Ownership, Filters and Crossing Handlers

Flexible Ownership in Dynamic Languages

Erwann Wernli Pascal Maerki Oscar Nierstrasz
Software Composition Group, University of Bern
scg.unibe.ch

Abstract

Sharing mutable objects can result in broken invariants, exposure of internal details, and other subtle bugs. To prevent such issues, it is important to control accessibility and aliasing of objects. Dynamic Ownership is an effective way to do so, but its owner-as-dominator discipline is too restrictive: objects are either accessible or not. We propose in this paper to control accessibility and aliasing with more flexibility using two mechanisms, *filters* and *crossing handlers*. We demonstrate the benefits of the flexibility offered by these mechanisms, and report on the adaptation of a Smalltalk web server with our approach. We conclude that our variant of dynamic ownership is flexible enough to accommodate an existing design, while at the same time constraining it enough to highlight design anomalies.

Categories and Subject Descriptors D.3.3 [*Software*]: Programming Languages — Constructs and Features

Keywords Ownership, Encapsulation

1. Introduction

Sharing objects is the essence of object-oriented programming, but sharing makes it hard to protect the integrity of the system: internal information of objects might be revealed, invariants could be broken, or thread safety could be compromised. To prevent such issues, it is important to control the accessibility of object members, and aliasing of objects themselves.

For instance, a web server might serve multiple web sites composed of web pages (see Figure 1). Multiple web servers might run within the same virtual machine on different ports, and must be isolated. Aliases to web pages must be forbidden across web sites and web servers.

Dynamic Ownership [17] structures objects in the heap in an ownership hierarchy, similarly to ownership types [6, 12, 28], and dynamically confines objects within their owner. This policy, known in the literature as *owner-as-dominator* [12], is an effective way to control accessibility and aliasing but is however too restrictive: objects are either accessible or not. To regain some flexibility, we propose in this paper to structure objects in an ownership hierarchy and to control accessibility and aliasing with two mechanisms, *filters* and *crossing handlers*.

Each object defines an implicit ownership boundary. Each ownership boundary has *in* and *out filters* that control the interface that objects within the boundary expose to objects outside the boundary, and inversely. References can cross one or more ownership boundaries and filters are cumulative. An object might see all, or only a limited subset of the methods of another object depending on their relative position in the tree. Ownership can be transferred at any time.

Reference transfer and ownership transfer are however subject to the restriction that all references crossing boundaries inward must expose at least one method. Without this restriction, objects could be aliased in an unrestricted manner; with this restriction, objects that expose no methods outside a given boundary are confined within the boundary. This restriction enables a flexible control of confinement using filters.

When a reference transfer that would violate this restriction occurs, the system reifies the reference transfer and triggers the corresponding *crossing handler*. By default, the crossing handler raises an exception and prevents such aliasing. Crossing handlers are reflective hooks that can be modified to take some action to return a reference that exposes at least one method.

Filters and crossing handlers default to the classic owner-as-dominator policy. We demonstrate in this paper how both mechanisms can be configured to relax the restrictive owner-as-dominator policy. Filters in particular can be used to expose parts of an aggregate, or expose a limited view of the objects within an owner. The latter case allows read-only objects to be exposed. Crossing handlers facilitate the systematic implementation of defensive copying, which complements well the owner-as-dominator policy. We also illustrate other techniques our approach enables.

We have implemented our variant of dynamic ownership in Smalltalk and adapted an available, open-source web server to use dynamic ownership. We found filters and crossing handlers easy to apply and believe they favor object-orientation, without imposing too strong encapsulation constraints.

In contrast to static ownership type systems, our approach is more flexible since it exploits the benefits of dynamic execution for ownership transfer and crossing handler. Also, unlike certain approaches for dynamic languages [4, 32] our approach does not bind policies to references and instead defines the policy to be enforced based on the position of the caller and callee in the ownership graph, which we believe is more natural.

The paper is organized as follows: section 2 presents filters and crossing handlers and their default behavior; section 3 and section 4 show examples of filters and crossing handlers; section 5 discusses the relationship between secure programming and ownership; section 6 and section 7 define the semantics and implementation of our variant of dynamic ownership; section 8 describes the adaptation of the web server and section 9 opens further perspectives. We discuss related work in section 10 before we conclude in section 11.

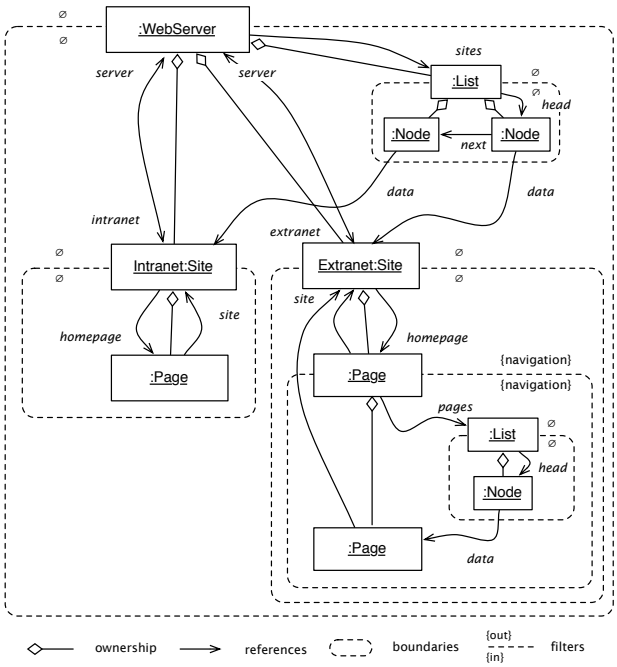


Figure 1. Example of an ownership topology.

```
Page>>inFilters ^ #(navigation)
Page>>outFilters ^ #(navigation)
```

Figure 2. Definition of the in and out filters for the `Page` class

```
Page>>fullUrl
  <topic:navigation>
   ^ site baseUrl , '/' , self relativeUrl.

Site>>baseUrl
  <topic:navigation>
  ^ server hostUrl , '/' , self name.

Site>>siteMap
  <topic:navigation>
  ^ homepage flattenChildren
            collect: [ :page | page fullUrl ].

Server>>hostUrl
   <topic:navigation>
   ^ hostname , ':' , port.
```

Figure 3. Methods belonging to the `navigation` topic

2. Filters and Crossing Handlers

Let us consider the web server in Figure 1. The web server contains web sites that are composed of web pages. The web server has a `hostname` and `port`. The web server uses a list to maintain references to its two web sites, called "Intranet" and "Extranet". Each web site has a name and a home web page. Web pages are instances of `Page`. Web pages can have subpages; a web page keeps references to its subpages in a list. Web pages have URLs that are of the form `http://host:port/site/relative/path`.

Information hiding is a well established principle and it is important to control the accessibility of object members to not break system invariants. For instance, to move a page between web sites, it is not enough to remove it from one list and add it to another list; in addition, the page and subpages must be updated to reference the right web site. The list of subpages should be manipulated only via methods exposed by the corresponding page.

Ownership Objects are organized at run-time in an ownership tree. In Figure 1, the web server owns the web sites; the web sites own the web pages; lists own their corresponding nodes. The nodes do *not* own the objects they reference, though. Each object defines an implicit ownership boundary that contains all the objects it owns directly and indirectly. The ownership tree is established at run-time: when new objects are instantiated, the owner of a new object is by default the sender of the `new` message.

Filters Methods belong to zero or more *topics*[1], and filters select sets of topics. From a given reference, a method is accessible only if its topics match the *in* and *out* filters of the boundaries crossed by the reference. The effect of filters is cumulative. An object can access all methods of its parent, children, and siblings. No filter is applied in the case of such message sends.

In Figure 1, the class `Page` uses the topic `navigation` as in and out filters. Other classes have empty filters. Figure 2 shows how filters are technically defined. Let us consider the methods in Figure 3 that deal with URL and site map generation. The methods belong to the `navigation` topic[2]:

- Pages can not access method `WebServer>>hostUrl` since web sites have empty out filters. To render the complete URL, pages must use `Site>>baseUrl`.
- Nested pages can access method `Site>>baseUrl` since the out filter of pages is `navigation`.
- In `Site>>siteMap`, the site flattens the tree of pages into a list. It can access method `Page>>fullUrl` on all pages in the list, since the in filter of pages is `navigation`.

