Object Graph Isolation with Proxies

Camille Teruel, Damien Cassou, Stéphane Ducasse

Object Graph Isolation with Proxies

Camille Teruel
Inria Lille Nord-Europe
camille.teruel@inria.fr

Damien Cassou
University of Lille 1, Inria
damien.cassou@inria.fr

Stéphane Ducasse
Inria Lille Nord-Europe
stephane.ducasse@inria.fr

ABSTRACT

More and more software systems are now made of multiple collaborating third-party components. Enabling fine-grained control over the communication between components becomes a major requirement. While software isolation has been studied for a long time in operating systems (OS), most programming languages lack support for isolation.

In this context we explore the notion of proxy. A proxy is a surrogate for another object that controls access to this object. We are particularly interested in generic proxy implementations based on language-level reflection. We present an analysis that shows how these reflective proxies can propagate a security policy thanks to the *transitive wrapping* mechanism. We present a prototype implementation that support transitive wrapping and allows a fine-grained control over an isolated object graph.

1. INTRODUCTION

Modern software systems tend to aggregate independently developed third-party components. It is the case in software that can be extended with plugins or in web pages known as mashups that aggregate functionalities from multiple sources. These components may not trust each other. In such kind of situations, a component may want to be isolated from the others.

Software isolation has been studied for a long time in operating systems (OS) where the classical solution is process isolation [2,8]. A process is isolated from other processes and has its own permissions.

Surprisingly, most programming languages lack support for isolation mechanisms. In this paper we study the notion of *proxy*. A proxy is a surrogate for another object that controls access to this object. Proxies have a wide range of use cases: from basic access control to complex dynamic analysis [14]. In this paper, we are particularly interested in generic proxy implementations based on language-level reflection. We present an analysis that shows how these reflective proxies can propagate a security policy to isolate an object graph.

The paper is organized as follows: Section 2 presents the general notion of proxies and the need for generic proxies. It also presents generic proxies implementation based on the language reflective facilities which set the context for the rest of the paper. Section 3 presents *transitive wrapping* [12, 16], a propagation mechanism based on reflective proxies. Section 4 presents conclusion and future works.

2. PROXIES

A proxy controls accesses to another object called its *target* [6]. There are many use cases for proxies: A proxy can for example check some access-control policy on the target, can forward requests to a remote object, and can log and analyze communications. Two main categories of use cases can be distinguished [16]:

- **Wrappers**: A wrapper is a proxy that wraps around a target object. The target is an object in the sense that the wrapper's runtime must recognize it as an object. This usually implies that the wrapper and the proxy live in the same address space. Wrappers are used for logging, profiling, checking contracts [15] or access control, etc.
- **Virtual objects**: A virtual object is a proxy that emulates an object behavior for an arbitrary target. The target does not need to be recognized as an object by the proxy's runtime because the proxy's job is precisely to fake its target as if it were an ordinary object. In other words, it doesn't matter what the target really is since the proxy tells the runtime how it can be handled as an object. For example, the target can be an object in another address space (*e.g.,* where the proxy implements transparent remote communication), it can be in a serialized form on a disk (*e.g.,* for application-level virtual memory [14]), or stored in a database (*e.g.,* for transparent automatic persistence).

When using a proxy for access control or other security-related concerns, this proxy is likely to be a wrapper. In such a situation, an important requirement is the absence of target leaking: it should not be possible to obtain access to the target from the proxy itself. Otherwise, code can just use the target it obtains to bypass the security policy that the proxy is supposed to enforce.

