IWST 2011

Proceedings of the 3rd edition of
the International Workshop on Smalltalk
Technologies

In conjunction with the
19th International Smalltalk Joint Conference

Edinburgh, august 2011

Program Committee

Table of Contents

Clustered Serialization with Fuel

Martín Dias[1][3] Mariano Martinez Peck[1][2] Stéphane Ducasse[1] Gabriela Arévalo[4][5]

[1]RMoD Project-Team, Inria Lille–Nord Europe / Université de Lille 1
[2]Ecole des Mines de Douai, [3]Universidad de Buenos Aires, [4]Universidad Abierta Interamericana [5]CONICET
{tinchodias, marianopeck, gabriela.b.arevalo}@gmail.com, stephane.ducasse@inria.fr

Abstract

Serializing object graphs is an important activity since objects should be stored and reloaded on different environments. There is a plethora of frameworks to serialize objects based on recursive parsing of the object graphs. However such approaches are often too slow. Most approaches are limited in their provided features. For example, several serializers do not support class shape changes, global references, transient references or hooks to execute something before or after being stored or loaded. Moreover, to be faster, some serializers are not written taking into account the object-oriented paradigm and they are sometimes even implemented in the Virtual Machine hampering code portability. VM-based serializers such as ImageSegment are difficult to understand, maintain, and fix. For the final user, it means a serializer which is difficult to customize, adapt or extend to his own needs.

In this paper we present a general purpose object graph serializer based on a pickling format and algorithm. We implement and validate this approach in the Pharo Smalltalk environment. We demonstrate that we can build a really fast serializer without specific VM support, with a clean object-oriented design, and providing most possible required features for a serializer. We show that our approach is faster that traditional serializers and compare favorably with ImageSegment as soon as serialized objects are not in isolation.

Keywords Object-Oriented Programming and Design » Serializer » Object Graphs » Pickle Format » Smalltalk

1. Introduction

In the Object-Oriented Programming paradigm, since objects point to other objects, the runtime memory is an object graph. This graph of objects lives while the system is running and dies when the system is shutdown. However, sometimes it is necessary, for example, to backup a graph of objects into a non volatile memory to load it back when necessary, or to export it so that the objects can be loaded in a different system. The same happens when doing migrations or when communicating with different systems.

Approaches and tools to export object graphs are needed. An approach must scale to large object graphs as well as be efficient. However, most of the existing solutions do not solve this last issue properly. This is usually because there is a trade-off between speed and other quality attributes such as readability/independence from the encoding. For example, exporting to XML [17] or JSON [9] is more readable than exporting to a binary format, since one can open it and edit it with any text editor. But a good binary format is faster than a text based serializer when reading and writing. Some serializers like *pickle* [14] in Python or *Google Protocol Buffers* [15] that let one choose between text and binary representation. Five main points shape the space of object serializers

1. Serializer *speed* is an important aspect since it enables more extreme scenarios such as saving objects to disk and loading them only on demand at the exact moment of their execution [3, 10].
2. Serializer *portability and customization*. Since many approaches are often too slow, Breg and Polychronopoulos advocate that object serialization should be done at the virtual machine level [4]. However, this implies non portability of the virtual machine and difficult maintenance. In addition, moving behavior to the VM level usually means that the serializer is not easy to customize or extend.
3. Another usual problem is *class* changes. For example, the class of a saved object can be changed after the object is saved. At writing time, the serializer should store all the necessary information related to class shape to deal with these changes. At load time, objects must be updated in case it is required. Many object serializers are limited regarding this aspect. For example, the Java Serializer [8] supports adding or removing a method or a field, but does not support the modification of an object's hierarchy nor the removing of the implementation of the Serializable interface.
4. *Loading policies*. Ungar [18] claims that the most important and complicated problem is not to detect the sub-

graph to export, but to detect the implicit information of the subgraph that is necessary to correctly load back the exported subgraph in another system. Examples of such information are (1) whether to export an actual value or a counterfactual initial value, or (2) whether to create a new object in the new system or to refer to an existing one. In addition, it may be necessary that certain objects run some specific code once they are loaded in a new system.

