2nd Workshop on Object-Oriented Language Engineering
for the Post-Java Era: Back to Dynamicity

The Unavoidable Failure of Class-Based Languages in the Processor Cloud Era

Sebastián González[2], Wolfgang De Meuter[1]
Kim Mens[2], Theo D'Hondt[1]

1. Programming Technology Lab
Vrije Universiteit Brussel, Belgium
{wdmeuter,tjdhondt}@vub.ac.be

2. Département d'ingénierie informatique
Université catholique de Louvain, Belgium
{sgm,kim.mens}@info.ucl.ac.be

April 20, 2004

Abstract

Classes are problematic in pervasive computing — systems we call "processor clouds" — where computers appear and disappear dynamically in a network. Problems with classes are mainly due to the dynamically distributed nature of these systems. The root cause of the problems is that the class-instance relationship is implicit. Two main manifestations are: state sharing vs. distribution conflicts, and inefficiencies caused by the binding of classes and their transitive closure to objects. The latter is worse in the case of statically typed languages.

1 Introduction

More than a decade ago, the combination of wireless networks and miniaturisation started buzzing people's minds with the possibility of interconnecting heterogeneous computers and embedding them in the human habitat, such that they would assist human activities by interacting in the most intricate ways. This attractive vision for the future of computing has received many names, each one with its own nuances: *ubiquitous computing* [Wei91], *pervasive computing* [Sat01], the latest (European) term *ambient intelligence* [Sha03], and more.

We are still expecting the paradigm shift from semi-isolated computers (e.g. client-server models using static networks), to heterogeneous devices interacting in dynamic networks. The enthusiasm has stood this many years as technology seems to converge more and more to the point of making pervasive computing possible. On the software side, we observe progress on the development of platform infrastructure: service discovery, checkpointing, migration, replication, etc., i.e. progress on middleware. Yet not a lot of effort has been invested on programming technology and software engineering: programming languages and composition mechanisms for the new computing environments, which we like to call "processor clouds" — hence the title of the paper.

Currently we are researching the application of object-based programming [Weg87] to processor clouds. This family of paradigms has members such as the actor-based, prototype-based and class-based paradigms. The least common denominator in the family is naturally the concept of *object*, i.e.

encapsulated state + message passing

In the encapsulation metaphor, state is "covered by a capsule" (an *interface*). Two objects differing in their internal representation but having a common interface look the same from the outside, and are thus interchangeable (*polymorphic*).

Object-based programming, in this broad sense, is considered to be well suited to the programming of processor clouds, for the following reason:

> Objects help abstracting semantically coherent software units by merging state and behaviour, the two essential ingredients in computer alchemy. Objects are easy to think about, and thanks to encapsulation and polymorphism are prepared for highly dynamic environments where emphasis is put on unanticipated interaction.

We agree on using object-based programming as a possible approach to processor cloud programming, but we see a major pitfall in attempting to use the class-based sub-paradigm exclusively. Our bet is that *classless* object-based technologies will make a programmer's life easier. A second hypothesis is that, in the spectrum of possibilities, dynamic typing will play a central role as well, but here we will treat this hypothesis tangentially. Our goal is to explain why we believe that class-based programming is doomed to fail in the context of processor clouds.

2 The Fundamental Problem

> "Distribution transparency is impossible to achieve in practice, and precisely because of that impossibility, it is dangerous to provide the illusion of transparency." [GF99]

The main problem with class-based languages for OO distributed programming is the implicit, unavoidable *class-instance* relationship, which becomes explicit under distribution. The kind of distribution we consider here is *dynamic distribution*, meaning that

- the distribution of objects among nodes can change, i.e. objects may migrate, and
- the network composition varies randomly, i.e. network nodes may appear and disappear.

The cost of keeping the class-instance relationship consistent (and thus implicit) behind the scenes in a dynamic distributed system is prohibitive. Furthermore, the programmer has to deal with the consequences of such relationships all the time, yet she has no way to control and manipulate them, or to avoid their use in specific situations.

The following sections show concrete incarnations of the problem.

3 Sharing vs. Distribution

Classes have many roles in class-based languages [BL92]. Among others,

- modelling of domain concepts (Scandinavian school),
- hierarchical organisation of these concepts (classification),
- instance description and creation,
- typing, and
- *static, implicit sharing mechanism of state and behaviour* [SLU89].