2.1 Generic Proxies

A naive way to implement proxies consists in creating a class with the same interface as the class whose instances need to be proxyfied. However, this kind of implementation has drawbacks. Firstly, the newly created proxy class works only for the intended target class; if one wants to wrap objects of other classes the logic has to be duplicated. Secondly, this implementation often involves code duplication for each altered method. Finally, the mechanisms implemented by these proxies may be difficult to reuse and compose. One needs a way to write a proxy in a generic manner. To overcome this problem, multiple solutions can be designed based

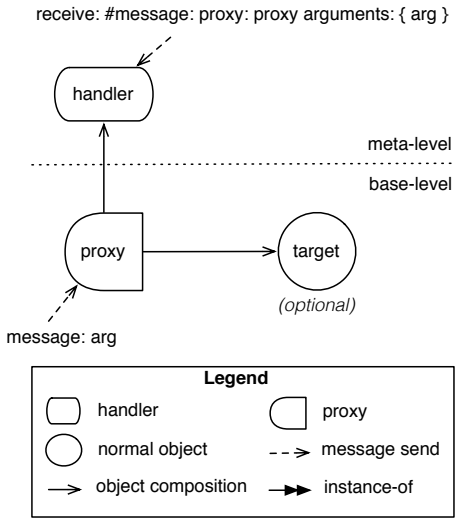

Figure 1: A message is sent to a proxy, triggering a hook in its handler.

on aspects [10] or generative programming [4] for example. Some programming languages such as *Smalltalk* [11], *Java* [5], and the future *ECMAScript 6* [17] added support for some sort of generic proxies. These implementations rely on the reflective abilities of their respective programming languages.

In object-oriented languages, reflection is often provided by an extensible interpreter. Such an interpreter manipulates *metaobjects* that represent some constructions of the language (such as objects, classes and methods). A metaobject can define hooks that are triggered by the interpreter when specific events occur. The set of hooks an extensible interpreter offers is called its *metaobject protocol* (MOP) [9].

When reflection is used to implement the proxy mechanism, the behavior of each proxy is often described by another object called its *handler*. A handler has the possibility to define some hooks that will be triggered when specific events on the proxy happens. These hooks are referred as *traps* [16].

A proxy can thus be regarded as a base-level object, its handler as its metaobject and the set of available traps as a metaobject protocol. Using a separate object as handler permits to ensure that the base-level and the meta-level are not conflated. This principle is called *stratification* [3].

The most common trap is the one that is triggered when a proxy receives a message. This trap is the only one available for *Java*'s *dynamic proxies* for example. Beyond this basic trap, some languages such as *EcmaScript* provide state-access and state-assignment traps among others.

Figure 1 shows a message sent to a proxy; this message is intercepted and the trap corresponding to a received message (here receive:proxy:argument:) is triggered in the handler. Similarly, other traps may be triggered depending on what the MOP has to offer.

In the context of reflective proxies with stratification, a proxy must encapsulate its handler in addition to its target: one should not be able to retrieve the handler from the proxy. Indeed, if a piece of code can retrieve a proxy's handler, the policy can be altered. If the MOP offers object-state access traps, the default implementation for proxies is thus to forward the request to the target if there is one or to raise an error if there is none. In no case the proxy should return its actual state *i.e.,* its handler and its target.

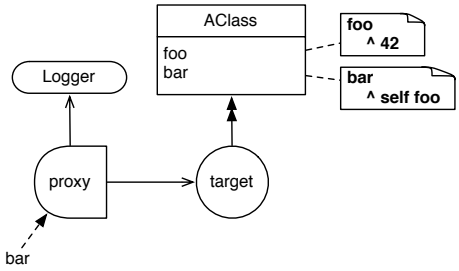

Figure 2: A wrapper proxy and its target

2.2 Discussion: Self-rebinding

This subsection briefly discusses the propagation of interception in the context of self-sends. Consider the situation described in Figure 2. In this situation, a generic wrapper proxy wraps a target whose class defines two methods foo and bar where bar performs a self send to foo. We want the proxy to log each message send so we attach it a Logger handler. When the proxy receives the message bar, two different outputs may be generated depending on the handler's policy: either simple forwarding or self-rebinding.