5. *Uniformity of the solution.* Serializers are often limited to certain kind of objects they save. For example MsgPack only serializes booleans, integers, floats, strings, arrays and dictionaries. Now in dynamic programming languages *e.g.,* Smalltalk, methods and classes as first class objects – user's code is represented as objects. Similarly the execution stack and closures are objects. The natural question is if we can use serializers as code management system underlying mechanism. VisualWorks Smalltalk introduced a pickle format to save and load code called Parcels [12]. However, such infrastructure is more suitable for managing code than a general purpose object graph serializer.

This paper presents Fuel, an open-source general purpose framework to serialize and deserialize object graphs using a pickle format which clusters similar objects. We show in detailed benchmarks that we have the best performance in most of the scenarios we are interested in. For example, with a large binary tree as sample, Fuel is 16 times faster loading than its competitor SmartRefStream, and 7 times faster writing. We have implemented and validated this approach in the Pharo Smalltalk Environment [2].

The pickle format presented in this paper is similar to the one of Parcels [12]. However, Fuel is not focused in code loading and is highly customizable to cope with different objects. In addition, this article demonstrates the speed improvements gained in comparison to traditional approaches. We demonstrate that we can build a fast serializer without specific VM support, with a clean object-oriented design, and providing most possible required features for a serializer.

The main contributions of the paper are:

1. Description of our pickle format and algorithm.
2. Description of the key implementation points.
3. Evaluation of the speed characteristics.
4. Comparison of speed improvements with other serializers.

The remainder of the paper is structured as follows: Section 2 provides a small glossary for the terms we use in the paper. Section 3 enumerates common uses of serialization. Features that serializers should support are stressed in Section 4. In Section 5 we present our solution and an example of a simple serialization which illustrates the pickling format. Fuel key characteristics are explained in details in Section 6. A large amount of benchmarks are provided in Section 8. Finally, we discuss related work in Section 9 and we conclude in Section 10.

2. Glossary

To avoid confusion, we define a *glossary* of terms used in this paper.

Serializing. It is the process of converting the whole object graph into a sequence of bytes. We consider the words *pickling* and *marshalling* as synonyms.

Materializing. It is the inverse process of serializing, *i.e.,* regenerate the object graph from a sequence of byes. We consider the words *deserialize*, *unmarshalling* and *unpickling* as synonyms.

Object Graph Serialization. We understand the same for *object serialization*, *object graph serialization* and *object subgraph serialization*. An object can be seen as a subgraph because of its pointers to other objects. At the same time, everything is a subgraph if we consider the whole memory as a large graph.

Serializer. We talk about serializer as a tool performing both operations: serializing and materializing.

3. Serializer Possible Uses

We will mention some possible uses for an object serializer.

Persistency and object swapping. Object serializers can be used *directly* as a light persistency mechanism. The user can serialize a subgraph and write it to disk or secondary memory [11]. After, when needed, it can be materialized back into primary memory [3, 10]. This approach does not cover all the functionalities that a database provides but it can be a good enough solution for small and medium size applications. Besides this, databases normally need to serialize objects to write them to disk [5].

Remote Objects. In any case of remote objects, *e.g.,* remote method invocation and distributed systems [1, 6, 19], objects need to be passed around the network. Therefore, objects need to be serialized before sending them by the network and materialized when they arrive to destination.

Web Frameworks. Today's web applications need to store state in the HTTP sessions and move information between the client and the server. To achieve that web frameworks usually use a serializer.

Version Control System. Some dynamic programming languages *e.g.,* Smalltalk consider classes and methods as first-class objects. Hence, a version control system for Smalltalk deals directly with objects, not files as in other programming languages. In this case, such tool needs to serialize and materialize objects. This is the main purpose behind Parcels [12].