Crossing Handlers Filters can lead to references that expose no methods. Outgoing references (crossing boundaries from inside to outside) of this kind are valid, while incoming references (crossing boundaries from the outside to the inside) are not. All incoming references in the system must expose at least one method. With this restriction, hiding all methods of an object expresses confinement.

This corresponds to the traditional principles of alias protection [28]: nodes of a list can reference the data they hold (outgoing references), but external (incoming) references to the nodes violate encapsulation and must be forbidden. Only the list can reference its nodes.

The event of an invalid reference transfer is reified and the corresponding *crossing handler* is triggered. By default, the crossing handler performs no action and raises an exception. It could however be adapted to perform some actions after which the reference transfer should be valid. Let us consider Figure 1, which uses the default crossing handler:

- When instances of `List`, `WebServer` and `Site` return a reference to objects they own, a crossing handler is fired since their in filter is empty. The default crossing handler will raise an exception and prevent the reference transfer.

[1] This terminology was chosen to avoid confusion with interfaces, categories, and protocols

[2] Readers unfamiliar with the syntax of Smalltalk might want to read the code examples aloud and interpret them as normal sentences: An invocation to a method named `method:with:`, using two arguments looks like: `receiver method: arg1 with: arg2`. Other syntactic elements of Smalltalk are: the dot to separate statements: `statement1. statement2`, and square brackets to denote code blocks or anonymous functions: `[ statements ]`.

- An instance of `Page` can return a reference to a subpages, since `Page`'s in filter contains the `navigation` topic.

Ownership Transfer When an object creates another object, it is assigned by default to be the owner of the new object. Since the default owner is not always appropriate, ownership can be transferred if necessary. Methods `Object>>owner` and `Object>>owner:` respectively query the current owner of an object, or modify it. Ownership transfer must preserve the tree structure of the ownership graph, and must not result in invalid incoming references.

2.1 Default Policy

By default, the set of *in filters* is empty, as well as the set of *out filters*. With an empty set of in filters, the topics of the methods are irrelevant and references to internal objects cannot be passed to the outside. The default crossing handler raises an exception when a reference to an internal object is passed to the outside, either as a return value or as a parameter. This corresponds to the classic owner-as-dominator policy [12].

Let us consider the web server in Figure 1. The list of sites is implemented as a list composed of nodes. Since the set of in filters is empty, an attempt to return a reference to a node will trigger the crossing handler which will raise an exception: the list is an aggregate and the nodes are effectively inaccessible outside the aggregate. With an empty set of out filters, objects within the boundary can only depend on the identity of objects outside the boundary.

3. Using Filters

Each object defines an implicit *ownership boundary*. We first show how implicit boundaries can be configured with filters to relax the owner-as-dominator policy. We then show how objects with no behavior can be added to a design and serve as *explicit boundaries*.

3.1 Iterators

Owner-as-dominator is too restrictive to implement common idioms like iterators: for efficiency the iterator must be owned by the aggregate to have access to internal data, but cannot then be returned to the outside [27].

In our approach, filters can easily be used to solve this situation. The list owns the iterator, which is then a sibling of the nodes and has full access to them. The in filters of the list contain the `iteration` topic, which match methods `next` and `current` of the iterator, shown in Figure 5. The iterator can be by consequence returned outside the list, while nodes cannot.

Several variants of ownership types using class nesting [6], ownership domains [2], relaxed constraints for dynamic aliases[3] [12] or additional access modifiers [23], have been devised to solve this problem. The implementation of dynamic ownership by Gordon and Noble [17, 28] relies on a special language feature to "export" objects to solve this issue. Filters and crossing handlers support this situation, while being general mechanisms.

3.2 Read-only References

With owner-as-dominator, encapsulated objects cannot be returned to the outside, which effectively prevents unwanted modification to the internal representation from ever happening. When internal state must be exposed, a safe alternative is to expose only a limited read-only view. This is known as representation observation [7].

[3] Static aliases correspond to references from instance variables. Dynamic aliases correspond to references from temporary variables, parameters, and return values. Static aliases are allocated in the heap. Dynamic aliases are allocated in the stack.

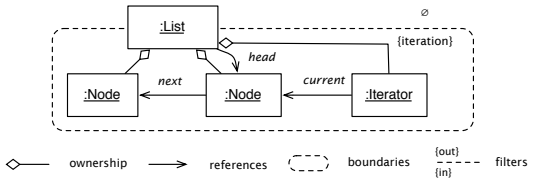


Figure 4. Iterators can be returned to the outside since they match the `iteration` topic.

```
Iterator>>next
  <topic:iteration>
   current := current next.

Iterator>>current
  <topic:iteration>
  ^ current data.
```

Figure 5. Methods belonging to the `iteration` topic

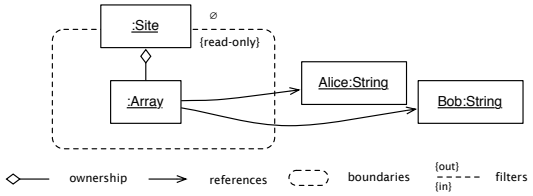


Figure 6. The site enables a read-only view of the internal array

Let us consider that each web site has several administrators that are stored in an array. Administrators can be changed only via a special administration page. The ability to obtain an unrestricted reference to the internal array from outside the web server would imply that the list can be freely changed. To prevent mutations, the method `Array>>at:` is assigned the topic `read-only` and the in filters of the web site matches the `read-only` topic. The situation is shown in Figure 6. This way, objects outside the web site only have a limited access to the array.

Since the effect of filters is cumulative, read-only access will be applied transitively to all objects within the boundary. This works well for nested and recursive structures, as was shown previously when limiting access to the `navigation` topic for web pages.

There have been several proposals for read-only references [7]. For dynamic languages, only few approaches have been proposed. Schaerli *et al.* proposed encapsulation policies [32], which enable fine-grained control of the interface objects expose. It however fell short in dealing with recursive structures. Arnaud *et al.* proposed a specific solution to this problem with read-only references [4]. In both cases, the policy is attached to a reference, not the object itself, and references with limited capabilities must be created explicitly. We believe these approaches are counter-intuitive since they define behavior based on the history of the reference instead of the dynamic context. Using the ownership topology appears to be much more natural.

3.3 Access Modifiers

The `public` and `protected` access modifiers can be simulated with explicit boundaries. Unlike implicit boundaries, explicit boundaries correspond to "non-domain" objects with no behavior of their own.

Each class categorizes its methods into one of two topics `public` and `protected`, in addition to other existing topics. Each object is

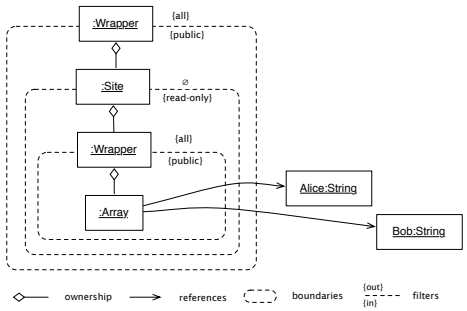


Figure 7. Each object is owned by a synthetic wrapper that exposes only `public` methods

