" (from a human point of view) of subroutine parameter values extends to "unpredictability" of reference values which are then stored in the object instances. In practice, this problem shows up when programmers cannot help themselves any more but insist on using a debugger.

Instance orientation requires as a prerequisite that the *principle of locality* [4] is applied to references in their totality. This ensures that the responsibility for correctness of references is assigned to few well-known places, and that it is rather easy to reason about the correctness of references. Enforcement of restrictions on reference relations is centralized. Moreover, instance-oriented transformations on views allow stepwise refinements, e.g. via intermediate views.

3 An example from the operating systems area

We explain the idea of instance orientation by an example from the operating systems area, where our original motivation comes from. We have developed a LEGO[2]-like lightweight component architecture for operating systems with a uniform interface type [5, 6, 2, 7]. Currently, we are working on a prototype implementation called ATHOMUX. The filesystem part is already working. According to figure 1, the architecture may be characterized as a "pipes and filters style" in the sense of software architecture [1].

The wires in figure 1 may be viewed as "transportation channels" which *logically* transport instances of a universal memory abstraction called "nest"; as a simplified metaphor the reader may think of it as an access channel for a (not necessarily persistent) file. The boxes are called "bricks" and may be characterized as "transformers" between nest *instances*; as a simplified metaphor, the reader may think of it as a unix process transforming file data or pipe data (but non-destructively as explained later). Brick instances are depicted with inputs at their left and outputs at their right, similar to functional units in electrical engineering or automation control [8]. Wires are directionally drawn from left to right. The "contents" of a nest is computed *on demand* in an incremental fashion, whenever an *operation* (e.g. a "read" or "write" operation) is called

[1] Folklore says that pointers are the "gotos" of data structuring.

[2] Brand names like LEGO are the property of their respective owner.

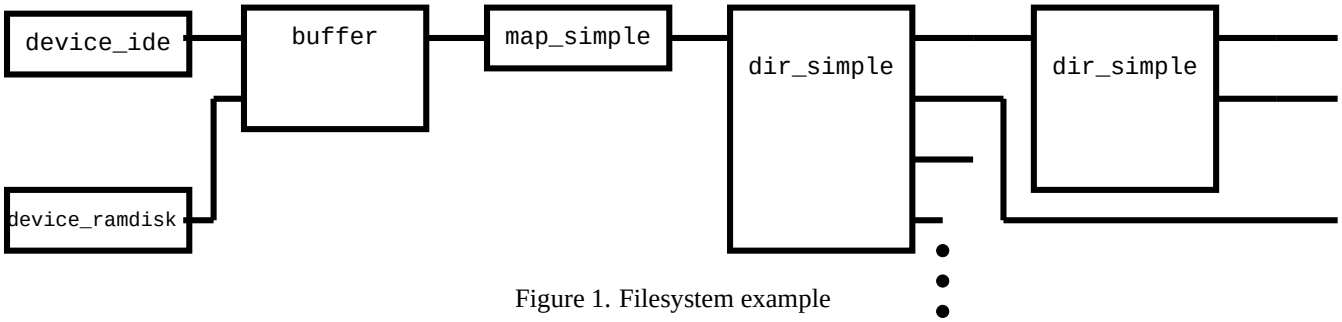
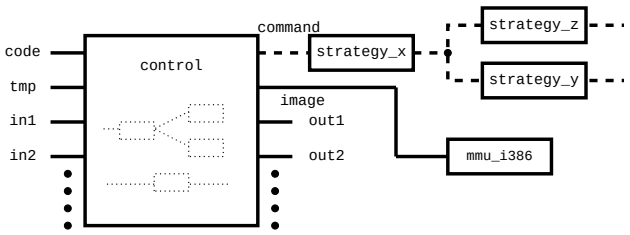


Figure 1. Filesystem example

at an output; often further operation calls will then occur at the inputs and be forwarded to other instances by the wires. By dynamic wiring, operation calls lead to anonymous connection-oriented directional communication (as opposed to OO, which employs connectionless undirected communication, often on known partners). Note that communication normally goes from right to left, in the opposite direction of logical transformation / transportation of the nest "contents".

