Flexible Dynamic Ownership in Smalltalk

Bachelorarbeit

der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Pascal Maerki

21.02.2013

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz
Erwann Wernli
Institut für Informatik und angewandte Mathematik

Abstract

Dynamic Ownership is an effective way to ensure encapsulation, but it is too restrictive. We propose a more flexible variant of Dynamic Ownership based on two mechanisms: filters and crossing handlers. Filters define interfaces for accessing objects and crossing handlers control aliasing. This thesis details our approach and reports on the implementation of a prototype in Pharo Smalltalk. We describe the adaptation of a Smalltalk web server with our approach and assess its performance. We conclude that our approach is flexible enough to suit existing designs, but our prototype entails a significant performance overhead.

Contents

Chapter 1

Introduction

Encapsulation is one of the fundamental features of object-oriented programming [Pie02]. The internal information of an object is generally only visible to members of an object's class, or in case of Smalltalk, only to the instance itself [BDN+09]. Encapsulation ensures information hiding, which leads to more reliable and better maintainable code [Mic87]. Also, encapsulation ensures preservation of invariants. One way to break encapsulation is by aliasing. Aliasing occurs when multiple references are pointing to a single object [NCP99]. For example, Figure 1.1 illustrates an object representing a university. This object has information about students enrolled at the university which are represented as a list. The list is composed of nodes which themselves contain references to the students. The university does and should not care about the internal representation of the list and should never be able to access the nodes directly. If the university object obtains a reference to a node nevertheless, it could manipulate it and break the invariants of the list.

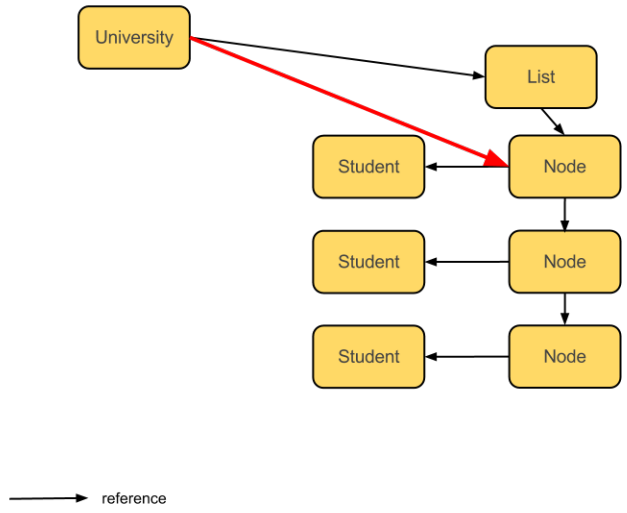

Figure 1.1: An internal node is exposed

An example of an invariant for a list is that the list's size must always be greater or equal to zero. If the university object removes a node directly without going through the list, the list size becomes inconsistent. It could then try to remove a node even when there are no more nodes present, leading to an exception.

1.1 Dynamic Ownership

One way to enforce encapsulation is Dynamic Ownership [GN07]. Dynamic Ownership adds ownership for objects: each object has exactly one owner and no cycles are allowed. These ownership relations form an ownership tree. A singular World object serves as the root of the tree. The objects can only be accessed through their owner. The owner must be used as interface for the objects it owns. All objects with a common owner belong to an encapsulation boundary. Figure 1.2 illustrates such an ownership tree for our initial example. The nodes are contained within the boundary formed by the list while the list itself as well as the students are contained within university's boundary. This means that only the university is able to manipulate the students and the list directly, while the nodes can only be accessed by the list object.

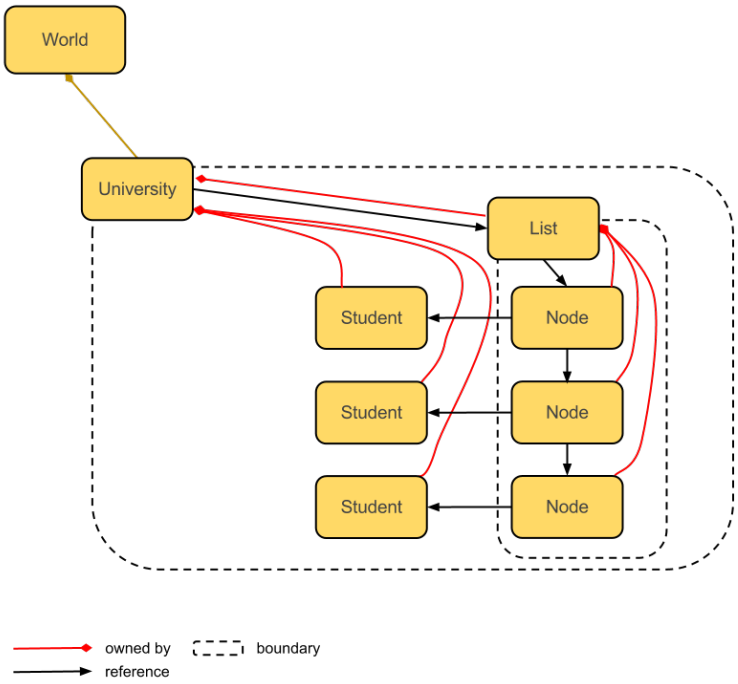

Figure 1.2: Ownership tree and encapsulation boundaries

In Gordon and Noble's Dynamic Ownership [GN07], messages can be sent only if message sends do not cross encapsulation boundaries inward. Messages can be sent to objects that are any links up but at most one link down in the tree. This restriction holds for message sends that do not access or manipulate mutable state of the receiver. Message sends that access or return mutable

state are restricted further to `self` or owned objects. This approach provides an effective way to control aliasing and information hiding, however it is too restrictive. Objects are accessible through their owner but there is no way to fully access objects from somewhere else [WMN12]. For instance, if in Figure 1.2 `University` would use an iterator to traverse the list, a special construct would be necessary with Gordon and Noble's approach [GN07].

We propose a modified version of Dynamic Ownership where we have more control over which methods are exposed to whom, and where we can allow or disallow aliasing in a flexible way. We introduce filters and crossing handlers: Filters define which interface is available to other objects based on the relative position in the ownership tree of the sender and the receiver of a message; Crossing handlers control aliasing by intercepting and possibly manipulating references passing an encapsulation boundary. We implemented a prototype of our proposal in Pharo Smalltalk and adapted a Smalltalk web server to use our version of Dynamic Ownership.

The thesis is organized as follows: Chapter 2 presents our proposal: filters and crossing handlers. Chapter 3 details the implementation of the prototype in Pharo Smalltalk while chapter 4 explains the adaptation of the web server. Chapter 5 discusses the performance overhead of our approach. Finally, we conclude in chapter 6.

Chapter 2

Filters and Crossing Handlers

As in Dynamic Ownership [GN07] we organize our objects in an ownership tree. However, unlike Dynamic Ownership, we make our approach more flexible with filters and crossing handlers. Objects are fully accessible to their owner. Filters define interfaces for other objects than the owner and crossing handlers control aliasing.

2.1 Filters

Filters are used to define the interface of objects confined within an encapsulation boundary. We use in-filters to define which messages can be sent to objects inside a boundary and out-filters to define which messages can be sent from within a boundary. Figure 2.1 illustrates how filters can be defined for our university example. Methods are categorized into topics which must match the filters. Figure 2.2 shows two methods of the `Student` class. One belongs to the `read` topic while the other belongs to the `readWrite` topic. In this example, there is only an in-filter for read-only methods. Therefore objects from outside the boundary are only allowed to access read-only methods of `List` or `Student`s. The list provides no filters, which means that no messages can be sent to the nodes.