Loading methods without compilation. Since classes and methods are objects, a good serializer should be enable the loading of code without the presence of a compiler. This enable binary deployment, minimal kernel, and faster loading time.

4. Serializer Features

In this section we enumerate elements to analyze a serializer. We start with more abstract concerns and then follow with more concrete challenges. We use these criteria to discuss and evaluate our solution in Section 7.

4.1 Serializer concerns

Below we list general aspects to analyze on a serializer.

Performance. In almost every software component, time and space efficiency is a wish or sometimes even a requirement. It does become a need when the serialization or materialization is frequent or when working with large graphs. We can measure both speed and memory usage, either serializing and materializing, as well as the size of the result stream. We should also take into account the initialization time, which is important when doing frequent small serializations.

Completeness. It refers to what kind of objects can the serializer handle. It is clear that it does not make sense to transport instances of some classes, like FileStream or Socket. Nevertheless, serializers often have limitations that restrict use cases. For example, an apparently simple object like a SortedCollection usually represents a problematic graph to store: it references a block closure which refers to a method context, and most serializers does not support transporting them, often due to portability reasons.

Besides, in comparison with other popular environments, the object graph that one can serialize in Smalltalk is much more complex. This is because it reifies elements like the metalevel model, methods, block closures, and even the execution stack. Normally there are two approaches: for *code* sharing, and for *plain objects* sharing.

Portability. There are two aspects related to portability. One is related to the ability to use the same serializer in different dialects of the same language. For example, in Smalltalk one would like to use the same, or at least almost the same, code for different dialects. The second aspect is related to the ability of be able to materialize in a dialect or language a stream which was serialized in another language. This aspect brings even more problems and challenges to the first one.

As every language and environment has its own particularities, there is a trade-off between portability and completeness. Float and BlockClosure instances often have incompatibility problems.

For example, Action Message Format (AMF), Google Protocol Buffers, Oracle Coherence*Web, Hessian, are quite generic and there are implementations in several languages. In contrast, SmartRefStream in Squeak and Pharo, and pickle [14] in Python are not designed with this goal in mind. They just work in the language they were defined.

Abstraction capacity. Although it is always possible to store an object, concretely serializing its variables, such a low-level representation, is not acceptable in some cases. For example, an OrderedCollection can be stored either as its internal structure or just as its sequence of elements. The former is better for an accurate reconstruction of the graph. The second is much more robust in the sense of changes to the implementation of OrderedCollection. Both alternatives are valid, depending on the use case.

Security. Materializing from an untrusted stream is a possible security problem in the image. When loading a graph, some kind of dangerous object can enter to the environment. The user may want to control in some way what is being materialized.

Atomicity. We have this concern expressed in two parts: for saving and for loading. As we know, the image is full of mutable objects *i.e.,* that changes their state over the time. So, while serialization process is running. It is desired that when serializing such mutable graph it is written an atomic snapshot of it, and not a potential inconsistent one. On the other hand, when loading from a broken stream and so it can not complete its process. In such case, no secondary effects should affect the environment. For example, there can be an error in the middle of the materialization which means that certain objects have been already materialized.

Versatility. Let us assume a class is referenced from the graph to serialize. Sometimes we may be interested in storing just the name of the class because we know it will be present when materializing the graph. However, sometimes we want to really store the class with full detail, including its method dictionary, methods, class variables, etc. When serializing a package, we are interested in a mixture of both: for external classes, just the name, for the internal ones, full detail.

This means that given an object graph living in the image, there is not an unique way of serializing it. A serializer may offer dynamic or static mechanisms to the user to customize this behavior.

4.2 Serializer challenges

The following is a list of concrete issues and features we consider in serializers:

Maintaining identity. When serializing an object we actually serialize an object graph. However, we usually do not want to store the whole transitive closure of references of the object. We know (or we hope) that some objects will be present when loading, so we want just to store *external references*, *i.e.,* store the necessary information to be able to look those objects in the environment while materializing.