2.2.1 Simple forwarding

When the handler's policy is a simple forward, the handler first logs the received message and then forwards it to the target. The target then executes its bar method that calls its foo method and finally returns 42. In the end the output is the following:

```
Message bar has been sent.
```

2.2.2 Self-rebinding

When the handler's policy is self-rebinding, the execution is slightly different. The execution begins the same, the proxy receives the message bar that is trapped and the handler logs the message. Then, instead of simply forwarding the message to the target, the handler searches for the method the target would have executed and then executes it but with the proxy as the receiver. Consequently in this execution, the self reference in the body of the foo method refers to the proxy instead of the target. As a result, the handler hooks are triggered for the whole execution of the message sent, not just for the initial message. So the proxy executes the bar method of the target and sends foo to itself. The handler's hook for message reception is thus triggered a second time. In the end, the output is the following:

```
Message bar has been sent.
Message foo has been sent.
```

We give the following definition:

DEFINITION 1. *A proxy is said to support* self-rebinding *if instead of forwarding messages to the target, its handler executes the body of the method the target would have executed with the* self *reference bound to the proxy.*

Note that self-rebinding makes sense only in the context of wrapper proxies. It is impossible for a virtual-object to support self-rebinding since there is no target method to lookup.

Java's *dynamic proxies* don't enable self-rebinding whereas *ECMAScript 6* proxies enable self-rebinding by default.

In our opinion, a proxy framework should support both simple forwarding and self-rebinding to let the user specify its needs with fine control.

2.2.3 Requirements

There are two requirements to allow self-rebinding. First, the runtime must provide a way to execute a method upon an arbitrary receiver. For example Pharo[1] has such a feature through the CompiledMethod's receiver:withArguments:executeMethod: primitive method. Secondly, the metaobject protocol must provide hooks for accessing an object's state so that a proxy's handler can intercept such accesses to manage them (usually by forwarding state access requests to the target).

2.3 Prototype Implementation

Here we describe our current prototype implementation of proxies done in *Pharo*, a *Smalltalk* environment. This implementation is based on the early design of *Ghost* [11]. Right now, this implementation only supports the trap for message reception. Modifying the compiler to instrument the bytecode allows us to send specific messages on some events like instance variables or global variable accesses. New traps are then available at the cost of additional message sends and conflation of base and meta-level.

We implemented the message reception trap by forcing an exceptional situation where the virtual machine (VM) cannot send a normal message to an object. In this situation, the VM sends a message that is normally implemented as an error signal. The trick is to redefine this message and to delegate message processing to a handler. This exceptional situation is the absence of a method dictionary in a class. When the VM performs the message lookup algorithm and finds a class that has no method dictionary, the VM sends a message cannotInterpret: with the reified message as an argument to the receiver. With this message send, the lookup starts in the superclass of the class that has no method dictionary. Otherwise, there would be an infinite loop.

Our implementation consists of a class Proxy. This class' metaclass has an instance variable that is an anonymous subclass of Proxy. This anonymous subclass has no method dictionary. The class Proxy overrides the cannotInterpret: method to perform the delegation to the handler. A new proxy is created as follows:

```
proxy := Proxy handler: someHandler target: someObject.
```

Here is the implementation of the handler:target: class-side method:

```
handler: aHandler target: anObject
  | newProxy |
  newProxy := self new
    handler: aHandler;
    target: anObject;
    yourself.
  self anonymousSubclass adoptInstance: newProxy.
  ^ newProxy
```

First, a new instance of proxy is instantiated and its handler and target are set. Then the anonymous subclass "adopts" the new proxy: that means that the class of newProxy is changed from Proxy to the anonymous subclass (adoptInstance: is a primitive method provided by Pharo). Likewise, when the VM performs the message lookup it doesn't find the method dictionary of the anonymous subclass. Consequently, the VM sends cannotInterpret: to the proxy but starts the lookup in the superclass of the anonymous subclass: Proxy. Proxy redefines cannotInterpret: as follows:

```
cannotInterpret: aMessage
  ^ handler respondTo:
    (Interception message: aMessage target: target proxy: self)
```

[1]http://www.pharo-project.org/home

This method creates an *interception* that is used to store the message, the proxy and the target. This method then asks the handler to perform its policy for the interception.