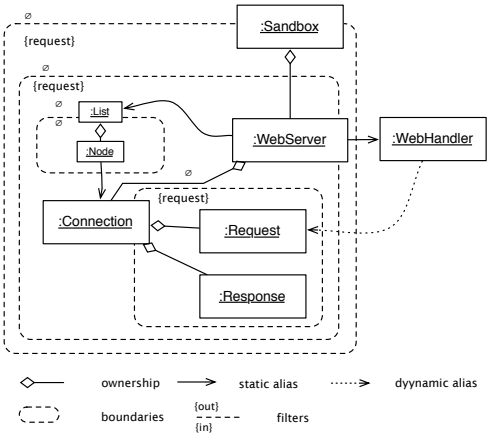


Figure 8. The web server is sandboxed

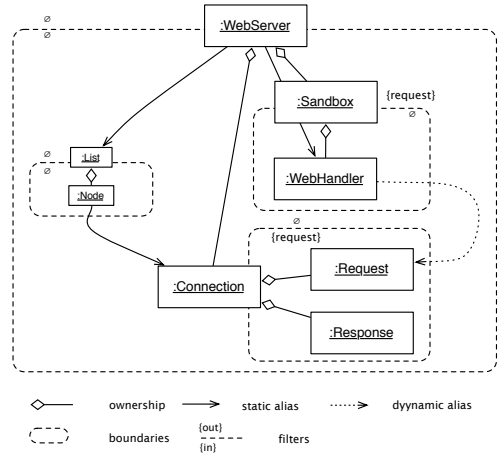


Figure 9. The handler is sandboxed

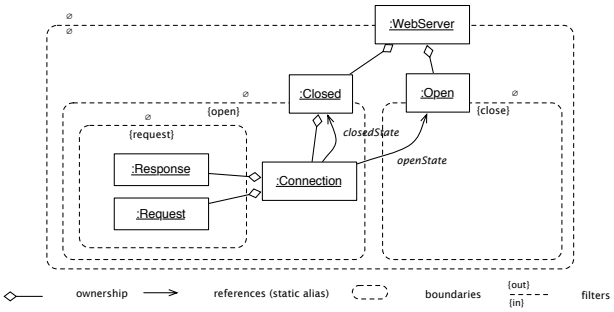


Figure 10. A connection can be owned either by the `closed` or `open` state.

then owned by a "wrapper" which exposes only public methods. Figure 7 shows this situation. Instead of owning the array directly, the web site owns now the wrapper of the array.

This strategy implements instance protected methods: an object cannot access the protected methods of another object. Whether the receiver of a message is `self` or an alias of `self` has no impact. The strategy is similar to accessibility of instance variables in Smalltalk.

This contrasts with Ruby and Newspeak. Ruby implements class protected methods. Newspeak implements instance protected methods, but the lack of consideration of the sender in the method lookup algorithm results in different semantics for self sends with `self` or an alias of `self` [4]. Since our approach works with objects and not classes, we cannot simulate `private`.

3.4 Sandboxing

An object has full access to its siblings. Therefore, for the sake of security, one might want to protect objects within an additional explicit boundary. We call this *sandboxing*.

Figure 9 shows the design of the web server with the web handler, and illustrates two forms of sandboxing. The server is a generic infrastructure that abstracts the HTTP protocol. It manages instances of `Connection`, `Request` and `Response` classes. HTTP request data can be read with `Request>>fields` and the response is produced with `Request>>sendResponse:stream:` that takes a stream and an HTTP response code. The actual treatment of the request is delegated to a `WebHandler`. The handler is an extension of the web server that implements the desired functionality, *e.g.*, list directories, evaluate templates, *etc.* The handler is an untrusted component from the point of view of the web server.

Two forms of sandboxing are possible, as shown in Figure 8 and Figure 9. In one case, the sandbox is around the web server, and in the other case it is inside the web server. In both cases, the web handler has only access to the `request` topic, which suffices to treat the request and produce the response.

3.5 First-class State

In our model, the interface an object exposes depends on its direct and indirect owners. While an object cannot "on its own" change the interface it exposes, we can very easily do so using first-class state [34]. Depending on which state it is owned, the object exposes a different set of operations. Ownership transfer is used to perform actual state changes.

Let us imagine that connections in Figure 9 have two states: open and closed. We model these settings with three objects, one being the connection itself, the two others the first-class states. The connection class defines two topics, `open` and `closed`. Each first-class state exposes one topic. When the connection is owned by the open state, it exposes only methods of the open topic. Inversely, when it is owned by the closed state, it exposes the closed topic. The connection must be owned by either state. Figure 10 shows such a situation. Similarly to sandboxing, the web server owns now the states instead of the connection. For objects that reference the connection, the pattern is transparent.

[4] §5.7 of the specification

Note that a useful variant of this technique can be used to "freeze" objects [20], after which they are immutable. All that is required is to implement a first-class "frozen" state, which filters out all mutating methods making the owned object effectively immutable.

4. Using Crossing Handlers

Examples in the previous chapter relied on the default crossing handler which raises an exception when an invalid reference transfer occurs. We now show how crossing handlers complement filters in useful ways.

4.1 Defensive Copying

Let us consider again the problem of the previous chapter with the administrators. With owner-as-dominator, encapsulated objects cannot be returned to the outside. When internal state must be exposed, it is a common practice to return a copy of the object. This technique is known as *defensive copying* [5].

Let us consider that each web site has several administrators whose names are stored in an array. Administrators can be changed only via a special administration page. The ability to obtain a reference to the internal array from outside the web server would imply that the list can be freely changed. A typical implementation of the `administrator` accessor would copy the array before returning it[5]:

```
Site>>administrator
^ administrator copy.
```

Languages do not have mechanisms to express such policies cleanly: developers must manually add code for copying objects whenever appropriate; copying and non-copying accessors might exist side by side and cause confusion; applying the technique systematically when modules grow is hard.

In our approach, the crossing handler can be overridden to implement the strategy. Whenever an encapsulated object is returned to the outside, the crossing handler is triggered. It copies the object being referenced and assigns it the sender as owner. The copy is then used for the reference transfer that is resumed.

```
Site>>handleCrossing: anObject sender: theSender
^ anObject copy owner: theSender.
```

4.2 Remoting

With distributed objects, objects can be local and remote. Remote invocations have a pass-by-value semantics while local invocations have a pass-by-reference semantics. Parameters of remote invocations must be serializable. Since the caller does not know whether the receiver is local or remote, all parameters must be serializable. This prevents useful optimizations, such as using implicit futures as parameters.

Local and remote objects can be organized into distinct boundaries in the ownership hierarchy. The local boundary exposes only serializable objects. When a non-serializable object is passed as parameter, the crossing handler is fired and can attempt to resolve the conflict. For instance, if an implicit future is passed as parameter of a remote invocation, the handler can wait until its value is available and pass it instead.

```
Local>>handleCrossing: aFuture sender: theSender
^ aFuture value.
```

[5] This example is intentionally close to Java's `Class.getSigners()` bug in early versions of the JDK. The method returned the internal array which could be tampered with by a malicious client to break security. This bug was motivational for ownership types [2].

4.3 Synchronization

Threads are objects. Objects that are owned by the threads are thread-local. Objects that are not owned by any thread are global. Rather than statically controlling thread locality [37], we control it dynamically. Threads expose only the `sharable` objects, *i.e.*, objects with at least one `sharable` method. Object that do not have `sharable` members cannot be passed to other threads or global object since the crossing handler triggers an error.

To pass thread local objects outside the boundary of the thread, they must be adapted first to become `sharable`. This adaptation can be done manually, or automatically in a crossing handler. For instance, a crossing handler can synchronize objects when they escape their threads by dynamically changing the class of the object (*e.g.*, Smalltalk's `become:` or `changeClassTo:`).

```
Thread>>handleCrossing: anObject sender: theSender
^ anObject synchronize.
```

Note that the handler is triggered independently of whether the receiver would create a static alias of the thread local object or not. It is more conservative than tracking whether objects are reachable by multiple threads.

5. Security

Dynamic ownership can be used to increase the security of open systems. We consider in this section the impact of reflection and ownership transfer from the perspective of security.

5.1 Ownership Transfer

Ownership transfer could be used to bypass the constraints imposed by filters and crossing handlers, and thus break encapsulation. In Figure 9, the handler is an untrusted component. Ownership is leveraged to constrain interactions between the handler and the web server to legal scenarios according to the principle of least privilege.

There are essentially two privilege escalation scenarios to consider: 1) a malicious object changes the owner of an object to obtain privileged access to it, and 2) a malicious object changes its owner to obtain privileged access to other objects.

Since ownership transfer is realized via regular message sends, it can be limited by using filters to mitigate the first threat: if ownership transfer is not exposed with in filters, external objects will not be able to transfer ownership of internal objects; if an object does not trust one of its internal objects, it can sandbox it (see subsection 3.4) and use an out filter to prevent ownership transfers. It is preferable to own only objects one trusts. Container objects should usually not own their content, *e.g.*, a list does not own the data it holds.

Filters and crossing handlers are however not sufficient to address the second threat. The web handler could for instance make the web request its owner. This way, it would be a sibling of the web response and have full access to it. It could use this privilege to break encryption protocols.