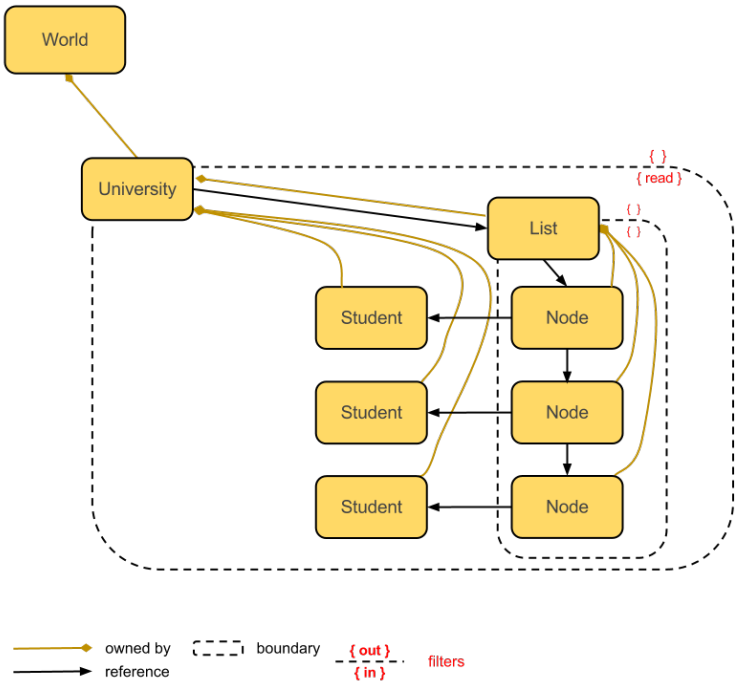

Figure 2.1: Filters on object boundaries

```
Student>>age
<read>
^age


Student>>increaseSemester
<readWrite>
semester = semester + 1
```

Figure 2.2: Methods with different topics

Let us consider a `Statistics` object which is needed to gather various pieces of information about the students registered at the university. It should not be allowed to modify the `Student` objects. Figure 2.3 shows that the `Statistics` object obtained a reference to a student.[1] The in-filter on `University`'s boundary only exposes methods in the `read` topic. This enforces the desired encapsulation. Also note that if `Statistics` would somehow obtain a reference to a node, it would not be able to send any messages to it since no filters are defined on the list's boundary.

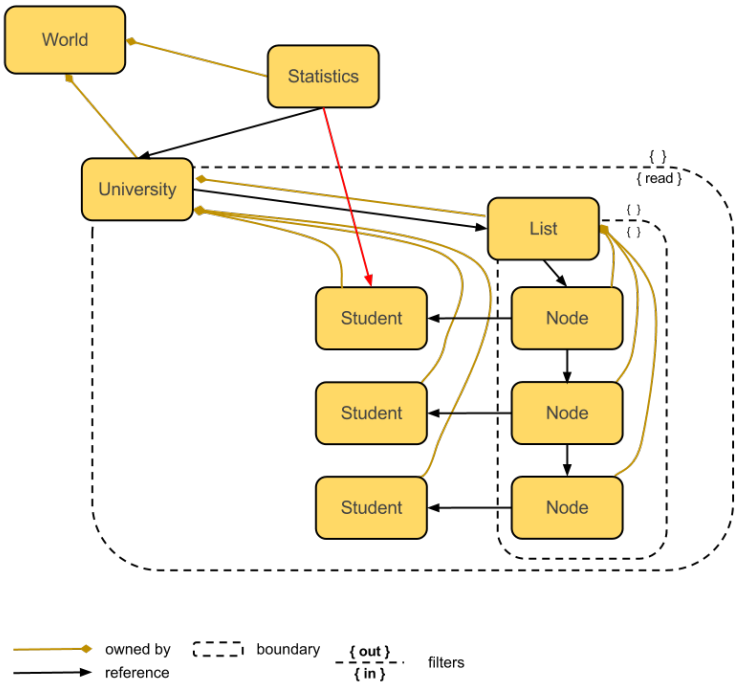

Figure 2.3: `Statistics` object can access `read` methods of `Student` object

[1] By default this would be prohibited by our crossing handler. However they can be specified to allow aliasing as long as methods are exposed.

2.2 Crossing Handlers

We use crossing handlers to control aliasing. Unrestricted aliasing can lead to encapsulation being broken. Whenever an object escapes an encapsulation boundary — meaning there is a reference obtained from outside pointing inside a boundary — the crossing handler is invoked. Figure 2.4 illustrates how `Statistics` obtains a reference to `List`. When the reference to `List` is returned to `Statistics` and escapes the `University` boundary, the crossing handler is invoked.

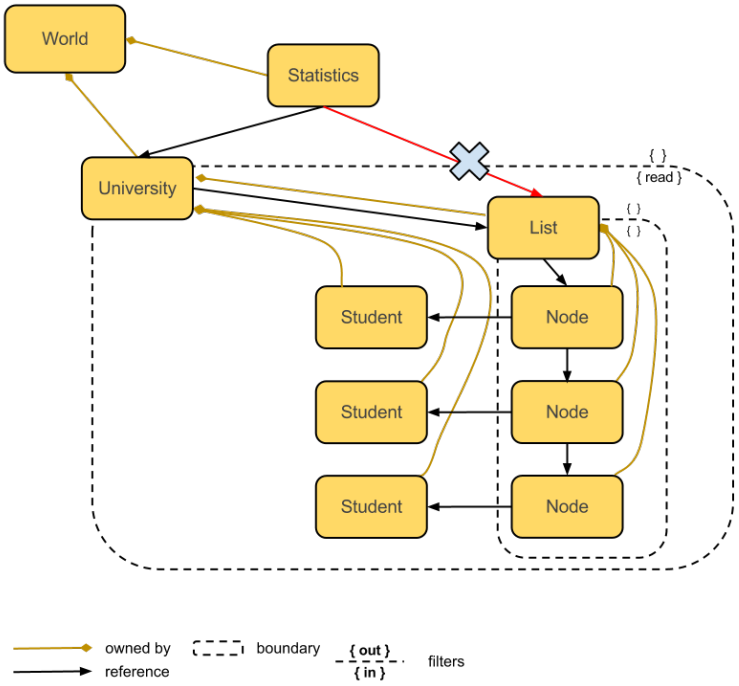

Figure 2.4: An object escapes its encapsulation boundary

By default, the crossing handler would raise an exception. However, it is possible to customize the behavior of the crossing handler. For example, instead of raising an error the crossing handler could create a copy of the list, including the nodes and the students referred to. The owner of this new list would be `Statistics`. This way, `Statistics` could read and manipulate the received copy without compromising the original data. Figure 2.5 illustrates the situation after the crossing handler has returned a copy of the whole structure.

There are many possibilities on how the crossing handlers can be customized. In addition to providing a copy, we can for example prohibit reference leakage altogether, allow references to objects as long as there are methods exposed through the filter mechanism or allow unrestricted aliasing of instances of certain classes.

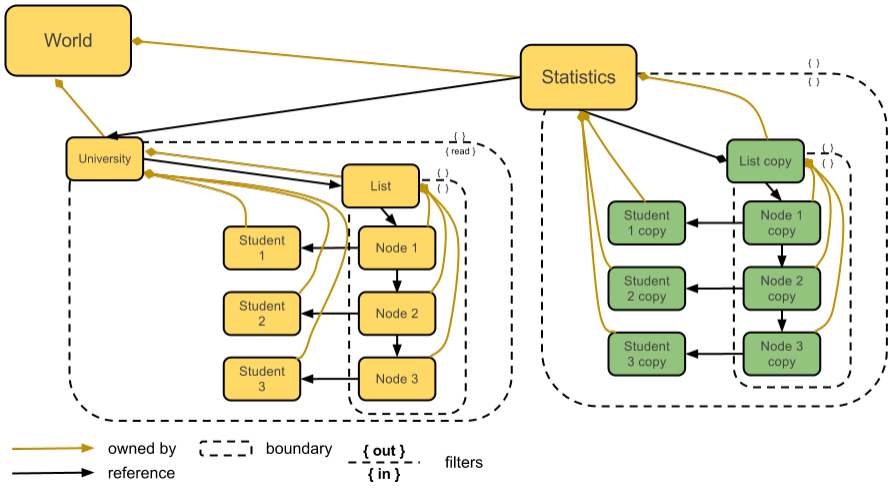

Figure 2.5: Crossing handler returns deep-copy of the escaped object

Chapter 3

Implementation

In this chapter, we discuss the implementation of our model as a prototype in Pharo Smalltalk[1].