We provide a default handler class that can be used in most situations. An instance of this class can define its policy in term of different actions:

- **deny:** and **denyAll:** are used to forbid one or several selectors to be sent.
- **forward:** and **forwardAll:** are used to forward one or several selectors to the target.
- **selfRebind:** and **selfRebindAll:** are used to perform a self-rebinding for one or several selectors.
- **on:do:** and **onAll:do:** are used to specify some custom action for one or several selectors. The second argument is a Smalltalk *block*[2] that takes an interception as argument.

Finally a default action is defined for selectors that have no specific action.

This set of methods permits to customize the access policy with a fine-grained control. One still can define its own handler for more complex message processing.

This prototype has the advantage to not rely on virtual machine modifications, but it also has drawbacks. First, this implementation has only one primitive trap for message sending interception. Other traps can be implemented in term of the primitive one by instrumenting bytecode. Then this implementation also conflates the base-level and the meta-level. A better implementation would be to rely on a mirror-based architecture [3] to avoid base-level and meta-level conflation.

3. OBJECT GRAPH ISOLATION

In this section we explain how proxies can be used to isolate a whole object graph from the rest of the objects, thanks to the technique of transitive wrapping. This technique can be used to execute an untrusted piece of code in an environment confined with a custom policy for example. Transitive wrapping is a powerful technique that allows one to isolate a graph of objects behind a layer of proxies created lazily. Transitive wrapping's goal is to wrap all objects crossing the boundary of the graph in both directions. A *membrane* [12, 16] uses transitive wrapping to ensure that every object obtained from the isolated object graph is revocable [3]. Usually, a membrane begins its life by wrapping a single object that is the root of the graph one wants to isolate. When a message is sent to this wrapper proxy, the arguments are wrapped as well as the return value of the message. Likewise, all objects obtained from the root wrapper are wrapped and all objects passed as argument to an object of the graph are wrapped too. The membrane can then be revoked and all the references obtained from the membrane or passed to it are revoked too. If a membrane offers a revocable graph, the underlying transitive wrapping mechanism can be applied to other policies.

Figure 3 shows a scenario where a graph of two objects is isolated (obj2 and obj3). In the beginning there is only one object wrapped (obj2), that represents the entry point of the isolated graph. An object (client) sends a message (message:) to the root wrapper (prx2) with another object (obj1) as an argument (step 1). Then, the wrapper wraps the arguments with a new proxy (prx1) and forwards the message with arguments wrapped (step 2). When this message is received, the target responds another object (obj3, step 3) that is wrapped with a new proxy (prx3, step 4).

[2]Smalltalk blocks are similar to closures.

[3]A revocable object reference allows one to share an object with some other objects and then revoke this access.