In Figure 1 we have an example of a method, *i.e.,* a CompiledMethod instance and its subgraph of references. Suppose that the idea is to store just the method. We know that the class and the global binding will be present on loading. Therefore, we just reference them by encoding their names.

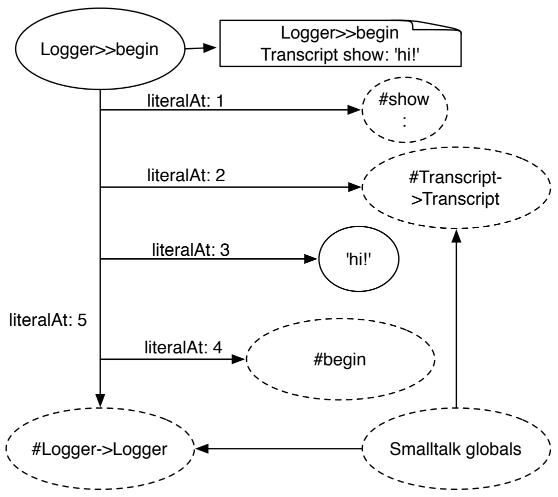


Figure 1. Serializing a method while maintaining identity of its referenced classes and globals.

In other words, in this example we want to serialize a method, maintaining identity of its class, the global Transcript, and the symbols. Consequently, when materializing, the only instances that should be created are the compiled method and the string. The rest will be looked up in the environment.

Transient values. Sometimes objects have temporal state that we do not want to store, and we want an initial value when loading. A typical case is serializing an object that has an instance variable with a lazy-initialized value. Suppose we prefer not to store the actual value. In this sense, declaring a variable as *transient* is a way of delimiting the graph to serialize. We are breaking the iteration through that reference.

There are different levels of transient values:

Instance. When only one particular object is transient. All objects in the graph that are referencing to such object will be serialized with a nil in their instance variable that points to the transient object.

Class. Imagine we can define that a certain class is transient in which case all its instances are considered transient.

Instance variable names. This is the most common case. The user can define that certain instance variables of a class have to be transient. That means that all instances of such class will consider those instance variables as transient.

List of objects. The ability to consider an object to be transient only if it is found in a specific list of objects. The user should be able to add and remove elements from that list.

Cyclic object graphs and duplicates. Since commonly the object graph to serialize has cycles, it is important to detect them and take care to preserve objects identity. Supporting that means decreasing the performance and increasing the memory usage. This is because each iterated object in the graph should be temporally collected: it is necessary to check whether each object has been already processed or not.

Class shape change tolerance. Often we need to load instances of a class in an environment where its definition has changed. The expected behavior may be to adapt the old-shaped instances automatically when possible. We can see some examples of this in Figure 2. For instance variable position change, the adaptation is straightforward. For example, version v2 of Point changes the order between the instance variables x and y. For the variable addition, an easy solution is to fill with nil. Version v3 adds instance variable distanceToZero. If the serializer also lets one to write custom messages to be sent by the serializer once the materialization is finished, the user can benefit from this hook to initialize the new instance variables to something different that nil.

In contrast to the previous examples, for variable renaming, the user must specify what to do. This can be done via hook methods, or more dynamically, via materialization settings.

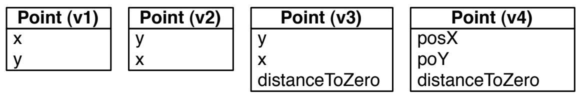


Figure 2. Several kinds of class shape changing.

There are even more kinds of changes such as adding, removing or renaming a method, a class or an instance variable, changing the superclass, etc. As far as we know, no serializer fully manage all these kinds of change. Actually, most of them have a limited number of supported change types. For example, the Java Serializer [8] supports the adding and the removing of a method or of a field, but does not support changing an object's hierarchy or removing the implementation of the Serializable interface.