Our "pipe and filters style" differs from known implementations of that style in the operating system area, such as UIO [9] or stackable filesystems [10], in a number of ways. First, it has no "consuming" semantics like pipes, where processing of data would "destroy" the old data and produce a new version instead. Rather, a brick adds to the ways we may look at the data, by providing a non-desctructive new *view* on the data: both the old view and the new view provided by the transformation will exist in parallel; the "old" view may for example be used by parallel wiring to other "consumers" or "clients". Data modifications carried out in one view (e.g. by "write" operations) will usually be immediately reflected in the other view (e.g. when "reading" the "modified" nest "contents"); a transformation may thus be characterized as logically acting bidirectionally. Second, we use nests as a *universal* (but nevertheless rather simple) abstraction on all layers of the whole operating system, not limited to filesystems (in contrast, stackable filesystems are based on conventional filesystem abstractions such as `inodes`, `vfs`, directory (sub)trees, etc). We take advantage of the fact that bricks may be instantiated dynamically at runtime, resembling the dynamic nature of filesystem subtrees directly by recursive instantiation of `dir_*`-bricks. Third, instance generation and maintainance is done by an instance-oriented recursive and self-describing methodology as sketched:



The idea is to create and maintain brick instances of any type by a special brick called `control`. In the above picture, a `control` instance does not create other brick instances and their wires directly, but rather creates an `image` which may be considered a kind of "process image" "containing" the instances and their wiring. In order to really "execute" that network, some `mmu` instance like `mmu_i386` has to follow after (possibly indirectly). This leads to a *physical* separation between the controlling instance for the wiring (relations between instances) and the actual environment where the instances are executed. In our ATHOMUX prototype, we implemented a `control_simple` without `image` output, running in an ordinary Linux process, but separating both levels logically. Our brick sourcecode is designed to run in other `control_*` environments without alteration.

Transformers in the sense of instance orientation are implemented by `strategy_*` bricks, connected by dashed wires in the picture. The `command` output of `control_*` is a nest instance containing some *abstract representation* of the instances network. In our ATHOMUX prototype, we chose ASCII strings describing instances, inputs and outputs, and their wiring in a C-like syntax. This simplifies manipulations and transformations of virtual brick networks by searching and replacing via regular expressions. By "writing" such representation strings, (de)instantiation of bricks, (dynamic) inputs/outputs, and wires are performed.

In general, an abstract representation of a (virtual) instances network can be both queried and manipulated via the dashed control wires. Thus a dashed control wire provides a controllable *view* on the system structure and the interrelations between instances. There may exist multiple views *in parallel*. In the picture, different `strategy_*`-instances are non-destructively *transforming* between views or creating virtual views which need not exist in "reality". Note that the wiring of `strategy_*`-instances may itself be controlled by `strategy_strategy_*`-instances and so on, forming a hierarchy[3] of control levels.

In our ATHOMUX prototype, we implemented a strategy brick called `fs_simple`. It "creates" a virtual

[3] At some certain level the hierarchy will either terminate by a level which controls itself, or at a level where never any modifi cations are necessary (e.g. conventional bootstrap mechanism).

brick instance `fs` with a variable number of outputs, each output having a complete path name as an instantiation parameter attribute, and translating that path name to a series of hierarchically instantiated `dir_simple` instances according to figure 1. We plan to simulate Unix `mount()` operations by an appropriate `mount_simple` strategy brick, creating some virtual union of other `fs_simple` or `mount_simple` instances. By parallel use of multiple `mount_simple` instances, we may easily provide each application with a different "mount table" (speaking in conventional sense, although there is no concept of a system-wide global table anymore), each corresponding to a different instance-oriented view of the system.