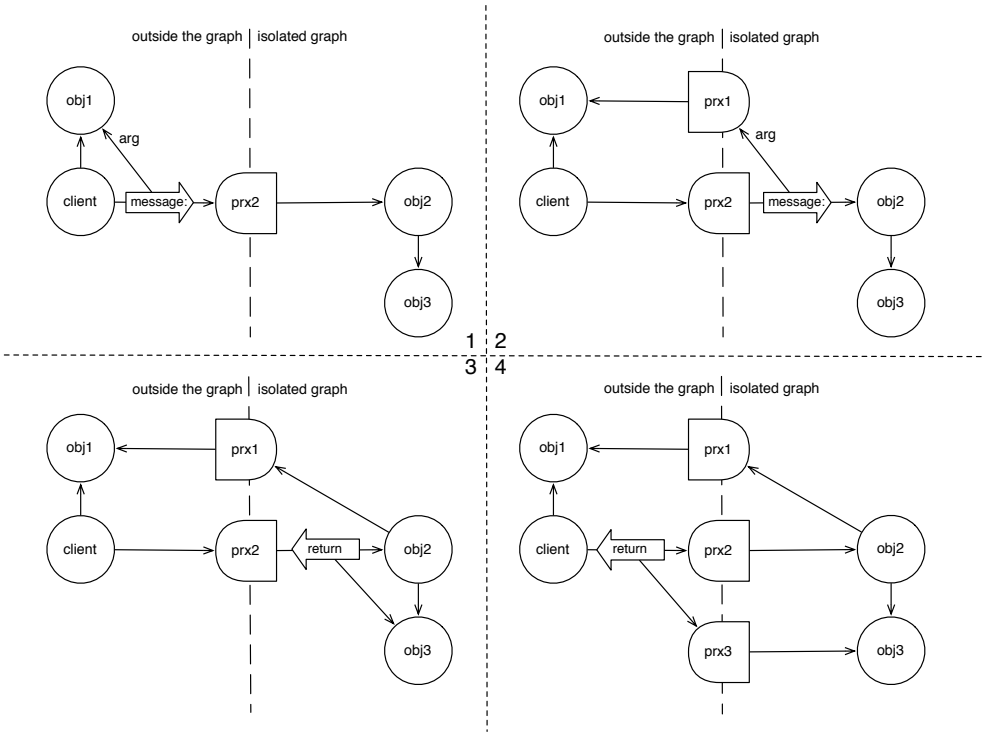

Figure 3: Scenario describing transitive wrapping. Handlers are not shown for the sake of simplicity. In step 1, the object client sends a message to a proxy proxy with obj1 as an argument. In step 2, the handler of proxy wraps the argument obj1 with a new proxy prx1. In step 3, the object target returns obj2. Finally, in step 4, the handler of proxy wraps the return with a new proxy prx2.

3.1 Discussion

Separating inside-out and outside-in policies.

In the context of isolation, it can be useful to differentiate between proxies that wrap objects from graph (outside-in proxies) from the ones that wrap objects that are not in the graph (inside-out). Outside-in proxies (such as prx2 and prx3 in Figure 3) are supposed to transform message arguments into inside-out proxies (such as prx1 in Figure 3) whereas inside-out proxies are supposed to transform message arguments into outside-in proxies *and* to ensure some policy. In such a configuration, outside-in proxies are just used as a facility to mark their target as *untrusted* to automatically wrap arguments of message send to it to protect them. Symmetrically, inside-out proxies ensure some policy and wrap arguments of messages to mark them untrusted. This means that inside-out and outside-in proxies have different policies. Typically the policy of outside-in proxies is almost empty.

Auto-unwrapping.

When a proxy receives a message, it is supposed to wrap the arguments. But what happens if one of the arguments is already a proxy for an object of the other side (*e.g.,* if client sends a message to prx2 with prx3 as an argument)? The first solution is to ignore this fact and to still wrap this proxy. Wrapping wrapper this way is both inefficient and useless. The second solution is to unwrap objects that come back to the side they belong to. Likewise an object is at most wrapped once when it is referenced from the other side but is always unwrapped when it is referenced from its side.

Identity preservation.

In a naive implementation, a proxy is created each time an object crosses the boundaries, *i.e.,* if an object crosses twice, two different proxies are created. This approach is inefficient because a proxy that already exists can be reused. A more important issue is that object identity is not preserved across the boundaries, potentially breaking code that rely on object identity. To ensure that an object has at most one proxy, a solution is to use a dictionary. Precautions need to be taken to ensure that garbage collection behaves correctly. The usage is to use a data structure called a weak map. A weak map is like a dictionary that removes its associations whose key could be garbage collected. Thanks to this data structure, it is possible to have reference cycles in the associations and still permit garbage collection. Weak maps are similar to *ephemeron tables* [7] except that objects are not warned before getting garbage collected.