3.1 OwnedObject

For our implementation, we need to extend the `Object` class to provide an owner field. However, `Object` is a very integral part in the Smalltalk implementation and most Smalltalk classes inherit from it. `Object` does not have any instance variables and none should be added.[2] Therefore we introduce the class `OwnedObject`, which provides the owner field instead. `OwnedObject` serves as the root class for all classes that are part of an application using our model. It would have been possible to add getter and setter methods for the owner to `Object` and store the owners in a global dictionary, however this approach is slower than our solution.[3]

3.2 Ownership

To implement the ownership tree, we use the owner field provided by `OwnedObject`. When an object is instantiated, it is assigned an owner. By default, the object creating an instance of a class is its owner. Otherwise, if the owner of all instances of a given class is always the same, this can be hard coded in the constructor. The owner can also be an argument of the constructor. Instances of classes that do not inherit from `OwnedObject` are owned by `WorldOwner`, which is a singleton. This includes primitive types like `SmallInteger`.

3.3 Compiler Rewriting

In order to check for the accessibility of methods and invoke the crossing handlers if necessary, each message send needs to be intercepted. Each message

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

[2] As described in `Object`'s class description in Smalltalk

[3] Table B.1 shows the time needed to store or load an object 100'000 times using a field or using a global dictionary.

send is wrapped into a check message, where the required checks are executed. To perform the rewriting of message sends, we use a modified compiler. Every class using our model needs to be compiled using it. We use the RBCompiler[4] which uses the visitor pattern to walk through the abstract syntax tree (AST) and allows us to easily intercept and change message nodes during compilation. For each message node intercepted, a new one is created and the original message is wrapped into it. Essentially, the original message is decomposed into its individual parts and reassembled as part of the new message. This check message will be called on the receiver of the original message and takes the following arguments:

- sender: The sender of the original message
- message: The selector of the original message (i.e. `#name`)
- block: The original message transformed into a block with arguments
- arguments: The arguments of the original message

Figure 3.1 shows how a message send is transformed.

```
anObject name: aName    ->    anObject
                               trap: self
                               messageName: #name
                               block: [:r1 :r2 | r1 name: r2]
                               arguments: {aName}
```

Figure 3.1: Message `name:` is wrapped into `trap:` method

Since it is possible that the arguments of a message contain another message, the rewriting happens recursively. Figure 3.2 shows how a message send with an additional message send in its arguments is transformed.

```
anObject name: anObject getName  ->  anObject
                                      trap: self
                                      message: #name:
                                      block: [:r1 :r2 | r1 name: r2]
                                      arguments: {
                                       anObject
                                       trap: self
                                       message: #getName
                                       block: [:r1 | r1 getName]
                                       arguments: {}
                                      }
```

Figure 3.2: Messages are transformed recursively

The original message node in the AST is then replaced with the new message node. We have to make sure that these newly created message nodes are not

[4]Part of Helvetia: `http://scg.unibe.ch/research/helvetia`

transformed again, which would lead to an infinite recursion. For this we verify if the selector of the message node in question is `#trap:`, and if that is the case, the node is excluded from rewriting. Also, the blocks we create contain a message node themselves, which should not be transformed either. To accomplish that, we created the class `RBOwnershipBlockNode`, which is a subclass of the RBCompiler's class `RBBlockNode`. It is now possible to check if the message node is part of a `RBOwnershipBlockNode` instead of a normal one and exclude it from the transformation if necessary.

Other techniques could be used to implement the check [Duc99]. Changing the compiler to do the rewriting proved to be a relatively simple approach. Instead of transforming the original message into a block, it could be called directly using reflection. Both approaches are comparable in performance. The evaluation of the block itself is faster than sending the message with reflection. However the creation of the block can lead to slower speed.[5] Since we also support reflection, it can be used if it proves to be faster for a particular application.

3.4 Topics

To assign methods to a topic, we use pragmas. Pragmas are annotations used to specify data about a program [BDN+09]. They can be used to tag a method and can be accessed at run-time. Pragmas are defined at the top of a method and consist of a user-defined keyword inside angle brackets. For example, if a method belongs to the readWrite-topic, we tag it with the pragma `<readWrite>`.

3.5 Dynamic Checks

For each message send that needs to be checked, the trap method is invoked on the receiver of the message. The logic consists of four steps:

- Check filters and topics
- Check if arguments escape a boundary
- Send the original message
- Check if return value escapes a boundary

Check Filters and Topics

First the topics of the message are matched against the filters. We need to find the path in the ownership tree from the sender to the receiver and collect all filters along the path. The filters are defined on each class using the methods `inFilters` and `outFilters`. If the topics don't match the filters along the path, an error is raised.

[5] Table B.2 shows the times for 1'000'000 executions of the method `name:aName`, that just returns the argument, using reflection and using a block.

Check If Arguments Escape a Boundary

Next the arguments of the original method are checked. If one of the arguments escapes its boundary, the crossing handler is invoked on the owner of the escaping object. The crossing handler consists of the methods `canCrossBoundary:`, `doCrossBoundary:` and `doNotCrossBoundary`. Each of these methods take the object in question as argument and are invoked on the owner of this object. `canCrossBoundary:` returns a boolean which indicates whether the object is allowed to cross the boundary. If the object can cross the boundary, `doCrossBoundary:` is called, otherwise `doNotCrossBoundary:` is called.

Send the Original Message

If no error is raised so far, the original message is now sent. This step must happen prior to checking for the return value escaping a boundary. As the original message is transformed into a block, this step boils down to the evaluation of the block with the original arguments. In Figure 3.1, the block `[:r | r name]` is now evaluated with the argument `aName`, which is equivalent to sending the original message.

Check If Return Value Escapes a Boundary

After the block is evaluated, the return value is stored in a temporary and we check if the value escapes a boundary when passed to the original sender. Again, the crossing handler is invoked if necessary.

If all checks are passed without raising an error, the return value is now passed to the original sender.

3.6 Defining Filters and Crossing Handlers

The crossing handler can be used to specify the exact behavior when an object escapes its encapsulation boundary. To specify the behavior, the methods `canCrossBoundary:`, `doCrossBoundary:` and `doNotCrossBoundary:` need to be implemented by the developer for each class serving as an encapsulation boundary. Additionally, each class needs to define its `inFilters` and `outFilters`. To let all or no messages pass the special filters `all` or `none` can be used correspondingly

A practical way to define filters and crossing handlers is to use traits. Traits are a collection of methods that are shared between different classes [BDN+09]. It is not necessary to use traits, the methods could be defined on each class separately. However, since multiple classes often share the same behavior, traits are an elegant way to reuse code. Our system comes with the following four predefined traits:

- `TDeepOwnership`: No object from outside the boundary has access
- `TNoOwnership`: No ownership constraints at all

- `TSelectiveOwnership`: Boundary crossing is allowed as long as methods are exposed to the receiver
- `TDefensiveCopyOwnership`: A copy of the object that crosses the boundary is returned instead of the original object

Additional traits can be added or the existing ones can be modified, according to the specific needs of the developer.