This role overloading makes classes conceptually less clear and difficult to manage. But the main problem concerning their use in distributed systems is the last one, i.e. that classes are a resource sharing mechanism. Their nature calls for the well known conflicts between sharing and distribution:

- When shared data is kept centralised, the node containing the data becomes a bottleneck, and worse, the entire system is fragile since there is a dependency on that single node — this leads to partial failure management problems.
- If, on the other hand, the data is replicated among multiple nodes (e.g. to increase robustness), the system must keep all the copies synchronised — a replica management problem.

Both problems are hard and sometimes unavoidable — but we should not call for them unnecessarily. In the case of class-based programming, classes must be either centralised or replicated in every node. Both alternatives harm the operation of a processor cloud.

We do not argue that information sharing can or should be avoided in distributed systems: in fact it is often essential. We *do* argue that, by using classes, trouble is taken *a priori* — trouble we cannot

avoid since instances cannot exist without their classes. Two concrete manifestations of the sharing vs. distribution conflict in classes are shown next.

3.1 State Sharing: Class Variables

A patent manifestation of the sharing vs. distribution problems is class variables (a.k.a. static variables). The majority of current class-based systems overlook this problem. The semantics of class variables are not enforced in the presence of distribution: copying a class containing class variables from one node to the next does not start an underlying replica management system that would keep all the variable copies synchronised. The semantics of class variables is thus *broken*. Worse, it cannot be fixed since having an underlying replica management mechanism is impossible in processor clouds: if two devices go out or reach and they update a class variable with different values and then they rejoin the network, the inconsistency cannot be solved.

The alternative of centralising classes is also not possible in processor clouds: in these dynamic networks there cannot be a central authority, e.g. suppose a device goes out of reach; who would be its central authority? it would have to stop operating until rejoining the network, wich is unacceptable.

One might reply to this state sharing argument by eliminating class variables from class-based programing, which is feasible. This would fix the problems presented so far. But the problem is in fact the sharing of *any* resource. As shown next, similar problems arise with code sharing — one of the principal roles of classes. One can eliminate class variables from class-based programming as suggested before, but one cannot "fix" the paradigm by overruling methods!

3.2 Code Sharing: Class Methods

Class variables are one part of the shared information in a class. Method implementations are the second part, they are also shared among class instances.[1] The problem of class variables is just the same for methods. Suppose a node receives the same class along two different paths, but the two versions have a different implementation for a method... which one is to be considered correct?

In class-based technologies, classes (e.g. class libraries) are often replicated among nodes without an appropriate replica or version management mechanism. The effects of distribution are somewhat relaxed by the standardisation of the basic classes (for instance all the *java.** packages in Java). Since a copy of the basic classes can be assumed to exist in every node, replication consistency is guaranteed. It is clear though that this is a partial solution only, as any user-written class breaks the replication harmony.

Even for standard libraries, there might be many versions of a same class circulating in the system (e.g. JRE 1.1 vs. JRE 1.2). This issue can be solved by backwards-compatibility, i.e. by marking obsolete methods "deprecated" as in Java. This way, given any two versions of the same class, one class will be a subset of the other (i.e. the newer class will contain all the methods of the older class, some possibly deprecated). The interference problem is solved then by choosing the newest version, which is supposed to be compatible with the older one. This solution is weak and ad hoc, as classes would grow forever, full with deprecated methods, becoming incontrollable entities. And even if only a few methods are deprecated, sharing this legacy code all the time is inefficient.

3.3 Workarounds

A solution is to make classes constant, so that replicas can be distributed without any synchronisation issues arising. This not only implies turning class variables into class constants (which is more or less equivalent to overruling class variables from class-based programming, as suggested before), but also freezing method implementations, which are also part of the class state. The consequence is that *every* change in the implementation of a class would forcibly imply the introduction of a *new* class in the system. This solution is similar in spirit to the approach used by Microsoft's COM interfaces. Defining a new class each time implies making existing instances incompatible with the new version. This again brings problems.