Custom reading. When working with large graphs or when there is a large number of stored streams, it makes sense to read the serialized bytes in customized ways, not necessarily materializing all the objects as we usually do. For example, if there are methods written in the streams, we may want to look for references to certain message selectors. Maybe count how many instances of certain class we have stored. Maybe list the classes or packages defined or referenced from a stream. Or to extract any kind of statistics about the stored objects.

Partial loading. In some scenarios, especially when working with large graphs, it may be necessary to materialize only a part of the graph from the stream instead of the whole graph. Therefore, it is a good feature to simply get a subgraph with some holes filled with nil. In addition, the tool could support some kind of lazy loading.

Versioning. The user may need to load an object graph stored with a different version of the serializer. Usually this feature allows version checking so that future versions can detect that a stream was stored using another version, and act consequently: when possible migrate it, otherwise throw an error message.

Alternative output format. Textual or binary: serializers like *pickle* [14] in Python or Google Protocol Buffers [15] let the user choose between textual and binary representation. While developing, we can use the text based one, which is easy to see, inspect and modify. Then, at production time, we can switch to the faster binary format.

Alternative stream libraries. Usually, there are several packages of streams available for the same programming languages. For example, for Pharo Smalltalk there are Xtreams, FileSystem, and Nile. A design that supports alternative implementations is desired.

Graphical progress update. Object graphs can be huge and so, the user has to wait until the process end. Therefore, it is important to have the possibility to enable this feature and show graphically the processing of the graph.

5. Fuel

In this section we present Fuel, a new framework to serialize objects. Fuel is based on the hypothesis that fast loading is always preferable, even it implies a slower serialization process. Fuel clusters objects and separates relationships.

5.1 Pickle Formats

Pickle formats are efficient to support *transport*, *marshalling* or *serialization* of objects [16]. Before going any further we give a definition of pickle and give an example.

"*Pickling* is the process of creating a serialized representation of objects. Pickling defines the serialized form to include meta information that identifies the type of each object and the relationships between objects within a stream. Values and types are serialized with enough information to insure that the equivalent typed object and the objects to which it refers can be recreated. *Unpickling* is the complementary process of recreating objects from the serialized representation." (extracted from [16])

5.2 Pickling a rectangle

To present the pickling format and algorithm in an intuitive way, we show below an example of how Fuel stores a rectangle.

In Figure 3 we create a rectangle with two points that define the origin and the corner. A rectangle is created and then passed to the serializer as an argument. In this case the rectangle is the *root* of the graph which also includes the points that the rectangle references. The first step is to analyze the graph starting from the root. Objects are mapped to *clusters* following some criteria. In this example, the criteria is 'by class', but in other cases it is 'is global object' (it is at Smalltalk dictionary), or 'is an integer between 0 and 2^{16}'.

```
FLDemo >> serializeSampleRectangleOn: aFileStream

  | aRectangle anOrigin aCorner |
  anOrigin := 10@20.
  aCorner := 30@40.
  aRectangle := Rectangle origin: anOrigin corner: aCorner.

  (FLSerializer on: aFileStream) serialize: aRectangle.
```

Figure 3. Code snippet for our example to show how Fuel serialization works.

Finally, in Figure 4 we can see how the rectangle is stored in the stream. The graph is encoded in four main sections: header, vertexes, edges and trailer. The 'Vertexes' section collects the instances of the graph. The 'Edges' section contains indexes to recreate the references of the instances. The trailer encodes the root: a reference to the rectangle.

Even if the main goal of Fuel is to be fast in materialization, the benchmarks of Section 6 show that actually Fuel is fast for both serialization and materialization. In the next section, there are more details regarding the pickle format and how the clusters work.

6. Fuel Key Characteristics

In the previous section, we explained the pickle format behind Fuel, but that is not the only key aspect. The following is a list of important characteristics of Fuel:

Grouping objects in clusters. Tipically serializers do not group objects. Thus, each object has to encode its type at serialization and decode it at deserialization. This is not only an overhead in time but also in space. In addition, each object may need to fetch its class to recreate the instance.