3.7 Optimization

The trap method needs to calculate the path between different objects in the ownership tree, which is costly. To improve the performance of the system, the amount of these calculations should be reduced as much as possible. We distinguish between static and dynamic optimizations. If we already know at compile time that messages do not need to be subject to our dynamic checks, we exclude them from our special compilation. This brings the biggest benefit, since it avoids the overhead of the trap method altogether. Furthermore, we introduce dynamic optimizations that use information available only at run time.

3.7.1 Static Optimizations

Literal self (and super) message sends that are defined statically can always be excluded from the dynamic checks. An object is always allowed to send messages or pass references to itself, since it has obtained these references earlier and the check was already executed then. When a self-send is defined statically, we exclude it from rewriting by our compiler. Second, in Smalltalk there is no syntax for control constructs. Instead they are expressed using message sends [BDN+09]. Object comparisons for identity or equality are also realized with message sends. We exclude the most common control flow messages (`ifTrue:`, `ifFalse:`, `whileTrue:`, etc.) as well as equality comparisons (`=`, `==`) from rewriting as well.

3.7.2 Dynamic Optimizations

We can distinguish between five different types of message sends, defined based on the relative position in the ownership tree of the sender and the receiver of a message:

- self
- child to parent
- parent to child
- sibling to sibling
- regular

Figure 3.3 illustrates the different types of messages. Certain parts of the trap method can be skipped based on the type of message send. Literal self sends are already skipped by the compiler. However, a self send can also happen

dynamically which can only be detected at run time. Therefore, self sends are skipped also in the trap method. For the other special cases of message sends, only a part of the trap method needs to be run. For messages from parent to child, we only need to check if the return value escapes the boundary, since an object can always send messages to its child and the arguments cannot escape a boundary; they are only passed into a boundary. Vice versa we only need to check if the arguments cross a boundary for message sends from child to parent. For messages sent from sibling to sibling, checking for filters can be excluded, but boundary crossing still needs to be checked. For all other message sends, all of the checks need to be conducted.

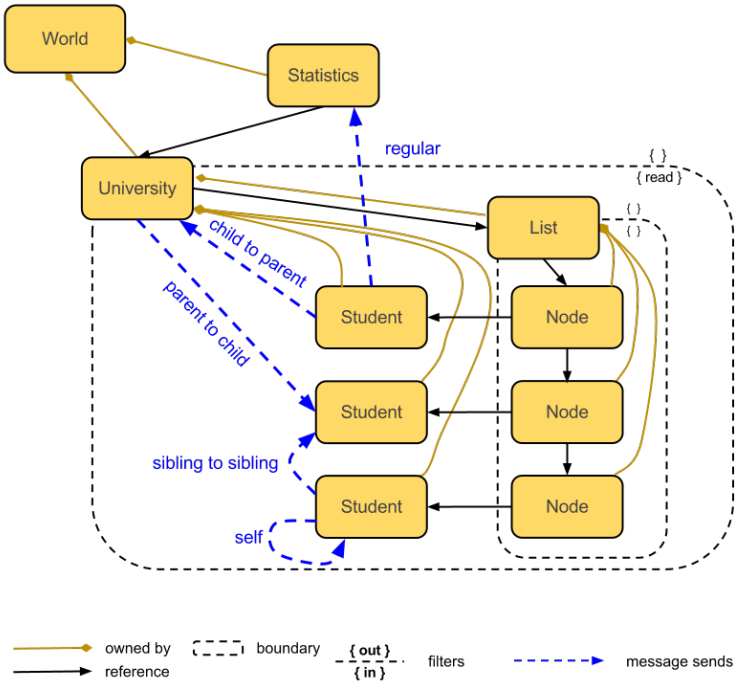

Figure 3.3: The five types of message sends

3.7.3 Caching

When conducting the checks, the path from one object to another in the ownership tree must always be calculated. When there is no ownership transfer, this path stays the same. Therefore we can cache the path from one object to another and do not need to recalculate it every time it is used for a check. This optimization should bring the most benefit when the cached paths are long. Also, when ownership is transferred, the cache must be invalidated since it could be inconsistent. As ownership transfers are usually rare, caching is still beneficial. However, if ownership transfers are used frequently, it might be better to disable the cache.

Chapter 4

Adapting the Web Server

In order to validate our approach, we adapted an existing Smalltalk web server[1], as well as the collection and stream classes it depends upon, to our model. On top of the web server we created a very minimal Wiki-application which allows users to view and edit pages. Since the collection and stream classes are an integral part of Smalltalk itself, we chose to adapt copies of these classes to avoid meta-circularity issues.

4.1 Adapting Web Server Classes

The web server is relatively small. The main classes in charge of handling the HTTP protocol are `WebServer`, `WebRequest` and `WebResponse`. In addition, we implemented the class `WikiHandler` which is used to view and edit `WebPage`s. We used three topics: `request`, `content` and `log`. When the `WikiHandler` needs to process a `WebRequest`, it locates the requested `WebPage` and renders the page. To obtain the necessary information from `WebRequest`, the `request` topic is used. With the `content` topic the `WikiHandler` is able to access nested pages. The last topic is used for logging, which happens on the `WebServer`. The ownership structure is as follows: `WebResponse` is owned by `WebRequest` which is owned by `WebServer`. `WikiHandler` owns the root `WebPage`. When there are nested pages, the sub-`WebPage`s are owned by the parent `WebPage`. [WMN12] Figure 4.1 illustrates the key objects as well as the topics and the ownership structure.

[1] `http://www.squeaksource.com/WebClient.html`

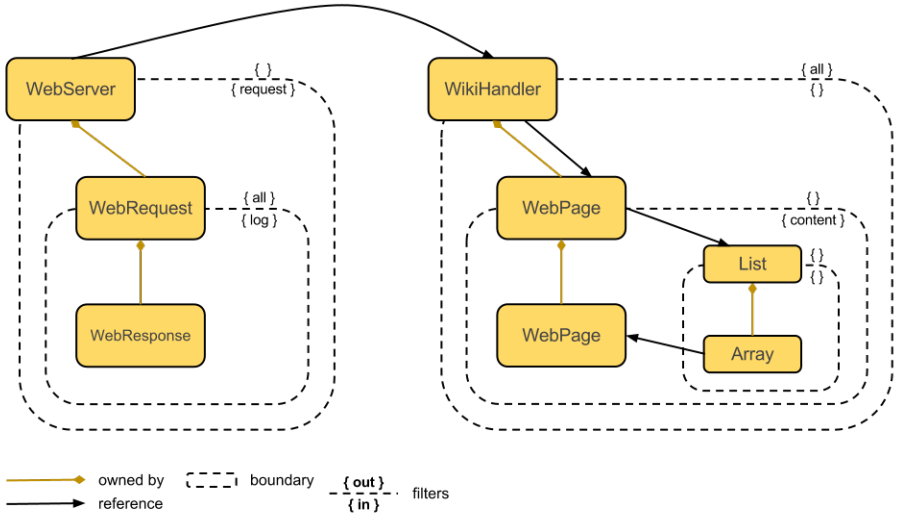

Figure 4.1: Web server objects, filters and the ownership topology [WMN12]

4.2 Adapting Collection and Stream Classes

4.2.1 Identifying Used Classes

We wanted to adapt all classes used by the web server, except primitive types like `SmallInteger`. Therefore, we needed to identify which classes are actually used by the web server. An approach would have been to identify used classes with static code analysis. Unfortunately, this generated too many false positives. Instead it proved easier to identify these classes dynamically using our existing trap mechanism. Since each message sent from already adapted code is intercepted by the trap method, we could modify it to log all classes effectively used.