To prevent such a case, the new candidate owner must accept the transfer first. The web request would for instance reject ownership of the web handler. In our approach, different objects can specify different ownership transfer policies by overriding the hook `Object>>acceptOwnershipOf: anObject`. Following the principle of deny by default, all transfers are by default rejected. For convenience, the acceptance check is however bypassed if the initiator of the transfer is the new owner itself. That is, `anObject owner: self` always succeeds.

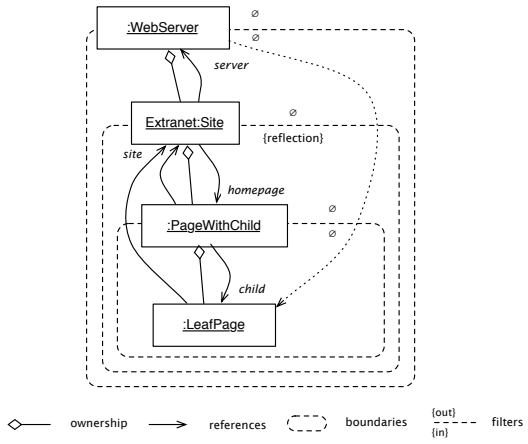


Figure 11. Reflection can be limited with filters. The dotted reference can be obtained only via reflection, since reflection bypasses accessibility and aliasing constraints.

5.2 Reflection

Unstratified reflective capabilities to query the class of an object (`class`), invoke methods (`perform`), and access the state of an object (`at` and `at:`) defeat encapsulation. For instance, it is possible with reflection to inspect the value of a field for which there is no accessor.

Since reflection is also realized via regular message sends, it can be limited using filters. If the out filters hide reflective methods but the in filters expose them, objects can only reflect on objects they own, but not arbitrary objects. This way, a site can for instance reflect on all the pages it owns transitively, but not on the page of another site. If we want to prevent reflection between siblings, sandboxing can be used in addition (see subsection 3.4). This would prevent that a web site reflects on another web site.

The effect of reflective methods is not subject to constraints imposed by filters and crossing handlers: it is possible to reflectively invoke a method that would otherwise be inaccessible, and reflective accesses to the state of an object bypass crossing handlers. This way, features to save and restore a website (including its pages) could for instance be implemented using a generic reflection-based serializer. If the effects of reflective methods were also constrained, many useful patterns that use reflection would not be applicable any longer.

Reflection can be used to break the encapsulation of an object at most one level down. Let us consider Figure 11. The web server can reflect on the homepage since the web site enables reflection. Despite the empty filters of the homepage, the web site can then reflectively obtain a reference to the leaf page. It would however fail to reflectively invoke methods on the leaf page or inspect its state.

6. Semantics

We have presented informally how our variant of dynamic ownership works with several examples. We describe in this section the semantics of the mechanisms more precisely, and how they can be integrated into a dynamic language. We omit custom crossing handler and use a set-theoretic approach to formalize the semantics of filters and the default crossing handler.

6.1 Ownership and References

The heap is a set of objects $\mathcal{O}$. The $world$ is a special object in the heap, $world \in \mathcal{O}$. The partial function $owner : \mathcal{O} \to \mathcal{O}$ maps an object to its owner, possibly the *world*. *owner* is defined for all objects except *world*. The *owner* function defines a partial order $\prec$ over the set of objects:

$$o_1 \prec o_2 \iff \exists n > 0, owner^n(o_1) = o_2$$

The *world* is the indirect owner of all objects: $o \prec world, \forall o \in \mathcal{O}$. The function *references* : $\mathcal{O} \to 2^{\mathcal{O}}$ defines the existing references (static or dynamic) across objects at a point in time.

6.2 Topics and Filters

For the purpose of explaining our model, we consider that methods are attached to objects, not classes. The details of method behavior is also irrelevant. We model only topics and filters as:

- *methods* : $\mathcal{O} \to 2^{\mathcal{M}}$ maps objects to methods names $m \in \mathcal{M}$;
- *topics* : $\mathcal{M} \to 2^{\mathcal{T}}$ maps method names to topic names $t \in \mathcal{T}$;
- *inFilters* : $\mathcal{O} \to 2^{\mathcal{T}}$ maps objects to set of topics that serve as in filters;
- *outFilters* : $\mathcal{O} \to 2^{\mathcal{T}}$ maps objects to set of topics that serve as out filters;

All methods belong to a special topic $all \in \mathcal{T}$, that can be used in filters.

If classes and inheritance were considered, rules for topic variance in overridden methods would need to be specified. Similarly to traditional access modifiers, subclasses can only make methods more visible, *i.e.*, they can only add topics to existing methods, not remove them.

6.3 Paths

The *ancestors* of an object o, $anc(o) = \{o' | o \prec o'\}$, is the set of all direct and indirect owners of that object, up to the $world$. Conversely, the *descendants* of an object, $desc(o) = \{o' | o' \prec o\}$ is the set of all objects that object owns directly or indirectly. The *depth* of an object, $d(o)$ is the cardinality of its set of ancestors, $d(o) = |anc(o)|$ The *first common ancestor* of two objects is:

$$com(o_1, o_2) = max(anc(o_1) \cap anc(o_2))$$

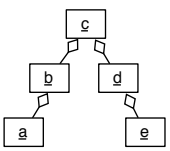
where max returns the object with the maximum depth. (Since the ownership relation forms a tree, there is a unique first common ancestor.)

The *outPath* from an object a to an object b is the sequence of ancestors of a up to the first common ancestor of a and b. Respectively, the *inPath* from an object a to an object b is the sequence of descendants starting from the common ancestors of a and b down to b:

$$outPath(a,b) = \begin{cases} owner(a), outPath(owner(a), b) & \text{if } d(a) > D \\ () & \text{if } o/w \end{cases}$$

$$inPath(a,b) = \begin{cases} inPath(a, owner(b)), b & \text{if } d(b) > D \\ () & \text{if } o/w \end{cases}$$

where $D = 1 + depth(com(a,b))$. The *path* from a to b is then the sequence $inPath(a,b), com(a,b), outPath(a,b)$. In the ownership tree below, $outPath(a,e) = (b)$ and $inPath(a,e) = (d)$. The *path* between a and e is (b, c, d).



6.4 Accessibility

An object $a \in \mathcal{O}$ can send message $m \in \mathcal{M}$ to object $b \in \mathcal{O}$ only if the topics of method m match the filters along the path from a to b. Formally, the condition can be expressed as a predicate: $\mathit{isAccessible}(a, b, m) \iff \forall o \in \mathit{outPath}(a, b), \mathit{outFilters}(o) \cap \mathit{topics}(m) \neq \emptyset$ and $\forall o \in \mathit{inPath}(a, b), \mathit{inFilters}(o) \cap \mathit{topics}(m) \neq \emptyset$.

Let us consider Figure 7. If method `Array>>at` has topics `read-only` *and* `public`, it is visible outside of the web site, since each boundary exposes either the topic `read-only` *or* the topic `public`.

Note that $\mathit{outPath}(a, b) = \emptyset$ and $\mathit{inPath}(a, b) = \emptyset$ if a and b are parent and child, child and parent, or siblings. In these cases, the accessibility condition is trivially satisfied, and both objects can access all methods of the other.

6.5 Validity of References

A reference is valid if the boundaries that it crosses into expose at least one method; the boundaries that it crosses out could hide all methods, though. The validity of a reference between two objects can be defined as a variation of the accessibility for an individual method: $\mathit{isValidReference}(a, b) \iff \exists m \in \mathcal{M}, \forall o \in \mathit{inPath}(a, b), \mathit{inFilters}(o) \cap \mathit{topics}(m) \neq \emptyset$. Note that unlike *isAccessible*, *isValidReference* considers only the *inPath* between the two objects. The system must establish only valid references: $\forall a, b \in \mathcal{O}, b \in \mathit{references}(a) \Rightarrow \mathit{isValidReference}(a, b)$.

6.6 Instantiation

When an object o requests the creation of another object, a new object is allocated in the heap $\mathcal{O}$ and it is assigned o as its owner. A reference from o to the new object is established. Since o and the new object are parent and child, o can access all methods of the new object. This preserves the partial order $\prec$ and the consistency of the references at run-time.