The purpose of grouping similar objects is to reduce the overhead on the byte representation that is necessary to encode the *type* of the objects. The idea is that the type is encoded and decoded only once for all the objects of that type. Moreoever, if recreation is needed, the operations can be grouped.

The type of an object is sometimes directly mapped to its class, but the relation is not always one to one. For example, if the object being serialized is Transcript, the type that will be assigned is the one that represents global objects. For speed reason, we distinguish between positive SmallInteger and negative one. From a Fuel perspective they are from different types.

Clusters know how to encode and decode the objects they group. Clusters are represented in Fuel's code as classes. Each cluster is a class, and each of those classes has an associated unique identifier which is a number. Such ID is encoded in stream as we saw in Figure 4. It is written only once, at the begining of a cluster instance. At materialization time, the cluster ID is read and then the associated cluster is searched. The materializer then materializes all the objects it contains.

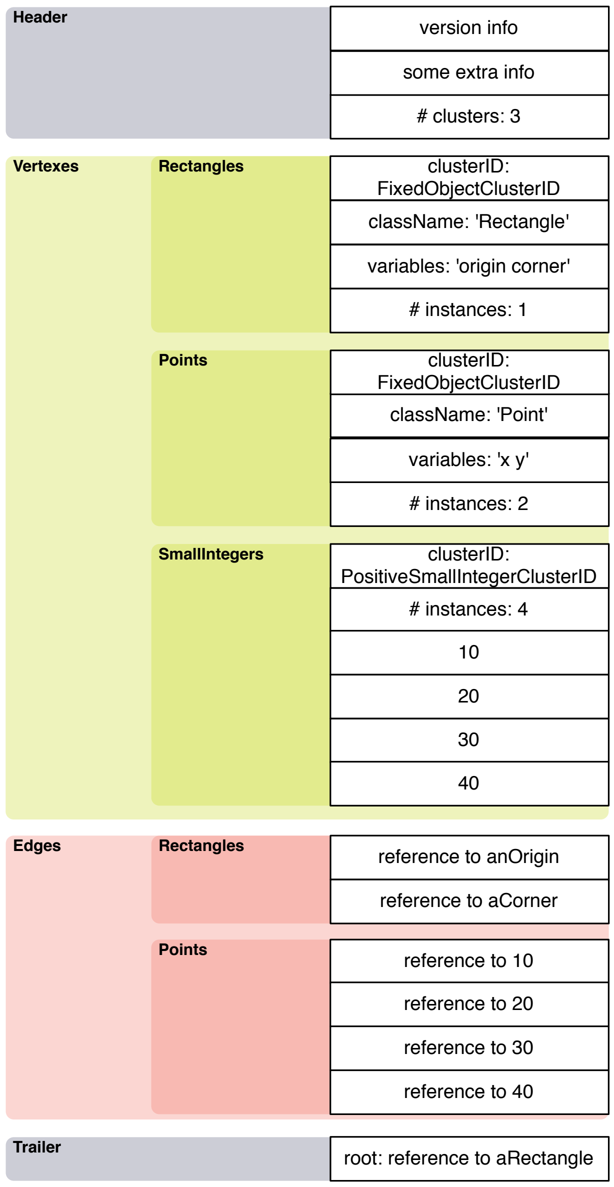


Figure 4. A graph example encoded with the pickle format.

Notice that what is associated to objects are cluster instances, not the cluster classes. The ID is unique per cluster. Some examples:

- PositiveSmallIntegerCluster is for positive instances of SmallInteger. Its ID is 4. A unique Singleton instance is used for all the objects grouped to this cluster.
- NegativeSmallIntegerCluster is for negative instances of SmallInteger. Its ID is 5. Again, it is singleton.
- FloatCluster is for Float instances. Its ID is 6. Again, it is singleton.
- FixedObjectCluster is the cluster for regular classes with indexable instance variables and that do not require any special serialization or materialization. Its ID is 10 and one instance is created for each class, *i.e.,* FixedObjectCluster has an instance variable referencing a class. One instance is created for Point and one for Rectangle.