4.2.2 Copying Part of the Collection and Stream Hierarchy

Once we identified which of the collection and stream classes had to be adapted, we copied these classes as well as their parent classes. This process was tedious since copied classes must be updated to reference each other and not the original classes. Our approach to solve this was as follows: First use a fresh image and rename all desired classes, using the refactoring tools. All references to these classes are updated for the whole image. Then export all the renamed classes from the fresh image and import them into the original image. After that you need to make sure that all already existing code references the renamed versions instead of the original ones. For this we provide a simple script that manually renames all references to the new classes.

4.2.3 Adapting Collection and Stream Classes

Figure 4.2 shows the list of all adapted classes. We discuss in general what considerations were necessary to adapt these classes as well as specific examples.

```
Collection
HashedCollection
Dictionary
IdentityDictionary
Set
IdentitySet
SequenceableCollection
Array
OrderedCollection
Magnitude
LookupKey
Association
Mutex
SocketStream
Stream
PositionableStream
ReadStream
WriteStream
LimitedWriteStream
```

Figure 4.2: List of adapted Smalltalk classes

Constructors

We need to assign an owner when an object is created. In Smalltalk however, there are no constructors. Objects are created by sending the message `new` to a class [WMN12]. When `new` is used we intercept it and assign the sender of `new` as the default owner. However, there are many special factory methods used as constructors which send `new` themselves. If this pattern is used, the class itself will be the owner of the created object, instead of the original sender. Therefore, we have to manually adapt such methods to assign the correct owner. For these methods we introduce a special meta-pragma which is used to bypass the method from our special compilation. We don't want to rewrite the adapted factory-methods, because `new` would be intercepted again and our manual owner assignment would be overwritten.

```
with: anObject
^ self new
add: anObject;
yourself
```

Figure 4.3: Method: `Collection class>>with:`

Figure 4.3 shows such a factory method of the class `Collection`, which is an abstract super-class for all collection classes. This method creates a new collection with the argument as an element. As this is a class-side method, the default owner would be the class instead of the object that sent the message.

```
with: anObject
<meta>
| o |
o := self new.
o ownerNoCheck: self thisSender.
o add: anObject;
yourself.
^o.
```

Figure 4.4: Adapted method: `Collection class>>with:`

Figure 4.4 shows the adapted version of this method. The pragma `<meta>` prevents the rewriting of this method with our compiler. After sending `new` we manually assign the sender as the owner of the newly created collection and add `anObject` afterwards.

There are many such factory methods on the collection and stream classes and we needed to identify and manually adapt each of them individually. Our system lacks a mechanism to handle these cases automatically.

Constructors on Instance Side

There are also methods on the instance side that return a new object. An example is the method `asArray` of `Collection`. This method creates a new `Array` and adds the collections content to it. As the default owner is the creator of an object, the collection would be the owner. However, conceptually the caller of `asArray` should be the owner of the new `Array`.

Another example is `shallowCopy`. This method creates a copy which shares the instance variables of the original object. It is called by `copy` to copy the object itself, afterwards `copy` calls `postCopy` to copy the instance variables. For a class like `OrderedCollection`, using `shallowCopy` means that the internal array is still the same for the copied object, which causes problems with our ownership model. The internal array of the copied object would then still be owned by the original object. We fixed that specific case by creating a new internal array for the copied `OrderedCollection`, but this unfortunately changes the functionality of the method. Also, the caller of `shallowCopy` should be the owner of the copy instead of the instance of `OrderedCollection` that creates the copy. This problem could have been handled by adopting `postCopy`, but because `shallowCopy` is used by methods like `sort`, we decided to handle it in `shallowCopy` itself.

Similar to the class-side constructors, these cases need to be identified and handled manually.

Ownership of Internally Used Objects

Some of the collection classes like `OrderedCollection` use an array to internally store their elements. Likewise streams use an internal collection. We need to

make sure that this internal array or collection is properly owned by the instance of the collection. As long as the initialization of the internal array or collection is done on the instance-side, the correct owner is assigned by default. However, the initialization is often done on the class-side in factory methods, which leads again to the class being assigned as the owner. These cases also need to be handled manually. Following is an overview of the changes:

Array The elements in an array are stored in the instance variables of the object itself [BDN+09]. There is no internal collection object. The elements are owned by their respective owner and not by the array. Since elements are already instantiated prior to being stored in an array, there is no need to handle ownership manually in this case.

OrderedCollection `OrderedCollection` uses an array internally, which is instantiated in the constructors on the class-side. We needed to rewrite the constructors to make sure that the array is instantiated after the `OrderedCollection` instance itself and that the instance is the owner of the array instead of the class. Additionally, `OrderedCollection` provides a mechanism that replaces the internal array with a bigger one when the collection grows. However, this happens on the instance-side, which means that the default owner assignment is already correct for this case.

Dictionary `Dictionary` uses an array internally wherein `Association`s are stored. An `Association` consists of a key and a value. Both, the array as well as all associations, are owned by the `Dictionary`. The key and values, however, are owned by their original owner. Constructors had to be adapted manually for `Dictionary` as well to ensure correct ownership.

Streams Since streams basically serve as a wrapper for collections, it can be argued whether those wrapped collections should be owned by the stream or not. One way would be to transfer ownership of a collection to the stream for the duration of its operation. Alternatively, the stream could gain access to the collection by defining appropriate topics and filters. In our adapted web server this issue did not come up because only `ByteArray`s were used for streaming. We consider `ByteArray` a primitive type which are not subject to the ownership constraints. However, if streams were used for different collections, policies for these cases needed to be defined.

Defining Filters and Crossing Handler

Whenever we adapt or create a new class under our model, we not only need to think about the ownership structure, but also about what filters and what kind of crossing handler we want to use. For the collection classes, we had to use a very strict variant. Since the collections act as a boundary for their internal arrays, no one outside should be able to send messages to them or obtain a reference to the internal array. Hence, `TDeepOwnership` was applied to all collections and streams.

Meta-circularity Issues with printOn

The method `printOn:` caused special problems. It is used internally by Smalltalk, for example in the debugger or when inspecting an object. Using the debugger causes `printOn:` to be called, which in turn calls the trap method, leading to crashes of the debugger. Because of this, we had to exclude `printOn:` from rewriting by our compiler.

4.2.4 Support of Array Literals

Unlike other collection classes, arrays in Smalltalk can be instantiated using a special literal syntax. When arrays are created that way, our system is not able to detect it and a standard array is created instead of an adapted one. Therefore, we had to modify our compiler to also transform this array creation into a creation of an adapted array. This rewriting is a side effect of our decision to copy the classes we wanted to adapt. Figure 4.5 shows an example of such a transformation.

```
indexArray:= {2. 3. 4.}. -> indexArray := DOArray withAll: {2. 3. 4}.
```

Figure 4.5: Transforming array instantiation

4.2.5 Unit Tests

To ensure that the original functionality of the collection and stream classes is unaffected by the adaptation, we also adapted the corresponding unit tests, following the same steps as for the collection classes themselves. Minor modifications of the unit tests were necessary to reflect the changes done to the collection and stream classes during the adaptation. Since the adapted unit tests need to refer our adapted `Array` class, we needed a special compiler for the unit tests that does just the array transformations, without adding the message interception.

4.3 Discussion

With the prototype and the adaption of the web server, we were able to demonstrate the applicability of using our system. The adaptation of collection and stream classes proved to be quite time-consuming. However, since we know now which approach to follow and what to take into account when adapting classes, adapting additional classes would be faster. Our system lacks a mechanism to deal with constructors, on the class as well as on the instance side. Identifying and handling these cases is quite tedious. Going forward, a way to deal with these cases automatically needs to be implemented.

Adapting the web server itself proved the flexibility of our approach with filters and crossing handlers. It was possible to adapt the web server without extensive changes to its structure. Still, you need to invest time into finding the right ownership structure and defining filters and crossing handlers. It should be investigated further how difficult it would be to adapt a bigger code base.