6.7 Aliasing

Since references cover both static and dynamic aliases, passing references as parameters or return values of method invocations creates new references between objects. The system must establish only valid references and prevent invalid reference transfer with an error.

Let us first consider return values. Let s, r, v be respectively the sender, receiver and return value of a message. The reference between r and v is valid, $v \in \mathit{references}(r)$ and $\mathit{isValidReference}(r, v)$, otherwise an error would have been raised previously. The system must ensure that the reference between s and v will be valid as well.

In practice, one does not need to check all boundaries along the $\mathit{inPath}(s, v)$ to assess the validity of the reference. Let us define path truncation $\ominus$:

$$a, b \ominus c = \begin{cases} a & \text{if } b = c \\ a, b & \text{if } o/w \end{cases}$$

Since the reference from r to v is known to be valid, it suffices to assess whether the boundaries $\mathit{inPath}(s, v) \ominus \mathit{inPath}(r, v)$ accept incoming references to v.

Figure 12 illustrates the references involved when `p1` sends a message to `s2` that returns a reference to `p2.1`. The in boundaries crossed between `p1` and `p2.1` are {`s2`,`p2`}. The in boundaries between `s2` and `p2.1` are {`p2`}. The list of boundaries to check corresponds to inPath(`p1`,`p2.1`) $\ominus$ inPath(`s2`,`p2.1`), which is {`s2`} in this case. Since `s2` has no in filters, the reference to `p2.1` exposes no method and `s2`'s crossing handler is fired. The handler raises an exception and prevents `p1` from obtaining a reference to `p2.1`.

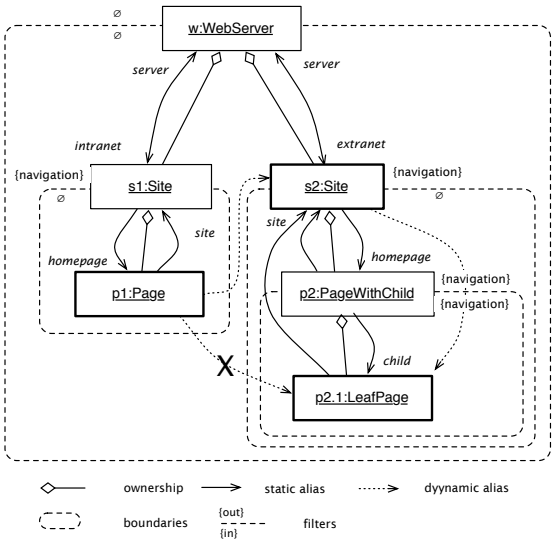


Figure 12. `p1` invokes a method on `s2` which returns a reference to `p2.1`. The in boundaries crossed from `p1` to `p2.1` are {`s2`,`p2`}. The in boundaries crossed from `s2` to `p2.1` are {`p2`}. The set of crossing handlers to check corresponds to {`s2`,`p2`} $\ominus$ {`p2`} = {`s2`}. `s2`'s crossing handler raises an exception and prevents that `p1` obtains a reference to `p2.1` (reference marked with a X).

The treatment of parameters is similar. Let s, r, p be respectively the sender, receiver and the parameter of interest of a message. The reference between s and p is known to be valid. The system must ensure that the reference between r and p will be valid as well. In practice, it suffices to assess whether the boundaries $\mathit{inPath}(r, p) \ominus \mathit{inPath}(s, p)$ accept incoming references to p.

Custom crossing handler could be modeled by rewriting message sends $o.m(p^*, \ldots)$ as $\lfloor o.m(\lceil p^* \rceil_{\mathbf{self},o}, \ldots) \rfloor_{\mathbf{self},o}$. The operators $\lfloor \ldots \rfloor_{\mathbf{self},o}$ and $\lceil \ldots \rceil_{\mathbf{self},o}$ perform the checks for the parameters and return value as described above. They invoke the special method *handleCrossing* $\in \mathcal{M}$ for each boundary that is crossed inside.

6.8 Ownership Transfer

The owner of an objects can be changed at run-time. The ownership transfer must preserve the partial order $\prec$ and the consistency of the references at run-time, though. Failure to do so results in a run-time error. In practice, this implies that the new owner must not be a descendant of the current owner, and that the system must verify the validity of all references to descendants of the impacted object.

Custom ownership transfer policies (see subsection 5.1) could be modeled by invoking a special method *acceptOwnershipOf* $\in \mathcal{M}$ on the new candidate owner. If the invocation returns false, the transfer is rejected.

7. Implementation

We have implemented a prototype of our variant of dynamic ownership in Pharo Smalltalk. Essentially, message sends must be intercepted to enforce the accessibility defined by filters, and execute crossing handlers if necessary. We implemented a compiler that transforms the original source and weaves it with additional logic, similarly to the technique used by Rivard to implement contracts [31]. Each call site is rewritten with one level of indirection that performs the additional logic, and then sends the original message.

```
| myself |
myself := self.
self print: anObject.                          "1"
myself print: anObject.                        "2"
[ :p | self print: p ] value: anObject.        "3"
[ :p | myself print: p ] value: anObject.      "4"
```

Figure 13. Message sends with similar intent

Closures and `self` The four statements in Figure 13 have the same intent: they send message `print:` to `self` with `anObject` as parameter. Self-sends are never filtered. In the first case, the self send is statically detected and is not rewritten with one level of indirection. The second case uses an alias `myself` of `self`. The self-send is not detected statically and the message send goes through one level of indirection. Cases 3 and 4 illustrate the peculiarities of closures. Within closures, `self` is not bound to the closure itself, but to the enclosing object. By consequence, the owner of a closure is the owner of the enclosing object. Therefore, despite the fact that closures are objects, `[...] value: anObject` in Figure 13 leads to a reference transfer that is within the same boundary. Also, since we rewrite the call sites, non-local returns within closures are correctly handled.

Primitive Types Instances of `String`, `SmallInteger`, *etc.* are immutable singleton objects. Dynamic ownership raises the question whether such scalar values can be owned or not. We can argue that since they are immutable, so leaking such a value cannot compromise internal invariants of the object and it therefore makes no sense to own scalar values. On the other hand, such a value might still represent sensitive information that one does not want to expose. In this case, similar values need to be treated as distinct objects since their owner can differ. For instance, two objects could each own an instance of a string with the same value. In our implementation, these objects are owned by the `world` and are not subject to dynamic accessibility and aliasing checks. This decision also applied to `nil`, the unique instance of `UndefinedObject`.

Control Flow Control flow is realized in Smalltalk with message sends. The most common control flow messages (`ifTrue:ifFalse`, `whileTrue:`, *etc.*) are not rewritten. The same treatment could be extended to boolean operators (`or:` and `and:`), assuming they are not redefined in other classes.

Ownership Transfer After the owner of an object *o* has changed, the system must ensure that all references to objects within *o*'s implicit boundary are still valid (*i.e.*, expose at least one method). In Pharo, `Object>>pointersTo` performs a linear scan of all objects in memory and returns all objects pointing to a particular object.

First-class Classes Smalltalk has no constructors. Objects are instantiated by sending the message `new` to the corresponding classes[6]. Classes are objects, and constructors are class-side methods that act as factories. In our implementation, classes are owned by the `world`. To assign a default owner to a newly instantiated object, our implementation intercepts the message `new`. The default owner is the sender of the `new` message. First-class classes pose two challenges to our model. First, constructor methods that use `self new` internally will produce objects owned by the class itself. Second, since classes are owned by the `world`, the constructor method might not be accessible if a boundary hides it. In our implementation, classes are treated in a special way to circumvent these problems.

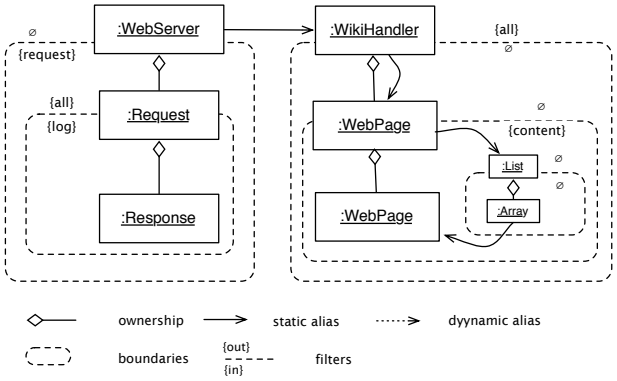


Figure 14. Key objects of the web server and their organization in an ownership structure.