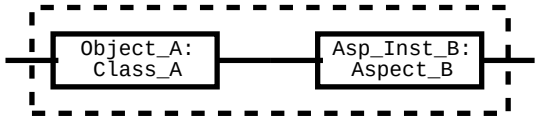
As an example for other instance-oriented views, we may implement `strategy_transparency` for providing network transparency, by automatically inserting `remote`-instances into brick networks wherever it becomes necessary to span several hosts on a computer network, and by providing a single virtual system image. The automatically inserted `remote`-instances may be completely hidden from users of `strategy_transparency`, so that they get the virtual impression that no network would exist at all, and as if everything would execute on a single "virtual computer".

As another example, we may employ translation of machine code similar to *code morphing* [11] to create hardware platform transparency. On the "virtual computer" level, we may provide a view in which hardware specific details are hidden. When combined with network transparency, it should not make any difference whether one buys a SPARC or an Intel computer and connects it to the network.

Note that there is often a need for different views in parallel. For example, a service technician may need a hardware-dependent view in order to locate a faulty device. Similarly, a network supervisor may be interested in views of the physical network configuration. The end user will be frequently interested in *reduced views* omitting unnecessary details. Instance orientation provides a very general means for (dynamically) creating and manipulating such views.

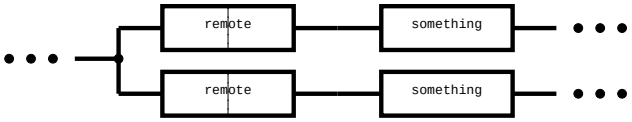
Further examples for `strategy_*` are automatic adaptation to varying workloads (load balancing), to unpredictable faults (fault tolerance), to recoverability (transactions), or centralized enforcement of security policies by automatically inserting `check_*` instances at strategic places.

As another demonstration for the power of instance orientation, we point out how to simulate a key feature of aspect-oriented programming (AOP, [12]; we assume the reader to be familiar with it): assume that in AOP an object class `Class_A` is combined with an Aspect class `Aspect_B`. In our model, we construct a `strategy_combine_A_B` which modifies any command for instantiation of bricks representing objects of type `Class_A` in such a way that the following pair of brick instances is always created in place of `Object_A`:

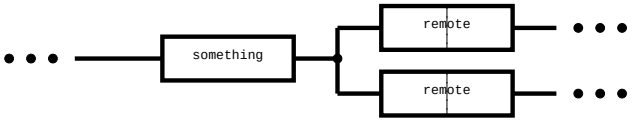


From a virtual viewpoint as indicated by the dashed surrounding box, the pair of instances (`Object_A`, `Asp_Inst_B`) having types (`Class_A`, `Aspect_B`) plays the same role as was originally planned for `Object_A` alone. By transparently pairing with `Asp_Inst_B`, we can dynamically create a combined functionality similar to AOP[4].

Finally, we look at another example where a key property of OO is relaxed: the notion of *identity* is softened. In a wired network based on *stateless* or *pseudo-stateless* brick instances, state is not represented by brick instances, but rather *explicitly* kept in nest instances. There may exist nest instances which are *equivalent* to each other (i.e. they form *aliases* in logical sense), although they are not identical. As an example, look at the following picture:



Because of the properties of statelessness, this is *functionally equivalent* to the following network:



Functional equivalency could serve as a key property for automated reasoning on instance oriented views. We may create `strategy_*` bricks transforming one configuration variant into another, depending on knowledge on the problem domain, on environmental properties, and on actual measurement data (e.g. for current load). Such transformations may for example be governed by rule-based methods, on an automatic basis[5].

In summary, creating multiple views on basis of separation between instances and their relations enables advanced functionality in a modular way. Conventional notions such as state and object identity are not vital for instance orientation, and may be relaxed, leading to more general and more flexible designs.

[4] In contrast to most incarnations of AOP, the combination need not be executed at compile time, although (dynamic) code-generation for a combined `Class_A_Aspect_B` brick type in the style of macro or inline expansion and automatic replacement by an appropriate `strategy_replace` is possible in our system (as well as for performance improvement on arbitrary but static subnetworks).

[5] In general, we may run into undecidability problems when using graph rewriting methods [13]. However, in practice the problem is often not to *recognize* functional equivalency, but rather to *create* it in a well-understood way.