Chapter 5

Validation

In this chapter, we discuss the performance overhead of our approach as well as data on the distribution of message sends of the web server.

5.1 Distribution of Message Sends

In chapter 3.7.2 we discussed different types of message sends and how they can be optimized. Figure 5.1 shows the distribution of message sends to adapted classes for a given test-run[1] of our adapted wiki. Figure 5.2 also includes message sends to primitive types. Those are excluded from the dynamic checks.

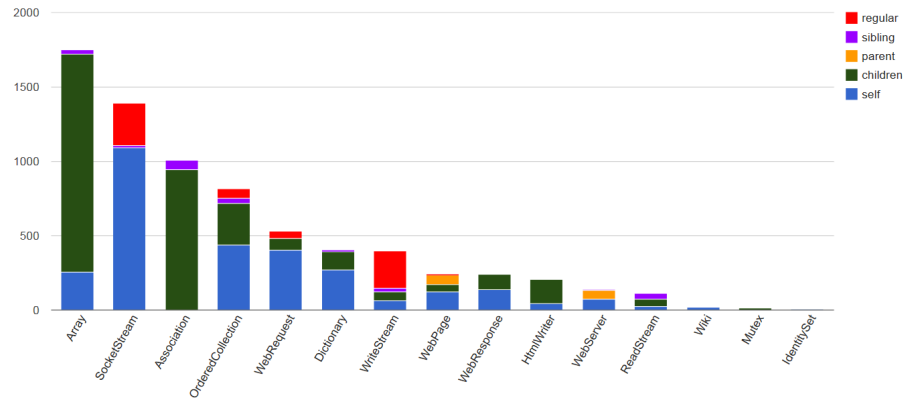

Figure 5.1: Distribution of message sends to adapted classes

Most of the message sends that are subject to dynamic checks belong to the types that can be optimized. We can also infer the topology of the ownership tree from this data. For example, we can see that instances of `Array` receive most of the messages from their parents. In general, this distribution of message sends can show us how well the design of an application matches with the chosen ownership topology. If we have a good object-oriented design, most of the

[1] View root page, edit root page, view first page, view second page, edit second page, view root page

messages will likely be sent from objects close to each other in the ownership tree. This will also lead to better performance, since these message sends are optimized.

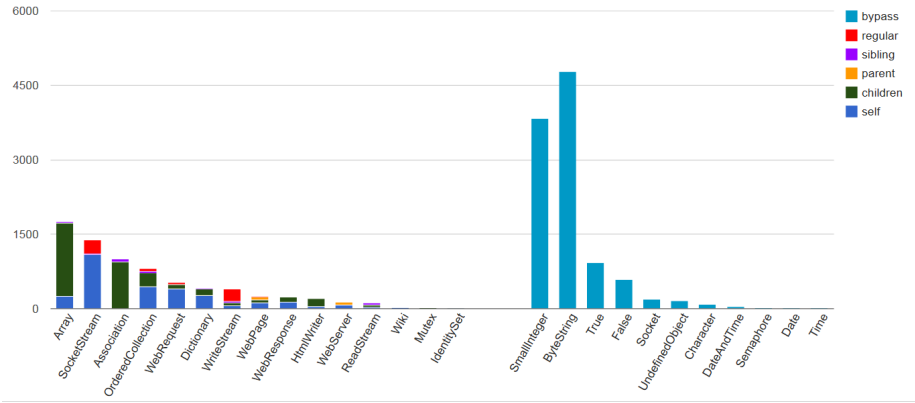

Figure 5.2: Distribution of message sends to adapted classes and primitive types

5.1.1 Design Changes the Distribution of Message Sends

A small change in the design of the application has an impact on the distribution of message sends. Here, we report on the impact of a change in the rendering of web pages. The web page is rendered using a `WriteStream`. This stream is owned by the `WikiHandler`. Originally, the `WikiHandler` sends messages `renderX: aStream` to the `WebPage` that has to be rendered. The methods `renderX: aStream` create a `HtmlWriter`, which is owned by the `WebPage`. The `WriteStream` and the `HtmlWriter` exchange messages. In the refactored version, we moved the `renderX: aStream` methods to the `WikiHandler`. The created `HtmlWriter` is now owned by `WikiHandler`. Figure 5.3 illustrates an excerpt of the ownership trees for these two cases. Figure 5.4 shows the distribution of message sends for the refactored version of the web server. When compared to the results from the original version shown in Figure 5.1, we can see that the majority of message sends to `WriteStream` are now sibling to sibling instead of regular message sends. However, some message sends to `WebPage` are now regular ones, although a smaller number.

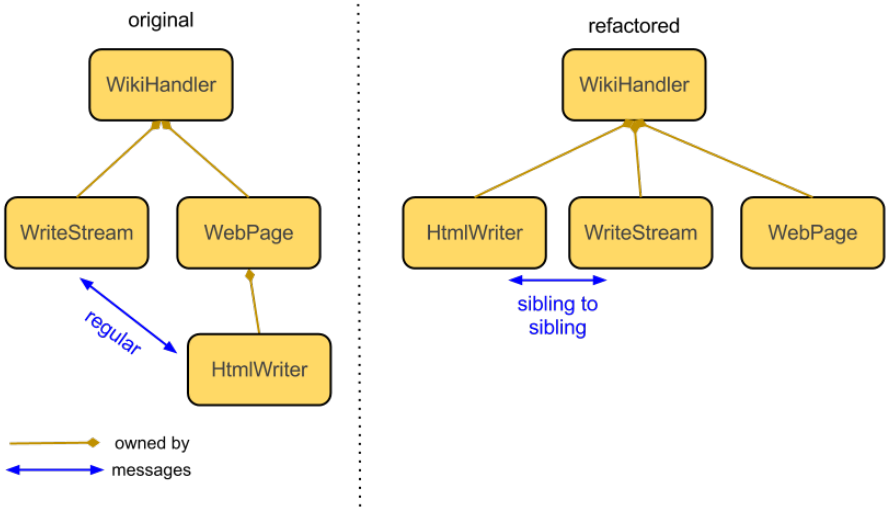

Figure 5.3: Excerpt of ownership tree for original and refactored Wiki

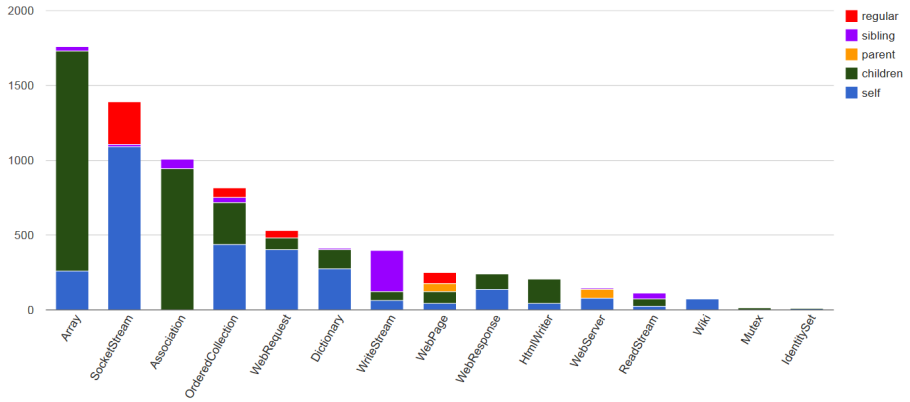

Figure 5.4: Distribution of message sends to adapted classes

5.2 Benchmarks

To assess the performance of our prototype, we conducted micro-benchmarks as well as benchmarks of the wiki. Here we report on the results.

5.2.1 Micro-Benchmarks

Table 5.1 shows the times for 10'000 executions of the method `returnParameter:42` invoked on itself, a child, a parent and a sibling or an other object with and without the caching described in chapter 3.7.3. The method takes a parameter and returns it without doing anything else.