[1] Changing a method implementation is analog to changing a class variable value. Either dynamically (e.g. in Smalltalk) or statically (e.g. by editing source code and recompiling), methods can be modified. The dynamic/static distinction doesn't add to the discussion in a distributed setting: the only important point is that nodes might have different definitions of a given class at the same point in time.

4 Transitive Closures

Classes have two possible links, the implicit *instance-of* link, of which some consequences were mentioned in section 3, and the *subclass-of* link. Another consequence of the *instance-of* link is that, upon transmitting an instance over the wire, the corresponding class must be sent too. The *subclass-of* link forces the recursive transmission of all the superclasses as well, for the object wouldn't be well defined otherwise (the set of methods and attributes would be incomplete).

In a dynamically typed language (e.g. Smalltalk), the transitive closure problem ends here. But in a statically typed language, argument-type classes, result-type classes and exception-type classes must be transmitted also — together with their corresponding transitive closures.

Classless systems do better. Conceptually, objects are transmitted complete, self-contained over the wire, with their state and behaviour together. Technically, caching mechanisms could be used to avoid the extra overhead of repeatedly sending the same behaviour. The big difference is that no meta-objects (e.g. classes) need to be sent, together with a ballast of related classes due to transitive closures and (depending on the language) static typing.

In one word, regarding mobility, classes are too heavy a drag on objects.

5 Conclusion

In class-based programming, each time one uses an object, two objects are actually involved, the instance and its class. Thus acquainting an object implies acquainting its class also, and necessarily the entire superclass chain. In statically typed languages, it implies acquainting also every parameter-type, return-type and exception-type classes — together with all the associated superclass chains.

People feel that large object-based systems would be very difficult to manage because of the lack of organisational structure. Advocators of object-based systems reply that all organisational functions carried out by classes can be accomplished in a simple and natural way by object inheritance in classless languages, with no need for special mechanisms. An example of such an organisational idiom is traits [UCCH91]. Traits are objects that contain shared behaviour and state (analogous to class methods and variables). Objects that want to share this common state and behaviour use the traits as parent objects. Thus traits objects in a classless language provide the same sharing capability as classes. The immediate impression that class-based programmers get is that traits are just a reinsertion of classes in the classless world, thus justifying the need for classes. The point we make is, traits (or any similar organisational mechanisms) are hidden in classes and in the class-instance implicit relationship. Now, if one moves to the world of distribution and mobility (where sharing of information becomes a key issue), the implicit relationship becomes problematic. A class-based programming supporter could argue that one can make these relationships explicit, e.g. letting an object specify, upon migration, if its class will stay in the origin node and be invoked via RPC protocols, or it will move also to the destination node; or letting a class specify whether its superclass will migrate together with it or not (in the latter case a remote-parent link would be established). But making explicit the relations is actually object-based programming! The object needs to be aware of its class. Doing all these things "manually" is precisely what object-based programming promotes. This is consistent with the thesis that distribution cannot be transparent.

The main conclusion would be that, although sometimes group-wide mechanisms such as classes are useful for reusing, organising and architecting systems, in processor cloud programming some characteristics of classes render them problematic. It is better to leave the programmer the explicit choice of what mechanisms to use. The problems do not reach the roots of the paradigm, though: objects (encapsulated state + message passing) are well suited to the task.

References

[BL92] G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proceedings of IEEE Computer Society International Conference on Computer languages*, 1992.

[GF99] Rachid Guerraoui and Mohamed E. Fayad. Oo distributed programming is *Not* distributed oo programming. *Communications of the ACM*, 42(4):101–104, 1999.

[Sat01] M. Satyanarayanan. Pervasive computing: Vision and challenges, August 2001.

[Sha03] Nigel Shadbolt. Ambient intelligence. *IEEE Intelligent Systems*, 18(4):2–3, 2003.

[SLU89] Lynn Stein, Henry Lieberman, and David Ungar. A shared view of sharing: The treaty of orlando. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 31–48. ACM Press/Addison-Wesley, Reading (MA), USA, 1989.

[UCCH91] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hlzle. Organizing programs without classes. *Lisp and Symbolic Computation*, 4(3):223–242, 1991.

[Weg87] Peter Wegner. Dimensions of object-based language design. In *Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 168–182. ACM Press, 1987.

[Wei91] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(30):94–104, September 1991.