4 Instance orientation and OO

Our examples from the previous section have drawn from the opportunity of an operating system designer to create own execution environments, and to apply execution environments recursively to themselves. Normally an application software engineer will have to rely on preexisting execution environments such as the Java or a C++ runtime environment.

First, *direct* use of pointers in the state of objects has to be eliminated. This could be achieved by exclusive use of signal-slot-mechanisms [14] or other OO incarnations of anonymous communication. Instead of storing pointers, we propose to inherit from a general link class, which will provide hidden parts of the necessary infrastructure for separate link management. Instead of calling other object's methods with `this->reference->method(args)`, a pointer to a function `this->reference_method()` should be provided[6] in the object's state. These pointers should be used only for indirect method *calls*, but never be maintained by the object itself. Management of pointer values is the task of the `control_*`-level. A `control_*`-instance will need to manage *heterogenous collections* of object instances in general.

As a consequence, passing around of object pointer values between object instances of the same level of control will become meaningless. Note that higher levels of control will have to deal with references belonging to lower levels, so they may pass around such references for the sole purpose of maintaining them. Interfaces will have to concentrate on services at a higher level, but should not depend on values of runtime references of the same level in any way.

In a pure instance oriented system, constructors for other objects should never be called from "normal" application code. Constructors may exist, but they should initialize nothing else but the "local" state of the instance. The logic for *calling* those constructors should be completely separated[7] and put into `strategy_*` classes.

Self-modification of behaviour (e.g. self-adaptation to varying demands) such as propagated by newer trends like "organic computing" remains possible, by *explicit* wiring of a command channel from application logic level to the `strategy` level. A good design should do that only in few well-chosen places. *Self*-control is a special case of more general instance-oriented separation of control.

5 Related work

In OO systems, separation between application and instantiation logic is principally possible with some *generational design patterns* [15] like Abstract Factory. However, other generational patterns like Builder or Factory Method will usually intermix logic for constructor calls with normal application logic. In practice, complex OO systems don't separate both levels consistently and uniformly. Consequently, multiple views on the system *as a whole* are hard to implement. However, appropriate patterns may be developed to combine instance orientation with OO. In addition to adapted patterns, automated checks for discovering inadvertent violations of instance oriented separation should be introduced.

Reflective[8] *systems* [16, 17] may supersede references at runtime similar to a *single* instance-oriented view. The key idea of reflection is to provide an *interpreter* for object behaviour on a meta-level, a so-called *meta-object*, which can be modified by the application itself. In a rough comparison, meta-objects could be seen in a similar role to our `control` level; a meta-meta-hierarchy is also possible (analogously to our strategy hierarchy). However, reflective systems provide no explicit means for maintaining *multiple* different meta-objects for a base object in analogy to multiple views at our strategy level. Reflective systems are intended for modification of the *one default* runtime behaviour of (self-maintaining) objects, while each instance-oriented view may expose a different *virtual behaviour*. Simply stated, our strategy level is not an *interpreter*, but rather an *organizer* for the lower level of control, which may co-exist with other organizers. While meta-objects remain *continually* responsible for alteration of default behaviour, our strategy level need no longer be active (or even exist) when a system does no longer require reconfiguration. Altering object behaviour by meta-object interpreters often requires whitebox intrusion, while instance orientation prefers blackboxes from start. Finally, instance orientation may not only be applied to stateful OO systems, but also to (pseudo-) stateless brick networks with a relaxed notion of identity.

Interrelations to aspect-oriented programming (AOP) [12] have already been mentioned. We view AOP as a super-principle over OO: the key difference between an object and an aspect is that one may create only pointers to objects, but not pointers to aspects. In all other respects aspects are similar to objects, in particular they may embody state and behaviour, and there may exist inheritance relations between aspect classes. As our examples suggest, instance orientation can simulate central features[9] of aspect orientation (possibly treating bricks as whiteboxes). Aspect-oriented designs primarily focus on properties of the *static* program text, and were invented for easier handling of cross-cutting concerns in large software systems at design time. In contrast, instance orientation focusses on (automated) reasoning on the system structure at *runtime*. Transformations by `strategy_*`-instances may lead to

[6] In ideal case, these slots could be generated automatically, e.g. from interface specifications of other classes.