Table 5.1: Micro benchmark for different types of message sends

message type	time without caching	time with caching
self (literal)	3 ms	3 ms
self (dynamic)	25 ms	25 ms
parent to children	28 ms	28 ms
children to parent	41 ms	41 ms
sibling to sibling	47 ms	48 ms
regular	390 ms	87 ms

The literal self message send is the time needed for the method to run without any checks, as literal self message sends are not transformed by our compiler. The dynamic self message send, however, shows the time needed to run the method including the trap method. There are no additional checks executed for self-sends since they are optimized in the trap method. We can see that the execution through the trap method without doing additional checks takes eight times longer than the normal execution of the method. We can also see that regular message sends take around ten times longer than optimizable ones. Because the expensive part in the checks is the calculation of the path in the ownership tree, caching helps reduce that overhead significantly for regular message sends. Since the path is very short for the other types of message sends, the performance of those is virtually unaffected by the caching.

5.2.2 Wiki Benchmarks

To be able to run consistent benchmarks, we needed a different solution than running the web server and measuring the time for requests, since this was too unreliable and the results were impacted by IO-operations. Thus we created a class `MockSocket` that mimics the real socket used by the web server. This `MockSocket` simulates multiple identical HTTP-requests. Table 5.2 shows the time needed for the execution of 1000 such read requests sent to the `WebServer`.

We already measured a significant overhead of one order of magnitude with the micro-benchmarks. This overhead is confirmed by the wiki benchmarks, which is a more realistic application.

Table 5.2: Time for execution of 1000 read requests on the wiki

	non refactored	refactored
Full ownership	202.8 ms	210 ms
No dynamic optimizations	285.8 ms	282.2 ms
No caching	195.6 ms	211,8 ms
No ownership	21 ms	25 ms

Dynamic Optimizations

The dynamic optimizations based on the type of message send we discussed in chapter 3.7.2 increase the performance by around 25 percent.

Caching

Unfortunately, the caching mechanism discussed in chapter 3.7.3 results in a slowdown for this benchmark. We can observe 1643 cache hits and 1161 cache misses. With this ratio, the caching mechanism entails an overhead instead of a speedup. One reason for this is the particular message distribution. Figure 5.5 shows the distribution of message sends for one execution of this benchmark. Figure 5.6 shows the sum for each type of message sends. As we can see, the vast majority of message sends are of type self and children, a smaller part of type sibling. Regular message sends account only for a very small part of the total. However, the caching mechanism brings the most benefit if the paths from sender to receiver in the ownership tree are long. Also the benefit of the caching is dependent on where checked message arguments and return values are positioned in the ownership tree. In the end, whether caching should be turned on or not has to be assessed on a case-by-case basis.

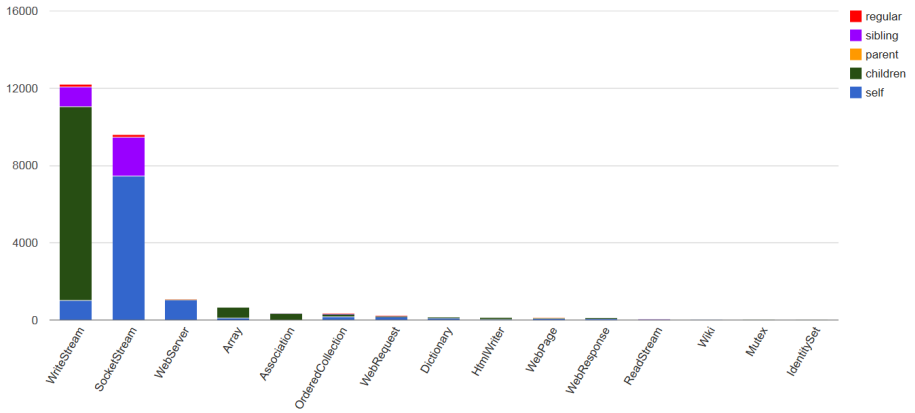

Figure 5.5: Distribution of message sends for wiki benchmark

Refactored Version

Since a portion of regular message sends changes to the type sibling to sibling in the refactored version of the wiki, we could expect a performance gain.

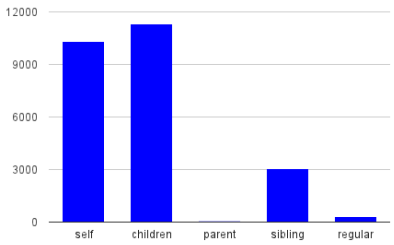

Figure 5.6: Aggregated distribution of message sends for wiki benchmark

Figure 5.7 shows the aggregated numbers of message sends of each type for the same benchmark in the refactored version. When compared to Figure 5.6, we can observe the small shift from regular to sibling. However, this shift corresponds only to a very small fraction of the overall message sends. Since the benchmark without ownership shows that the refactored version is a bit slower than the original one for reasons unrelated to ownership, it seems that the smaller amount of regular messages can't make up for the initial speed loss.

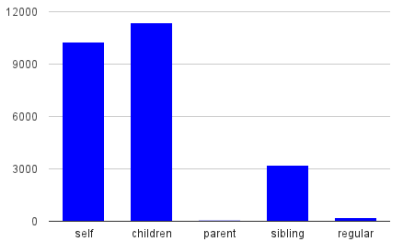

Figure 5.7: Aggregated distribution of message sends for refactored wiki benchmark

5.3 Discussion

The distribution of message sends is a measure of the quality of an application's design with regard to encapsulation. The vast majority of message sends in our web server are to self and to children, regular message sends account only for a very small amount of the total.

The performance overhead of our prototype is significant. We could achieve some improvements with our optimizations, but not all of our approaches proved beneficial. It remains to be seen how much the performance could be improved further, or if the overhead remains too big for a deployed application. If that is the case, the system could only be used in a development environment and could be turned off when an application is deployed.

With the adaption of the web server we showed it is possible to retrofit dynamic ownership to an existing system. The strength of our system lies in supporting the development of new applications or the extension of existing ones. We believe that the constraints introduced by our system and the effort required to design applications accordingly will lead to a better design, which is less susceptible to bugs. In that sense, if the web server application were to be extended, it would benefit from using our ownership system.

Chapter 6

Conclusions

Our variant of dynamic ownership allows the developer to tailor it to his specific needs through the flexibility provided by filters and crossing handlers. The filters allow for a much more fine-grained encapsulation than just to hide an object or not. It is possible to specify exactly which methods are exposed to the outside and which are hidden based on the topology of the ownership tree. Crossing handlers provide even more flexibility.

Using the system forces the developer to think about encapsulation which, we believe, is leading to a better design. However, the high flexibility also leads to a bigger complexity. For bigger applications, it can be quite an undertaking to adapt all classes, define the ownership topology, determine the topics and filters and decide what kind of crossing handlers to use.

By adapting the web server, we proved that our system is working with a real-world application. Following our approach, additional classes could be adapted without a substantial effort. However certain cases, like the constructors, still need to be handled manually. Our system would benefit from an automatic solution in this area.

Even with optimizations, the performance overhead is still too significant for a deployed application. Either the performance has to be improved with better caches and additional optimizations, or the system should only be used during development.

Overall, our system provides the benefits of traditional Dynamic Ownership systems while offering more flexibility, but compared to not using an ownership system at all, it entails the drawbacks of added complexity and a performance overhead.

