# Wild Abstraction Ideas for Highly Dynamic Software

Wolfgang De Meuter, Jessie Dedecker*, and Theo D'Hondt

Programming Technology Laboratory, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
{ wdmeuter, jededeck, tjdhondt}@vub.ac.be

**Abstract.** In this position statement we will propose a few roughly sketched language features of which we feel that they might dramatically change the way we can structure and reason about systems that have to operate in highly flexible environments. The features proposed have never been incorporated in a real language and have never been implemented. Our goal is to be provocative rather than demonstrative. Nevertheless, we try to stay down to earth in the sense that our proposal is perfectly implementable with current day hardware. No biological stuff. We do not claim this is a scientific document, but we hope it provokes some thoughts and that it may give rise to science in the future.

## 1 The Kitchen with no Buttons

Harry is a single. His friend Gina is visiting him. They plan to prepare diner together for old times sake. Harry has an ultra modern kitchen of the famous ZanuKnecht brand with all amenities: a gammaray oven, cooker, coffee machine, blender, an old microwave oven, fridge, freezer and a cooker hood. Needless to say, all equipement - except for the microwave - is 100 percent pure free of buttons. All operation is done by means of the latest handheld Gizmo model Harry always has at hand. The microwave was adjusted with an Analogue/Gizmo adapter so that its CPU now works with the device as well. These days, such a device is as common as the good old European cellular 20 years ago. Gina also has one, albeit slightly obsolete, but it does what it's supposed to do because it is just good enough to run Utopia2 scripts. Gina's old Baunussi kitchen is completely different. E.g. she has an old fridge with a built-in freezing compartment that cannot be regulated separately and she has one of those old cooker hoods that still operates independently from the cooker. She doesn't even have a blender. But she does have one of those latest cellular ovens that make the two ovens of Harry look like antique scrap. Both run Utopia2 scripts. Harry those of Zanuknecht. Gina the ones released by Baunussi.

They set about. While Harry is chopping vegetables Gina sets about to prepare the broth. Harry sends her (via the Gizmo2Gizmo-standard) his cooker objects (kitchen-click-control-click-share-click-address book-Gina-click) and off

* Research assistant of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.)

she goes. Although her Gizmo still works with old fashioned pixel technology, she is perfectly able to use these objects because of the Gizmo-classic-pixels-converter she installed. It ensures a best-possible projection onto pixel displays of those modern liquid morphs interfaces everyone is running today.

While Harry is busy cutting vegies, Gina asks him from within the kitchen if he is willing to run over to the 7-isnot-11 to get some butter. Normally, Harry always does his shopping at WallSmart around the corner, but Gina insists on the "Le Chat Qui Pleure" brand that is being sold exclusively at 5-isnot-11. So he has to upload the products of WallSmart into his Gizmo. He points the Gizmo to the Gizmo-dock next to the cable-tv-plug and selects www.wallsmart.supermarkets, clicks "dairy produce" and sets of. While walking on the sidewalk, Harry runs through the dairy produce assortment of WallSmart. After a while he finds Le Chat Qui Pleure butter and buys a pack. The i-ticket on the Gizmo screen notifies him that the pack is indeed bought and that the two euros are withdrawn from his bank account. Phone! Gina phones to ask whether he can send her his cooker-hood-objects. The broth is boiling and the kitchen windows are steamed up. Kitchen-click-cooker hood-click-share-click-click-ok. He quickly sends her his apartment-environment-objects in the same way. Otherwise, Gina wouldn't even be able to put on the radio or turn on the light. But before doing so, he excludes the safety-box-objects and the body-objects. After all, what he has seen on the scales this morning is none of her business. House-click-domotics-click-share-click-excludefrom-click-sca... scales-click-saf... safety box-click-click-click. Upon arrival at WallSmart, he beams the i-ticket to the door. The Gizmo displays "counter 5". He walks to the counter, says hello to the friendly lady, beams her the i-ticket and receives his butter. Of course, the i-ticket is now erased from the Gizmo. If not he could be eternally collecting butter with the same ticket. Back home, the broth is simmering and the cooker hood is humming softly. Precisely at that time Gina turns of the cooker hood on her Gizmo. The humming stops as well. Harry reacts by pressing cookerhood-on on his Gizmo since the whole show is still vaporous. Suddenly, Harry wonders what those blinking red letters are doing on the microwave's display. Gina replies that she downloaded the new Utopia2 software because Harry's scripts already missed two new releases. A little agonized by the freedom she permitted herself, Harry thinks that he should have exluded the script-installation-objects from the apartment-environment-objects he just sent her. Nevertheless, Gina beams him the new scripts using Gizmo2Gizmo and he stores them as the default for the microwave. Of course, the setting from the old scripts on his Gizmo are automatically converted, for otherwise he would have to manually synchronise all those cookery book scripts again between his Gizmo and the kitchen devices.