8. Experiments

We used an existing Smalltalk web server[7] to experiment with filters and crossing handlers. We report here on our experience using these mechanisms.

8.1 Adapting the Web Server

Figure 14 shows the key objects of the web server and their organization in an ownership structure. The examples in the previous chapters were intentionally very similar to the design of this web server. The key classes are `WebServer`, `WebRequest` and `WebResponse`. Since the web server is only an infrastructure component to handle the HTTP protocol, we implemented in addition a minimal `WikiHandler` to view and edit `WebPage`. The web server uses several classes of the collection hierarchy that we adapted for dynamic ownership. The system consists of about 20 classes.

Ownership and Topics Two topics were mainly used to relax ownership: `request` and `content`. The first topic allows the web server to pass the request to the wiki handler. To process the request, the handler first locates the web page for the requested URL, and then renders the web page. The `request` topic enables the wiki to read (but not write) attributes of request such as the URL. The `content` topic enables the handle to access nested web pages and renders them. The response is owned by the request. It cannot be accessed by the web server. Because the web server logs entries it creates with `WebServer>>logEntryFor: request response: response`, a third topic `log` was necessary. Only 14 methods needed to be annotated, and the effort was low. Note that implementing `Request>>createLogEntry` would be more object oriented and wouldn't need the `log` topic. In that sense, ownership favors object-orientation.

Factory Methods By default, the owner of an object is the one that invoked the corresponding factory method (see section 7). This is not always appropriate. For instance, `Collection>>select:` returns a collection that must have the same owner as the original collection. Such methods must be adapted to transfer the ownership after creation. The new owner accepts the transfer if it comes from an object it already owns.

Cloning Cloning should produce a new object that is indistinguishable from the original one. The implementation in `Object` was adapted to produce a copy of the object with the same owner as

[6] In Pharo, the primitive method is actually called `basicNew`

[7] Original: http://www.squeaksource.com/WebClient.html Experiment: http://ss3.gemstone.com/ss/DynOwn.html

```
fields
| fields |
fields := Dictionary new.
self getFields associations:[:a| fields add: a].
self postFields associations:[:a| fields add: a].
^fields
```

Figure 15. This code violates ownership and encapsulation since it adds the internal associations of a dictionary to another one without copying.

```
fields
| fields |
fields := Dictionary new.
self getFields keysAndValuesDo:
            [:key :val | fields at: key put: val ].
self postFields keysAndValuesDo:
            [:key :val | fields at: key put: val].
^fields
```

Figure 16. The modified code does not access the internal state of dictionaries.

the original object. The implementation must be further adapted for specific classes. For instance, copying a dictionary copies the associations its owns as well. This kind of copying is called sheep cloning [21]. In this case, the owner of the copied associations must change to be the copied collection. Cloning can be considered to be a special factory method.

Utility Methods Class-side utility methods raise problems of ownership transfer similar to constructor methods (see section 7). We moved utility methods to a trait that could be reused whenever necessary. Code like `WebUtils decodeUrl: aString` is then rewritten as `self decodeUrl: aString`. This reformulation avoids problematic boundary crossings, and accessibility restrictions. The trait contains 7 such methods.

Ownership Bugs The method `WebRequest>>fields` in Figure 15 violates encapsulation and ownership: it adds the internal associations of a dictionary to another one without copying. It is the goal of dynamic ownership to identity such violations. Figure 16 shows the modified code without the aliasing bug. This example shows that such code exists and that dynamic ownership can detect design anomalies.

8.2 Performance

Naturally, enforcing dynamic ownership entails an overhead. The dynamic checks entail finding the common owner to two objects, and several manipulations of lists. Also, listing and comparing topics is expensive. Since our implementation was not optimized for performance, the overhead is significant.

The check for validity of references after an ownership transfer `transfer` was disabled during our evaluation of performance. Indeed, `pointersTo` is a very expensive operation that performs a linear scan of all objects in memory at the application level. Ownership transfers are occasional, so we can afford some overhead for them, but it would require support from the virtual machine in a production implementation.

Certain message sends can be optimized easily: messages sent to an alias of `self` can bypass all dynamic checks; messages sent to a child require only checking whether the return value crosses a boundary; messages sent to a parent require only checking whether the arguments cross a boundary; messages sent to a sibling require checking whether the arguments and the return value cross a boundary, but do not require checking for accessibility of methods.

self (static)	self (dynamic)	parent to children	children to parent	sibling to sibling	other
3	25	28	41	47	390
3	25	28	41	48	87

Figure 17. Times (ms) for 10'000 executions of the method `returnParameter: 42` invoked on itself, a child, its parent, a siblings, or another object. The method takes a parameter as argument and returns it as-is. The first line shows measures when caching of accessibility checks is disabled, the second when it is enabled.

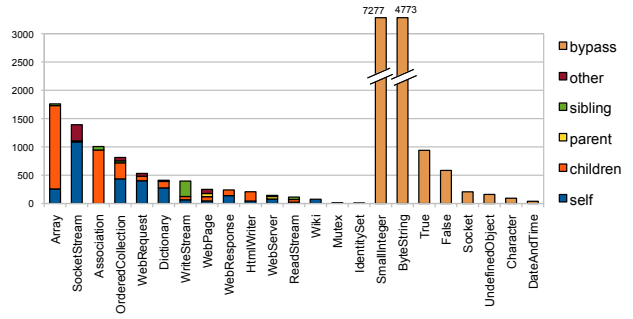


Figure 18. Distribution of message sends for the key classes (the graph is best viewed in color). The majority of interactions between objects are message sends that can be optimized.

Figure 17 shows micro benchmarks for message sends. Static self sends (*i.e.*, using `self`) are not rewritten and correspond to the performance of the original system. Dynamic self sends (*i.e.*, using an alias of self) go through one level of indirection, but bypass all dynamic checks. Dynamic self sends indicate that the level of indirection itself entails a degradation of factor 8 (25 vs. 3). Messages sent to a child, a parent, or a sibling can be partly optimized. Messages sent to "other" objects in the ownership structure require the execution of expensive logic to enforce the generic accessibility and aliasing constraints. The two lines show the times when the results of accessibility checks (see subsection 6.4) are cached or not. The overhead of the micro benchmarks range between factor 8 (25 vs. 3) and 29 (87 vs. 3) when caching is enabled. Note that passing a different parameter, or more than one parameter, might degrade the performance further.

Figure 18 shows the distribution of message sends across the five categories for key classes in the web server. On the left are measures for objects subject to dynamic checks. On the right are measures for primitive types (see section 7) not subject to dynamic checks. We can see that the majority of interactions between objects consists of message sends that can be optimized. Macro benchmarks of the code of the web server using a mock socket (independent of IO) indicate a degradation of about factor 13 when caching of accessibility checks is enabled (204 vs. 2521). This overhead must be put in perspective with the overhead of the level of indirection itself that is of factor 8.

9. Discussion

Our experiments show that our variant of dynamic ownership can be effectively put to work to better control aliasing and accessibility. We found them flexible enough to accommodate an existing design, and at the same time constraining enough to highlight design anomalies. The system we studied was however small. We plan in future work to investigate how our approach scale for bigger systems. Here we sketch some improvements that our experiment suggests.

Flexibility with Impersonation Constructor methods and class-side utility methods are problematic since classes are owned by the `world`. For instance, invoking the copy constructor `OrderedCollection from: aCollection` implies that the reference `aCollection` is first transferred to the object `OrderCollection`. The transfer might raise a crossing error. Rather than solving the problem specifically for classes, we plan to explore *impersonation* of objects.

When object r executes an impersonated method in response to a message from object s, the owner of r resolves temporary to the owner of s for the current execution. Impersonated methods could be prefixed for instance with `%`, *e.g.*,`OrderedCollection>>%new: aCollection ^self new withAll: aCollection`. To avoid a security breach, the sender must authorize the impersonation when the message is sent, *e.g.*,`OrderedCollection %from: aCollection`. Message sends prefixed with `%` can be dynamically dispatched to regular or impersonated methods; message sends without the prefix can be dynamically dispatched only to regular methods.