[7] In general, either a complete refactoring of an existing OO design is required, or, preferably, an instance oriented design from scratch. Good design methods for instance oriented systems are yet to be developed.

[8] As the name *reflection* suggests, *self*-modification of behaviour seems to be the main motivation, while in instance orientation self-modification is just one of many possible configurations.

[9] We are not sure whether the full spectrum of aspect-oriented features as described in [18] can be simulated (we leave that question for further research).

system structures which could not be anticipated at design time or system generation time. AOP does not *explicitly* deal with different views at the structural level in parallel. Thus, instance-oriented runtime transformations are more general than aspect-oriented object composition.

The principle of locality for instantiation logic or connection logic has certainly been used already in practice. When a Unix shell invokes filter processes and interconnects them with pipes, this may be seen as an abstract `control_*` unit; the shell script governing the whole process may be seen as an abstract `strategy_*` unit. In application areas of "pipe and filter" architectural style (e.g. digital signal processing), functional units are interconnected with wires. Instance orientation may be seen as an extension by *explicit* and *automated* reasoning on the instance network level by *transformations* leading to *multiple views*.

In reconfigurable hardware architectures [19, 20], configuration memories play a role similar to our `control` level. However, most research in the field focusses not only on hardware-specific problems [21], but deals also with dynamic wiring of a *fixed* number of preexisting hardware devices, while in our software system we may instantiate a potentially unbounded number of runtime instances of any type, even of dynamically generated types (e.g. by dynamic generation of `code` input for `control` instances). Probably, instance orientation could be well-suited for future data-flow-like machine architectures, employing "virtual" hardware devices simulating a potentially unbounded number of physical devices. The latter would correspond to virtual addressing in von-Neumann machines which intends to simulate potentially unbounded memory in spite of bounded physical memory.

Separation of application logic from instantiation logic should be principally possible with some coordination models [22, 23], although original coordination languages like Gamma have been developed for coordination of nondeterministic control flow, not for coordination of (persisting) relations between (persisting) object instances. Additionally, research in that field seems to have focused on *single-level* images of relations between objects where *replacement* and *rewriting* (or sometimes *accumulation of results*) occurs in a *single* shared coordination medium, even when applying parallel and sequential composition on the (static) *programs*. Applying instance orientation to coordination would mean to create non-destructive transformations between multiple *coordination media* instances, and to control that transformation process in turn by enhanced coordination.

In essence, instance orientation deals with parallel and sequential composition of *runtime instances* (not restricted to static program code), and does explicit reasoning and runtime controlling on that composition by means of one recursive methodology.

6 Conclusions

We have presented the idea and some examples for benefits of instance orientation in a specialized application area. We have argued that non-destructively creating and maintaining views at separated controlling levels over instance relations bears a high potential for increasing modularity, stability, protection, and flexibility, and reduction of software engineering costs at large and/or complicated systems.

We have pointed out that instance orientation may simulate central features of AOP. Specialized configurations for organic computing are possible. We have presented examples where the notion of object identity was relaxed, leading to a more general and more flexible system design. We have presented examples for instance oriented transformations even on *stateless* "objects". Conventional notions such as state and object identity are not vital for instance orientation.

Instance orientation is basically independent from OO and thus combinable with it. In some respects like sequential and parallel composition of instances, it is more similar to functional programming [24], but also allows for *stateful* dynamic reconfiguration.

Further research is required on instance orientation and on instance-oriented design methods. OO has undergone years of research and improvement of methods; instance orientation is at its beginnings. Substantial claims on pros and cons of instance orientation and on its applicability to various areas can be made only after intensive research; explorative and comparative studies are highly appreciated. We have thus refrained from making any general claims in comparison to other programming methodologies, but rather presented the principle and demonstrated by some examples that it can lead to obvious advantages in some places.

References

[1] M. Shaw and D. Garlan, *Software Architecture, Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[2] T. Schöbel-Theuer, "On variants of genericity," in *Proceedings of the Fifteenth International Conference on Software Engineering & Knowledge Engineering (SEKE 2003)*. Knowlegde Systems Institute, 2003, pp. 359–365.