Ok, let's stop it here. Which objects are on which machine? What is shared? What is replicated? How does Utopia2 look like? We honestly have no idea, but we're pretty sure it will not be an easy task to express this complexity in Java.

## 2 Wild Prototype-based Languages

One of the things we certainly think we should get rid of is the static structure of software currently imposed by classes, inheritance hierarchies and types. Indeed, about half of the work required by writing this kind of software using middleware solutions, lies in the specification and generation of interfaces, abstract superclasses, wrappers, stubs and the like. I.e. stuff to make type-checkers happy. Therefore, it is our conjecture that the future will be prototype-based. We believe that the dynamicity of the systems described above requires such techniques as extensive meta programming, dynamically updating objects with unanticipated functionality and runtime structural modification of objects will be inevitable. Objects will probably reside on different machines at the same time such that methods can - transparantly or not - run in contexts they originally were not designed for.

In the context of the scenario we think these prototype-based languages will need built-in support for

- **unusual scoping mechanisms** that allow objects to see 'more' than what is in their lexical scope and/or in their encapsulation boundary. But of course, 'wild dynamic or global scoping' is out of the question. Instead some limited forms of dynamic scoping will be needed. For this we will propose two mechanisms, namely, networked delegation and dynamic scoping of first class methods.
- **special sharing mechanisms** that allow (mobile) (networked) independent objects to 'see and share each others parts'. For this we will propose networked delegation and cloning families.
- **broadcasting mechanisms** that allow messages to be send 'in one stroke' to many (related) objects.
- **funneling and collection mechanisms** that allow a controlled reception of multiple results from messages and broadcasts.

We will now list a number of wild language features and roughly sketch the semantics we have in mind for them. As the title of the paper suggests, our goal is to be speculative and provocative, not demonstrative. The features suggested will probably generate more problems than they offer solutions.

## 3 Dynamic Scoping

After the "design error made by Lisp" it is commonly agreed that "good" programming languages "should" implement lexical scoping. We claim that software that has to operate in highly dynamic contexts might benefit from dynamic scoping. Consider for example a mobile object that accesses a variable `printer` that is in its scope. The following scenario might be possible. The object decides to move to another machine and while receiving the object, the receiving procedure redefines the `printer` variable. In the case of a dynamically scoped variable, the object will lookup the `printer` variable along its runtime stack. Therefore the

redefined version of the receiving procedure will be found and not the one that is in its lexical scope. Hence, the object will use the printer of the system where it resides and not the printer of the system where it was created.

Maybe it sounds "too easy" but we think that dynamic scoping might be exactly the language feature needed to reason about the dynamic context in which mobile agents will have to operate.

## 4 Networked Delegation

One of the things we strongly believe is that objects will no longer reside on one single machine but will be distributed across different nodes in a network. To accomplish this technically, we propose a feature called *distributed delegation*. The idea is to have objects on machines whose parent objects reside on another machine. Messages sent to the child are automatically delegated over the network to the parent. Arguments are marshalled and results unmarshalled. Furthermore, self-sends in the parent will come back - as usual - to the original receiver of the message. Hence, self sends and super sends might involve network traffic. Of course it is by no means our goal to promote code-reuse over a network: it is not our intention to deny the importance of things such as v-tables and to state that "in the future networks will be so fast and reliable that they will be fast enough to support a default method lookup over the network". However, we *do* think that the delegation metaphor can significantly simplify the way we reason about distributed objects. Below are a few arguments to support this claim.