3.2 Implementation

In our implementation, when one wants to wrap an object graph, two wrappers are created: one is the handler of outside-in proxies and the other one is the handler of inside-out proxies (as shown in Figure 4). These wrappers thus always come by pair. Each wrapper has a reference to its *co-wrapper*, a weak map to preserve identity and a separate policy handler to encapsulate the policy. We created these co-wrapper pairs to avoid two symmetric wrapping logics and code duplication. We add a separate policy handler to make a distinction between the wrapper that is a handler executing the propagation logic and the separate handle that implements some policy. Likewise, handlers that are not aware of the propagation mechanism can be used in graph isolation. This approach favors composability and reusability of policies.

When a wrapper is asked to wrap an object it executes the fol-

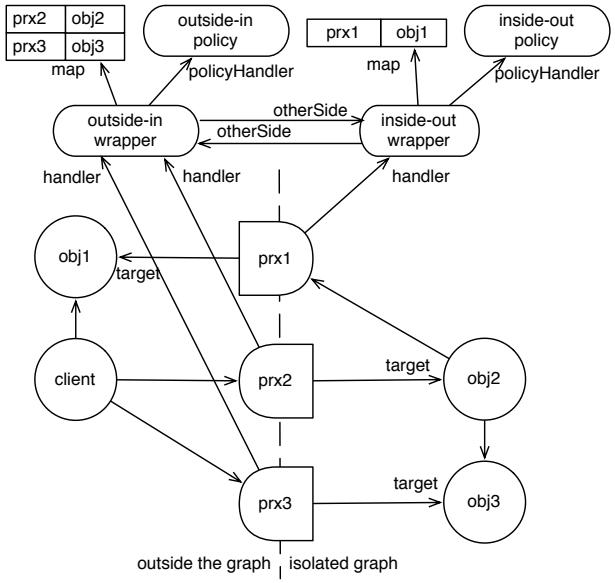

Figure 4: Shows a transitive wrapper pair with their respective map and policy handler in the context of the scenario of Figure 3

lowing code:

```
wrap: anObject
  (self map includesKey: anObject)
    ifTrue: [ ^ anObject ].
  ^ self map
    keyAtIdentityValue: anObject
    ifAbsentPut: [ Proxy handler: self target: anObject ]
```

If the object is already a proxy in the wrapper's map, the wrapper just returns this proxy. This ensure that an object is not wrapped twice. Then the wrapper looks if its map already contains a proxy wrapping the object. Otherwise, the wrapper creates a new proxy and adds a new entry in the map. In both cases a proxy on the object is returned.

The following code shows the message interception logic:

```
respondTo: anInterception
  | response |
  anInterception arguments
    replace: [ :arg | self map at: arg ifAbsent: [ self otherSide wrap: arg ] ].
  response := self policyHandler respondTo: anInterception.
  ^ self otherSide map at: response ifAbsent: [ self wrap: response ]
```

When a proxy receives a message it asks its co-wrapper to wrap all the arguments. The wrapper then delegates the message processing to its associated policy handler. The response can either be a proxy or an object of the other side. In the case of a proxy the wrapper looks for the associated object into its co-wrapper map to unwrap the proxy. If it is an object, the wrapper just wraps it.

Example.

The following code shows our prototype implementation at work.

```
"Create the wrapper"
outInWrapper := TransitiveWrapper new.

"Set prefixes that are used in object printing"
outInWrapper policyHandler prefix: 'suspicious'.
outInWrapper otherSide policyHandler prefix: 'protected'.

"Create an association and wrap it"
association := Association key: Person new value: Circle new.
wrapped := outInWrapper wrap: association.
"Prints: suspicious<a Person->a Circle>"

wrapped key.
"Prints: suspicious<a Person>"

wrapped key: Object new.
"Prints: suspicious<protected<an Object>->a Circle>"

wrapped key.
"Prints: an Object"
```

First a transitive wrapper is created. The inside-out policy modifies the print of wrapped objects by prefixing the normal string with suspicious. Similarly, the outside-in policy prefixes the print string of objects with protected. Then an association[4] of two objects is wrapped. If one prints the wrapped object, one observes the suspicious prefix; it means that the inside-out policy is taken into account. Then if wrapped is asked for its key a wrapped version of the original key is returned with an inside-out policy. If one sets the key of wrapped to be a new object, this object is wrapped with an outside-in policy, as the presence of the protected prefix demonstrates. Finally, if wrapped is asked for its newly setted key, this latter is automatically unwrapped as the absence of prefix shows.