This enhancement enables forms of borrowing. Let us consider that the web server of Figure 1 uses a serializer to save and restore the web pages. If the serializer is a utility class owned by the `world`, it cannot access the web pages. To grant access to the web pages, the web server can let the serializer borrow the web pages by transferring its ownership to the serializer. This however assumes that the serializer accepts the transfer, and will transfer the ownership back. Impersonation is more appropriate to grant the serializer temporary access to the web pages.

Improving Performance The dynamic checks follow the path between objects in the ownership tree. The identity of objects along the path is however not relevant, but only the classes of those objects matter to the dynamic checks. Computing information about the relative positioning of objects in the ownership tree is expensive. Our implementation caches the results of accessibility checks, but not the aliasing checks. We plan to investigate how this information could be effectively computed and cached in this case as well. The cache could be maintained per object, or per call site.

In tracing VM [16], traces are recorded and compiled to native code at run-time. Recorded traces are reused by speculating on branching and types. Guards in the traces validate this speculation, and if a guard fails, the trace exits. To support dynamic ownership, a tracing VM could speculate on the relative positions of objects in the ownership tree, in addition to branches and types. Checking the validity of references after ownership transfers would also benefit from VM support, since it is an expensive operation.

Note that dynamic ownership can be considered to be a special kind of contract. Filters could be turned off during execution of production systems for performance reasons. As long as the system does not use custom crossing handlers, they can be turned off as well (indeed, if the system uses custom crossing handlers, removing them might impact the application behavior).

Composition with Dynamic Topics It is easy to assign topics to a class as long as they are used in a specific context. If a class is reused in multiple contexts, it might be complicated to assign topics that satisfy all contexts. Let us consider Figure 6: the site enables a read-only view of the internal array. Let us imagine that the site uses another array internally to keep encryption keys. The second array will also be read-only since there is no way to distinguish from both contexts where arrays are used. The solution to this problem would be to have topics per objects: two instances of the same class could have different topics. We plan to achieve this with rewriting rules attached to individual objects: one array rewrites the topic `read-only` to `admin-read-only`, the other to `key-read-only`. The web site can decide to expose only `admin-read-only`. For composition, a rewriting rule attached to object o would impact o, but also objects owned by o.

10. Related work

The risks related to aliasing have been since long recognized [18], and our work relates to a large body of research.

The closest related work is Dynamic Ownership by Gordon and Noble [17], which itself built on previous concepts of Dynamic Alias Protection [29], Flexible Alias Protection [28] and Ownership Monitoring [19]. In contrast to Dynamic Ownership, filters enable objects to be accessed outside their owner's boundary via a possibly limited interface. Also, we not only check accessibility when messages are sent, but also aliasing of objects in return values and arguments. Flexible Alias Protection [28] enforces "external independence", a property which states that internal objects must not depend on mutable state of external objects: in Dynamic Ownership, invocations to external objects raise an exception if state is mutated or if a value is returned. Our approach can be used to define a topic that is used to categorize such legal methods, but it cannot raise an exception if a method is wrongly categorized.

Ownership Types Since the work by Clarke *et al.* [14] that introduced the owners-as-dominators model, many variants of static ownership types have been proposed. Similarly to our work, these approaches aim at relaxing the owner-as-dominator model to regain flexibility. In an extension of their previous work [12], Clark *et al.* enable dynamic aliases to expose internal objects such as iterators. Boyapati *et al.* used inner classes instead [6]. With Ownership Domains [2, 3], objects can be organized into various domains with different access constraints. Universe Types [25] similarly partition objects into universes and control references between them. Variant Ownership Types [23] parameterized types with an accessibility context in addition to the ownership context, thus giving more fine-grained control over aliasing. In these mechanisms, a member of a class can be accessed if its type can be named. Filters and crossing handlers are dynamic. Gradual Ownership[33] combines static and dynamic typing. Dynamic checks are introduced for code that has not been statically typed. Other variants of ownership types exist which address other aspects of aliasing, for instance uniqueness of references [8, 30], thread-locality [37] or data transfer between actors [13]. Our examples of crossing handlers drew inspiration from the two latter works.

Limited Interfaces In addition to accessibility, Variant Ownership Types [23] can specify whether references are writable or read-only. Universe Types [25] enforce the owner-as-modifier discipline, where read-only references across universes are allowed, but only the owner of an object can modify it. Our approach can encode the owner-as-modifier discipline by exposing a special `read-only` topic. Several languages can define write-once variables, *e.g.*, C++'s `const` and Java's `final` keywords. Used with references, `const` does not provide transitive read-only access. Schaerli *et al.* proposed encapsulation policies [32], which enable policies to be bound to references, but does not consider transitivity. There have been several proposals of type systems that support transitive read-only references [7, 38] (independently of ownership). Arnaud *et al.* proposed a variant of transitive read-only references for dynamic languages [4]. Filters are flexible and able to expose limited interfaces; an interface with only read-only methods is just a special case. In contrast to deep read-only references, read-only access is only transitive to objects within the boundary. It can be considered as a benefit, or a limitation depending on the context. Our approach is syntactic and it assumes that methods have been correctly categorized in the `read-only` topic by developers. Traditional access modifiers can be used to limit interfaces. Modifiers can implement class privacy or object privacy. Our approach implements object privacy, which was shown to be more intuitive [36]. Arbitrary accessibility rules can be easily implemented with techniques that reify message sends, such as composition filters [1]. The Law of

Demeter [22] is a design principle which dissuades invocations to objects returned by previous invocations. Organizing the design in layers, where objects in a given layer can only call objects in the layer below, is a way to enforce the law. The law of Demeter, layers, and confinement with ownership are design principles that prevent interaction between distant entities.

Security Encapsulation controls accesses from external to internal objects; secure programming controls accesses from an object to its external environment. Global namespaces compromise security since accesses to global namespaces cannot be controlled. In Java, access to the class namespace can be controlled with class loaders and security managers, which are mechanisms outside of the base language. In the object-capability model [24], objects can only send messages to objects that have been obtained previously with message sends. In this model, global namespaces and reflection are loopholes. Filters can be used to limit access to external resources, since filters work in both in and out directions. To control interactions between modules, objects can be wrapped into membranes [15, 24, 35], which transitively impose revocability on all references exchanged via the membrane, both inward and outward. When the membrane is revoked, the wrapped module is guaranteed to become eligible for garbage collection; revoked references raise exceptions when used. Ownership boundaries resemble membranes that intercept outward transfer of references. Newspeak is a language that follows the object-capability model. In Newspeak, external dependencies must be provided when an object is created [10]; there is no global namespace, only nested virtual classes. Also, Newspeak decouples reflection from classes via mirrors [9]. Tribal Ownership [11] exploits class nesting to define an implicit ownership structure for objects. From the perspective of security, ownership transfer must be limited. Ownership transfer is hard to support in static type systems [26], but very natural in a dynamic approach.

11. Conclusions

We have proposed a variant of dynamic ownership with filters and crossing handlers. With these mechanisms, the owner-as-dominator policy can be relaxed to control the sharing of internal objects in a flexible way.

We have illustrated our approach with several examples and experimented with filters and crossing handlers by adapting a web server. Our conclusions are the following:

1. Filters and crossing handlers can be effectively put to work to better control aliasing and accessibility. We found our variant of dynamic ownership flexible enough to accommodate an existing design, and at the same time constraining enough to highlight design anomalies.
2. The strength of our approach lies in the cumulative effect of filters. Filters at various boundaries compose naturally to define the specific interface that is exposed to another object. Boundaries can filter independent concerns.
3. Objects can be easily confined, not only within their direct owner, but also within indirect owners. Outside of their direct owner, only limited interfaces are exposed. Object interactions tend to be local with respect to the position of objects in the tree, *i.e.*, interactions between distant objects in the tree are rare. Filters fit well with this locality.
4. Filters have a wider applicability than crossing handlers. Crossing handlers can however facilitate the implementation of certain patterns, and follow the tradition of providing reflective hooks in dynamic languages, such as Smalltalk's `doesNotUnderstand:`.
5. Since accessibility is defined based on the relative positions of objects in the ownership tree, ownership transfer require approval by the new owner or it might introduce an encapsulation breach.
6. Reflection can be scoped and limited with filters.