One of the nice things of delegation is that it gives rise to parent-sharing: whenever two objects choose the same object to be their parent, the state of the latter object becomes shared by the two child objects. Messages sent to one child object can give rise to state changes in the parent object, and are those state changes the second child object might 'feel' whenever it receives messages. Again, the reason is that the parent is shared by the two child objects. Here is why we think this might be an interesting language feature to consider:

**Controlled sharing of state.** Everyone knows that the biggest problem of parallel (and also distributed) systems is due to state sharing. Functional programs in which no expression can ever give rise to side effects is trivially executed in parallel: a function that calls n other functions can simply spawn n processes for those functions since their execution can never influence each other. On the other extreme, imperative programs in which concurrently running parts communicate through uncontrolled shared variables are very difficult to manage (Java is a very good example thereof). A reaction might be (as was witnessed by actor languages or CSP-based formalisms) to copy as much as possible from the functional world into the imperative world and to localise state changes by having variables owned by their specific processes. As algorithms expressed in actor languages and in CSP show us, however, this yields programs half of which the text is about the concurrency: guarded conditionals, sending around messages to control concurrency and so on. An algorithm as simple as the good old factorial is as good as unrecognisable in these formalisms. We therefore think

an intermediate form of sharing of state is necessary: some variables have to be accessible by different processes or different concurrently running objects, but only by those objects that are "properly subscribed to those variables". We think the concept of an object that is shared in a controlled way by a number of descendants might be a good alternative.

**Views.** Imagine a university where the registration service has an object for every student. Then the sports department can have an object that represents the student 'as a sportsman': the object contains the information about the student's sport skills, whether he paid his sports card, and so on. This information is of no relevance to the registration service, so it resides on machines in the sports departement. In the same way, the university medical center has an object that represents the student's medical record. Again, the medical information of the student is not relevant to the registration service and therefore resides in the medical databases. But of course, we want the 'identity information' of the student as a person not to be duplicated. The solution would be very simple in a prototype-based language with networked inheritance: all the services have their own objects; the object at the registration service is the shared parent and the other objects denote this object to be their parent. This way the information that is conceptually shared is also technically shared. Messages sent to the student as a sportsman that are 'sports specific' will be handled by the part of the object that resides on the sports department machines. The others will be delegated to the registration served, *and*, self-sends will come back to the part of the object that resides in the sports departement. This way, the distributed delegation mechanism allows us to have objects centralized on one machine while views on those objects that have nothing to do with the centralized part can reside on other machines. The delegation happens automatically and no extra "message forwarding methods" have to be written.

**Transparant Network References.** One of the issues in a distributed system is the ever lasting distinction between "real objects" and "network references". The latter are "empty" objects that serve as proxies for the real objects, on the other side of the network they represent. They implement the same interface as the objects they represent. Nevertheless, they only implement empty methods that do nothing but forward the message (after marshalling the arguments) over the network and await the result under the form of a real object or under the form of a network reference.

The fact that network references are "empty objects that merely delegate the messages over the network" pleads for the delegation mechanism we are promoting here. A network object is nothing but an empty object that denotes the object it represents as its parent object. Messages sent to the descendent - i.e. the network reference - are automatically delegated over the network - i.e. to the object represented by the network reference.

**Client Server.** The client-server metaphor for organising systems (also called master-slave) becomes very trivial in our proposal: the server is the parent and the child objects are the clients. Messages sent to the client which are not answered by the client, can be automatically delegated to the server. Super-sends

in the client are ways the clients (i.e. the child objects) can talk to the server and self-sends in the server go back to the right client because that client is precisely the meaning of the `self` pseudo-variable.

**Discussion**

The mechanism described in the previous section gives rise to a lot of technical issues to be resolved. We mention only two of them to counter the "easy" criticisms on our proposal:

**Copy-Down Methods.** As explained before, it is not our goal to use delegation over a network for code reuse purposes. We might encounter situations in which two view objects have the same method. Common object-oriented refactoring practice requires us to distill such a method into the parent object. But when this parent is on a different machine this might give rise to unacceptably slow method lookups. A solution might be to introduce a form of `copyDown` keyword: child objects specify the signature of a method in a parent (i.e. they override the method) but as the body of the method they mention `copyDown` which means that the code from the parent should be copied to the child. The result is that the program text will mention the code only once in the parent, but in reality it will be copied to the child 'client' objects.

**Private Self Sends.** If we would employ the standard delegation semantics in every possible object at every possible network node, then as good as every simple recursion occurrence (i.e. every self-send) would give rise to network traffic. Therefore a mechanism such as private methods in Java or C++ might be necessary. Other solutions might be the introduction of another pseudo variable `me` that is not late bound.

**Concurrency Issues.** Of course the standard concurrency issues immediately turn up in our proposal: when concurrent messages are being sent to two child objects that both simultaneously arrive at the parent object, this will have to be dealt with properly. To solve this, we will have to look at the work already done in combining objects and parallel programming. Currently, we are looking to the (forgotten) actor technology for this.