4. CONCLUSION AND FUTURE WORK

We presented the concept of proxy, that is an object that represents another object called its *target* to change its behavior. We saw the need to express proxy policies in a generic way. A common solution is to rely on the reflective facilities of programming languages to implement generic proxies. The behavior of such a reflective proxy is specified in a *handler*. We presented a prototype implementation of reflective proxies in *Pharo*.

We showed how transitive wrapping can be used to isolate an object graph with two separate policies: one managing inside-out communications and the other one managing outside-in communications.

In the future we want to add new traps such as object state reads and writes and class environment accesses. This future solution needs to be stratified to ensure a clean separation of the base and meta-level. Such an infrastructure will be the testbed for other isolation mechanisms like virtual copies [13], worlds [18] and handles [1].

Acknowledgements

This work was supported by DGA, Inria Lille - Nord Europe, Ministry of Higher Education and Research, Nord-Pas-de-Calais Regional Council, FEDER through the 'Contrat de Projets État Région (CPER) 2007-2013'.

5. REFERENCES

[1] J.-B. Arnaud, M. Denker, S. Ducasse, D. Pollet, A. Bergel, and M. Suen. Read-only execution for dynamic languages. In *Proceedings of the 48th International Conference Objects, Models, Components, Patterns (TOOLS'10)*, Malaga, Spain, June 2010.

[4] An association is just an object referencing two other objects as its *key* and its *value*

[2] G. Back, W. Hsieh, and J. Lepreau. Processes in kaffeos: Isolation, resource management and sharing in java. In *4th USENIX International Symposium on Operating System Design and Implementation (OSDI)*, 2000.

[3] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices*, pages 331–344, New York, NY, USA, 2004. ACM Press.

[4] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[5] P. Eugster. Uniform proxies for Java. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 139–152, New York, NY, USA, 2006. ACM.

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

[7] B. Hayes. Ephemerons: A new finalization mechanism. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, 1997.

[8] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, et al. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, oct 2005.

[9] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[11] M. Martinez Peck, N. Bouraqadi, M. Denker, S. Ducasse, and L. Fabresse. Efficient proxies in Smalltalk. In *Proceedings of ESUG International Workshop on Smalltalk Technologies (IWST'11)*, Edinburgh, Scotland, 2011.

[12] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[13] S. Mittal, D. G. Bobrow, and K. M. Kahn. Virtual copies — at the boundary between classes and instances. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 159–166, Nov. 1986.

[14] M. M. Peck, N. Bouraqadi, M. Denker, S. Ducasse, and L. Fabresse. Marea: An efficient application-level object graph swapper. *Journal of Object Technology*, 12(1):2:1–30, jan 2013.

[15] T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, and M. Flatt. Chaperones and impersonators: Run-time support for contracts on higher-order, stateful values. Technical report, NU-CCIS-12-01, 2012.

[16] T. Van Cutsem and M. S. Miller. Proxies: design principles for robust object-oriented intercession APIs. In *Dynamic Language Symposium*, volume 45, pages 59–72. ACM, oct 2010.

[17] T. Van Cutsem and M. S. MILLER. On the design of the ecmascript reflection api. Technical report, Technical Report VUB-SOFT-TR-12-03, Vrije Universiteit Brussel, 2012.

[18] A. Warth, Y. Ohshima, T. Kaehler, and A. Kay. Worlds: Controlling the scope of side effects. In *ECOOP 2011*. LNCS, 2011.