Appendix A

Installation Guide

1. Get Pharo 1.3: `http://www.pharo-project.org/pharo-download/release-1-3`
2. If your unfamiliar with adding and loading repositories with Monticello, read Chapter 6.3 from PharoByExample [BDN+09]
3. Open the Monticello Browser and add the following HTTP repositories (Figure A.1):

```
MCHttpRepository
    location: 'http://source.lukas-renggli.ch/helvetia'
    user: ''
    password: ''

MCHttpRepository
    location: 'http://ss3.gemstone.com/ss/DynOwn'
    user: ''
    password: ''
```

4. Load AST-Compiler package from Helvetia (Figure A.2)
5. Load DynamicOwnership package from DynOwn (Figure A.2)
6. Recompile all DynamicOwnership classes using `ClassesUtils recompileAll`
7. Execute `WorldOwner resetAll`
8. Start the wiki with `Wiki new buildSampleWiki ; run`
9. You can now access the wiki in your web browser with the URL `http://localhost:9999/example?action=view`

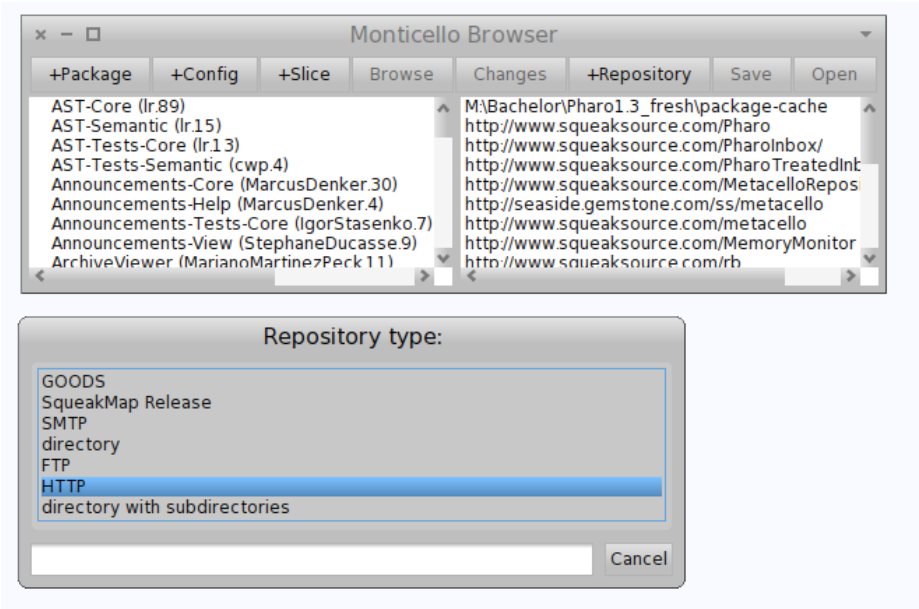

Figure A.1: Monticello browser

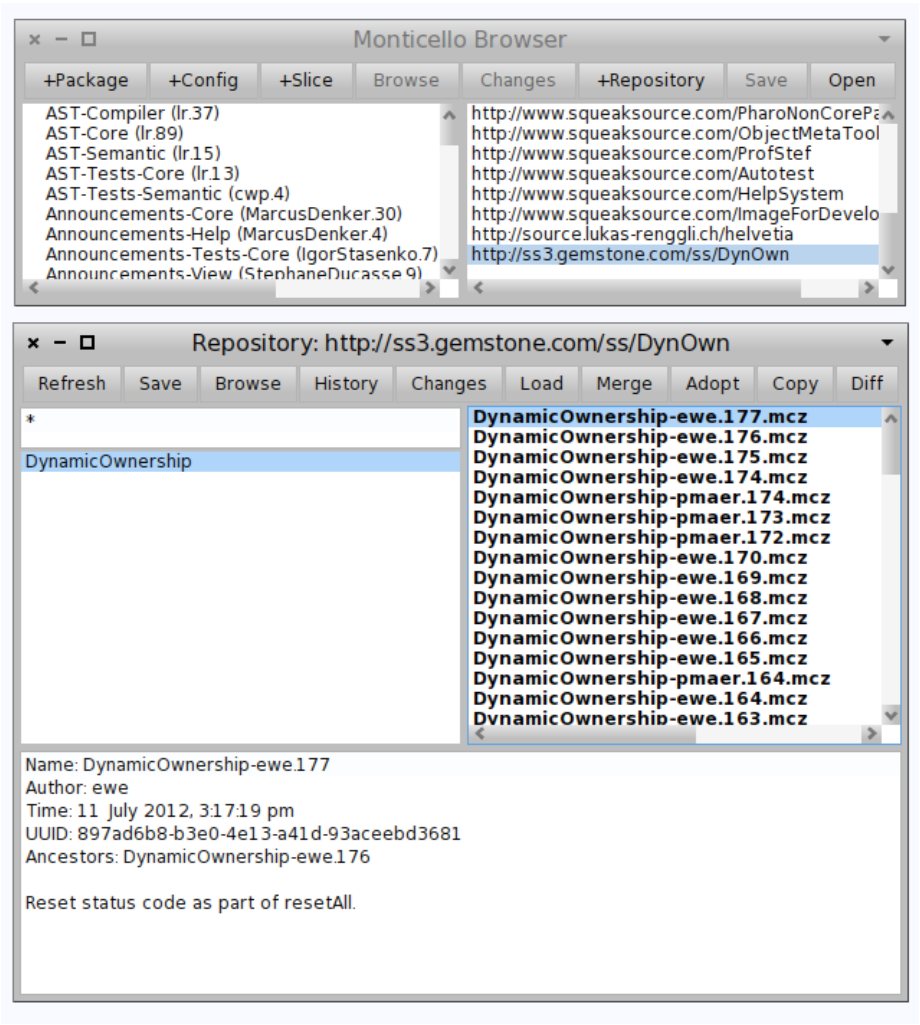

Figure A.2: Loading a package

Appendix B

Benchmarks

Table B.1: 100'000 store or load operations using a field or a global dictionary

	Field	Global Dictionary
Store	4 ms	48 ms
Load	2 ms	39 ms

Table B.2: 1'000'000 executions of `name:aName` using reflection or using a block

Reflection	131 ms
Block (evaluation)	96 ms
Block (creation and evaluation)	184 ms

List of Tables

List of Figures

Bibliography

[BDN+09] Black, Andrew ; Ducasse, Stéphane ; Nierstrasz, Oscar ; Pollet, Damien ; Cassou, Damien ; Denker, Marcus: *Pharo by Example*. Square Bracket Associates, 2009 `http://pharobyexample.org`. – ISBN 978–3–9523341–4–0

[Duc99] Ducasse, Stéphane: Evaluating Message Passing Control Techniques in Smalltalk. In: *Journal of Object-Oriented Programming (JOOP* 12 (1999)

[GN07] Gordon, Donald ; Noble, James: Dynamic ownership in a dynamic language. In: *Proceedings of the 2007 symposium on Dynamic languages*. New York, NY, USA : ACM, 2007 (DLS '07). – ISBN 978–1–59593–868–8, 41–52

[Mic87] Micallef, Josephine: Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages / Department of Computer Science, Columbia University. 1987. – Forschungsbericht

[NCP99] Noble, James ; Clarke, David ; Potter, John: Object Ownership for Dynamic Alias Protection. In: *Proceedings of the 32nd International Conference on Technology of Object-Oriented Languages*. Washington, DC, USA : IEEE Computer Society, 1999 (TOOLS '99). – ISBN 0–7695–0462–0, 176–

[Pie02] Pierce, Benjamin C.: *Types and programming languages*. Cambridge, MA, USA : MIT Press, 2002. – ISBN 0–262–16209–1

[WMN12] Wernli, Erwann ; Maerki, Pascal ; Nierstrasz, Oscar: Ownership, filters and crossing handlers: flexible ownership in dynamic languages. In: *Proceedings of the 8th symposium on Dynamic languages*. New York, NY, USA : ACM, 2012 (DLS '12). – ISBN 978–1–4503–1564–7, 83–94