### 4.1 Us

We can think of the child objects as views on the parent, in the same way graphic UI's in object oriented systems are seen as views on a model. A code-pattern that often re-occurs consists of an iteration construction that sends the same message to all the views. We therefore propose the introduction of an `us` pseudo variable. When an object sends a message to that pseudo variable, that message is send to all the descendants of that object. Of course, these 'multiple self sends' might give rise to the situation where the parent receives its own message back a number of times. Indeed, suppose we have a parent object (that implements `m`) with 5 children, 2 of which override `m`. When the parent sends a `us.m()`, the message will be sent to the 5 children. 2 of these will be handled, but the other 3 will (due to method lookup) arrive back in the parent. We therefore propose a

kind of funneling mechanism. Messages distributed to `us` stay connected in such a way such that several messages that meet each other in a shared parent are being merged to one message.

Suppose that a parent object represents a model and that the views on the model are implemented as 'views' (i.e. child objects) on that parent. As we know from the MVC technique, every state change in the parent normally gives rise to a `changed` message being sent to all the views. Using the `us` concept this is simply encoded as `us.changed()`.

## 5 Multivalues

The notion of `us` can be generalized into what we will call *multivalues*. Multivalues are values that *are* several objects. Sending a message to a multivalue (that might be the value of a variable `v`) will concurrently send the message to every object in the multivalue. But the idea is that, when, some of these objects have a parent shared, and when these objects do not implement the message, that the messages merge in order to become only one message in the shared parent.

Two operators `+` and `/` are defined on multivalues. If `v` is a variable holding a multivalue, then the value of the `v+o` is a multivalue that is the same values as the value of `v` augmented with `o`. Likewise, `/` removes an object from a multivalue.

Multivalues allow one to write extremely simple message expressions that have a large action radius. For example, the expression `v.m().n()` will send `m` to every object in the multivalue `v`. The result can be a multivalue which is sent `n` in its turn. Again, this proposal raises more questions than it solves. E.g., in the case of a cascaded message like `v.m().n()` one has to answer the question whether `n` will wait for every value in `v` has returned the result of sending `m`, or, whether the sending of `n` can "already begin" once the first result of sending `m` start "to come in". Honestly, at the time of writing, we have no idea.

Notice that it is not the idea to think of multivalues as sets of values that can be processed sequentially. However, if we *do* want to return to the realm of sequential processing after having several multivalue expressions, we will need type converting operators to convert the multivalue to a set or an aray much in the same way Pascal features functions such as 'round' and 'trunc'.

## 6 Multimessages

Another feature we would like to bring to the fore is what we call *computed messages*. The idea of a computed message is as follows. Imagine an expression like `o.m(o')`. In standard OO-practice, both the receiver `o` and the argument(s) `o'` are evaluated while `m` is "fixed in the program text". The idea of a computed message is to allow `m` to be an arbitrary expression that is supposed to evaluate to a message symbol. This would allow expressions like `o.if(x=5,'message1,'message2)(o')` in which the concrete message being sent is computed at runtime. Combined with the idea of multivariables, the

outcome of this computation might be a multivariable such that many messages might be sent in parallel (of course, with no order imposed).

The idea of a computed message enables meta programming techniques as we know them in Smalltalk (using `perform:`) without having to go through the hassle of manipulating (meta) classes and/or meta objects. Furthermore, an interpreter can still optimise the message expressions in which the message itself is a symbol, such that we don't have to pay for the feature in case it is not used.

## 7 Cloning Families

Another sharing mechanism that might be useful arises from the notion of so called cloning families in the prototype-based language Kevo. The idea is that when objects are cloned, the clones form a cloning family together. In Kevo this is just an optimization technique, but we believe it can be used to structure programs in a better way. Indeed, we could think of some kind of instance variables for objects that are never cloned (like statics in Java). All the objects in the cloning family share those static variables and the effect of assignments done by one member of the cloning family are visible inside the other members of the cloning family. This is a form of sharing between different prototypes.

This language feature of static variables along cloning families could be a conceptual way to think about replication. On the conceptual level, the static variable is shared between all the objects in the cloning family. On the implementation level this merely consists of replicating state changes between the members of the family.

## 8 Conclusions

Of course, there is not really a lot to conclude. We did not implement these "features" and, at the time of writing, we have only a very sketchy semantics for them in mind. Nevertheless, we think that they are worth spending some research time on, which is what we will do in the near future. Again, our goal was to be provocative instead of demonstrative. We want to use the workshop as a 'test audience" for the ideas.