We believe the our model is easy to use and promotes object-orientation. We plan to investigate how the overhead of our implementation can be reduced, possibly by using additional caches. Also, we plan to investigate how filters scale to larger code base and introduce topic rewriting to avoid collision in topic names.

Acknowledgment

We would like to thank Jorge Ressia, Marcus Denker, and Andrea Caracciolo for reviews of earlier drafts of our paper. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Synchronizing Models and Code" (SNF Project No. 200020-131827, Oct. 2010 - Sept. 2012).

References

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In *Proceedings of the Workshop on Object-Based Distributed Programming*, ECOOP '93, pages 152–184, London, UK, UK, 1994. Springer-Verlag. ISBN 3-540-57932-X. URL `http://dl.acm.org/citation.cfm?id=646775.705734`.

[2] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP*, pages 1–25, 2004.

[3] J. Aldrich, C. Chambers, and D. Notkin. Archjava: connecting software architecture to implementation. In *ICSE*, pages 187–197, 2002.

[4] 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 on Objects, Models, Components, Patterns (TOOLS EUROPE'10)*. LNCS Springer Verlag, July 2010. URL `http://www.bergel.eu/download/papers/Berg10eReadOnly.pdf`.

[5] J. Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008. ISBN 0321356683, 9780321356680.

[6] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. *SIGPLAN Not.*, 38(1):213–223, 2003. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/640128.604156.

[7] J. Boyland. Why we should not add readonly to Java (yet. In *In FTfJP*, pages 5–29, 2005.

[8] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 2–27, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-42206-4. URL `http://dl.acm.org/citation.cfm?id=646158.680004`.

[9] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. *SIGPLAN Not.*, 39:331–344, October 2004. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1035292.1029004. URL `http://doi.acm.org/10.1145/1035292.1029004`.

[10] G. Bracha, P. von der Ahé, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda. Modules as objects in Newspeak. In *ECOOP*, pages 405–428, 2010.

[11] N. Cameron, J. Noble, and T. Wrigstad. Tribal ownership. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 618–633, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6. doi: http://doi.acm.org/10.1145/1869459.1869510. URL `http://doi.acm.org/10.1145/1869459.1869510`.

[12] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. *SIGPLAN Not.*, 37(11):292–310,

Nov. 2002. ISSN 0362-1340. doi: 10.1145/583854.582447. URL http://doi.acm.org/10.1145/583854.582447.

[13] D. Clarke, T. Wrigstad, J. Östlund, and E. B. Johnsen. Minimal ownership for active objects. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 139–154, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-89329-5. doi: http://dx.doi.org/10.1007/978-3-540-89330-1_11. URL http://dx.doi.org/10.1007/978-3-540-89330-1_11.

[14] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. *SIGPLAN Not.*, 33(10):48–64, Oct. 1998. ISSN 0362-1340. doi: 10.1145/286942.286947. URL http://doi.acm.org/10.1145/286942.286947.

[15] T. V. Cutsem and M. S. Miller. On the design of the ECMAScript reflection api. Technical report, Vrije Universiteit Brussel, 2012.

[16] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542528. URL http://doi.acm.org/10.1145/1542476.1542528.

[17] D. Gordon and J. Noble. Dynamic ownership in a dynamic language. In *Proceedings of the 2007 symposium on Dynamic languages*, DLS '07, pages 41–52, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-868-8. doi: http://doi.acm.org/10.1145/1297081.1297090. URL http://doi.acm.org/10.1145/1297081.1297090.

[18] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Mess.*, 3(2):11–16, Apr. 1992. ISSN 1055-6400. doi: 10.1145/130943.130947. URL http://doi.acm.org/10.1145/130943.130947.

[19] S. Kent and I. Maung. Encapsulation and aggregation. In *In TOOLS Pacific 18*. Prentice Hall, 1995.

[20] K. R. Leino, P. Müller, and A. Wallenburg. Flexible immutability with frozen objects. In *VSTTE '08: Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments*, pages 192–208, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-87872-8. doi: http://dx.doi.org/10.1007/978-3-540-87873-5_17.

[21] P. Li, N. Cameron, and J. Noble. Cloning in ownership. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '11, pages 63–66, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0942-4. doi: 10.1145/2048147.2048175. URL http://doi.acm.org/10.1145/2048147.2048175.

[22] K. J. Lieberherr and I. M. Holland. Assuring good style for object-oriented programs. *IEEE Softw.*, 6(5):38–48, Sept. 1989. ISSN 0740-7459. doi: 10.1109/52.35588. URL http://dx.doi.org/10.1109/52.35588.

[23] Y. Lu and J. Potter. On ownership and accessibility. In *In ECOOP'06, volume 4067 of LNCS*, pages 99–123. Springer-Verlag, 2006.

[24] M. S. Miller and J. S. Shapiro. Paradigm Regained: Abstraction Mechanisms for Access Control. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 224–242, 2003.

[25] P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*, pages 131–140. Fernuniversität Hagen, 1999. Technical Report 263.

[26] P. Müller and A. Rudich. Ownership transfer in universe types. *SIGPLAN Not.*, 42(10):461–478, Oct. 2007. ISSN 0362-1340. doi: 10.1145/1297105.1297061. URL http://doi.acm.org/10.1145/1297105.1297061.

[27] J. Noble. Iterators and encapsulation. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33)*, pages 431–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0731-X. URL http://dl.acm.org/citation.cfm?id=832260.833174.

[28] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *ECOOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 158–185, London, UK, 1998. Springer-Verlag. ISBN 3-540-64737-6.

[29] J. Noble, D. Clarke, and J. Potter. Object ownership for dynamic alias protection. In *In Proceedings TOOLS '99*, pages 176–187. Society Press, 1999.

[30] J. Östlund, T. Wrigstad, D. Clarke, and B. Åkerblom. Ownership, uniqueness, and immutability. In R. F. Paige and B. Meyer, editors, *Objects, Components, Models and Patterns, 46th International Conference, TOOLS EUROPE 2008*, volume 11 of *Lecture Notes in Business Information Processing*, pages 178–197. Springer, 2008. ISBN 978-3-540-69823-4. doi: http://dx.doi.org/10.1007/978-3-540-69824-1_11.

[31] Rivard. Smalltalk: a reflective language. *In Proceedings of Reflection'96*, 1996.

[32] N. Schärli, A. P. Black, and S. Ducasse. Object-oriented encapsulation for dynamically typed languages. *SIGPLAN Not.*, 39(10):130–149, Oct. 2004. ISSN 0362-1340. doi: 10.1145/1035292.1028988. URL http://doi.acm.org/10.1145/1035292.1028988.

[33] I. Sergey and D. Clarke. Gradual ownership types. In *ESOP*, pages 579–599, 2012.

[34] J. Sunshine, K. Naden, S. Stork, J. Aldrich, and E. Tanter. First-class state change in Plaid. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 713–732, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048122. URL http://doi.acm.org/10.1145/2048066.2048122.

[35] T. Van Cutsem and M. S. Miller. Proxies: design principles for robust object-oriented intercession apis. *SIGPLAN Not.*, 45(12):59–72, Oct. 2010. ISSN 0362-1340. doi: 10.1145/1899661.1869638. URL http://doi.acm.org/10.1145/1899661.1869638.

[36] J. Voigt, W. Irwin, and N. Churcher. Intuitiveness of class and object encapsulation. In *6th International Conference on Information Technology and Applications*, 2009.

[37] T. Wrigstad, F. Pizlo, F. Meawad, L. Zhao, and J. Vitek. Loci: Simple thread-locality for java. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 445–469, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03012-3. doi: 10.1007/978-3-642-03013-0_21. URL http://dx.doi.org/10.1007/978-3-642-03013-0_21.

[38] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kie, un, and M. D. Ernst. Object and reference immutability using java generics. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 75–84, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-811-4. doi: 10.1145/1287624.1287637. URL http://doi.acm.org/10.1145/1287624.1287637.