[3] K. J. Lieberherr, I. Holland, and A. J. Riel, "Object-oriented programming: An objective sense of style," no. 11, San Diego, CA, September 1988, pp. 323–334, a short version of this paper appears in *IEEE Computer Magazine*, June 1988, Open Channel section, pages 78-79.

[4] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *CACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[5] T. Schöbel-Theuer, "Skizze einer auf nur zwei abstraktionen beruhenden betriebssystem-architektur: Nester und bausteine," Arbeitspapier und Vortrag auf dem Herbsttreffen der GI, Fachgruppe Betriebssysteme, Berlin, 7. – 8.11.2002.

[6] ——, "A way for seamless integration of databases and operating systems," in *Proceedings of the International Conference on Computer Science and its Applications (ICCSA-2003)*. National University, 2003, pp. 82–89.

[7] ——, "A lego-like lightweight component architecture for organic computing," in *to appear at the GI workshop on organic computing*. GI Jahrestagung, 2004.

[8] "Iec standard 61131-3," http://www.holobloc.com/stds/iec/sc65bwg7tf3/html/news.htm.

[9] D. R. Cheriton, "Uio: A uniform i/o system interface for distributed systems," *Transactions on Computer Systems*, vol. 5, no. 1, pp. 12–46, 1987.

[10] J. S. Heidemann and G. J. Popek, "File-system development with stackable layers," *Transactions on Computer Systems*, vol. 12, no. 1, pp. 58–89, 1994.

[11] "Code morphing," http://www.transmeta.com/technology/architecture/code_morphing.html.

[12] G. Kiczales *et al.*, "Aspect-oriented programming," in *European Conference on Object-Oriented Programming (ECOOP)*, ser. LNCS 1241. Springer-Verlag, 1997, http://www2.parc.com/csl/groups/sda/publications/papers/Kiczales-ECOOP9%7/for-web.pdf.

[13] M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, Eds., *Term Graph Rewriting: Theory and Practice*. Wiley, 1993.

[14] "Signals and slots," http://www.trolltech.com/products/embedded/whitepaper/qt-embedded-white%paper-5.html.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[16] P. Maes, "Concepts and experiments in computational reflection," in *Conference on Object Oriented Programming Systems Languages and Applications*. ACM SIGPLAN, 1987, pp. 147–155, http://doi.acm.org/10.1145/38765.38821.

[17] Y. Yokote, "The apertos reflective operating system: The concept and its implementation," in *OOPSLA'92 Proceedings*. ACM, 1992, pp. 414–434, sony CSL technical report SCSL-TR-92-014, ftp://ftp.csl.sony.co.jp/CSL/CSL-Papers/92/SCSL-TR-92-014.ps.Z.

[18] G. Kiczales *et al.*, "An overview of aspectj," in *European Conference on Object-Oriented Programming (ECOOP)*, ser. LNCS 1241. Springer-Verlag, 2001, http://aspectj.org/documentation/papersAndSlides/ECOOP2001-Overview.pdf%.

[19] S. P. Kartashev and S. I. Kartashev, *Designing and Programming Modern Computer Systems*. Prentice Hall, 1989, vol. II: Supercomputing Systems: Reconfigurable Architectures.

[20] H. Li and Q. F. Stout, Eds., *Reconfigurable Massively Parallel Computers*. Prentice Hall, 1991.

[21] R. W. Hartenstein and H. Grünbacher, Eds., *Field Programmable Logic and Applications, 10th International Conference FPL 2000*, ser. LNCS 1896. Springer Verlag, 2000.

[22] P. Ciancarini and C. Hankin, Eds., *Coordination Languages and Models*, ser. LNCS 1061. Springer Verlag, 1996.

[23] D. Garlan and D. L. Metayer, Eds., *Coordination Languages and Models*, ser. LNCS. Springer Verlag, 1997.

[24] A. J. Field, *Functional Programming*. Addison-Wesley, 1988.