If we analyze once again Figure 4, we can see that there is one instance of PositiveSmallIntegerCluster, and two instances of FixedObjectCluster, one for each class.

It is important to understand also that it is up to the cluster to decide what is encoded and decoded. For example, FixedObjectCluster writes into the stream a reference to the class whose instances it groups, and then it writes the instance variable names. In contrast, FloatCluster, PositiveSmallIntegerCluster or NegativeSmallIntegerCluster do not store such information because it is implicit in the cluster implementation.

In Figure 4 one can see that for small integers, the cluster directly writes the numbers 10, 20, 30 and 40 in the stream. However, the cluster for Rectangle and Point do not write the objects in the stream. This is because such objects are no more than just references to other objects. Hence, only their references are written in the 'Edges' part. In contrast, there are objects that contain self contained state, *i.e.,* objects that do not have references to other objects. Examples are Float, SmallInteger, String, ByteArray, LargePositiveInteger, etc. In those cases, the cluster associated to them have to write those values in the stream.

The way to specify custom serialization or materialization of objects is by creating specific clusters.

Analysis phase. The common approach to serialize a graph is to traverse it and while doing so to encode the objects into a stream. Fuel groups similar objects in clusters so it needs to traverse the graph and associate each object to its correct cluster. As explained, that fact significantly improves the materialization performance. Hence, Fuel does not have one single phase of traverse and writing, but instead two phases: analysis and writing.

The analysis phase has several responsibilities:

- It takes care of traversing the object graph and it associates each object to its cluster. Each cluster has a corresponding list of objects which are added there while they are analyzed.
- It checks whether an object has already been analyzed or not. Fuel supports cycles. In addition, this offers to write an object only once even if it is referenced from several objects in the graph.
- It gives support for global objects, *i.e.,* objects which are considered global and should not be written into the stream but instead put the minimal needed information to get it back the reference at materialization time. This is, for example, what is in Smalltalk globals. If there are objects in the graph referencing the instance Transcript we do not want to serialize it but instead just put a reference to it and at materialization get it back. In this case, just storing the global name is enough. The same happens with the Smalltalk class pools.

Once the analysis phase is over, the writing follows: it iterates over every cluster and for every cluster write its objects.

Stack over recursion. Most of the serializers use a depth-first traversal mechanism to serialize the object graph. Such mechanism consists of a simple recursion:

1. Take an object and look it up in a table.
2. If the object is in the table, it means that it has already been serialized. The, we take a reference from the table and write it down. If it is not present in the table, it means that the object has not been serialized and that its contents need to be written. After that, the object is serialized and a reference representation is written into the table.
3. While writing the contents, *e.g.,* instance variables of an object, the serializer can encounter simple objects such as instances of String, Float, SmallInteger, LargePositiveInteger, ByteArray or complex objects (objects which have instance variables that reference to other objects). In the latter case we start over from the first step.

This mechanism can consume too much memory depending on the graph, *e.g.,* its depth, the memory to hold all the call stack of the recursion can be too much. In addition, a Smalltalk dialect may limit the stack size.

In Fuel, we do not use the mentioned recursion but instead we use a stack. The difference is mainly in the last step of the algorithm. When an object has references to other objects, instead of following the recursion to analyze these objects, we just push such objects on a stack. Then we pop objects from the stack and analyze them. The routine is to pop and analyze elements until the stack is empty. In addition, to improve even more the speed, Fuel has its own SimpleStack class implementation.

That way, Fuel turns a recursive trace into depth-by-depth trace. With this approach the resulting stack size is much smaller and the memory footprint is smaller. At the same time, we increase serialization time by 10%.

Notice that Fuel makes it possible because of the separate analysis phase before the actual object serialization.

Two phases for writing instances and references. The encoding of objects is divided in two parts: (1) instances writing and (2) references writing. The first phase includes just the minimal information needed to recreate the instances *i.e.,* the vertexes of the graph. The second phase has the information to recreate references that connect the instances *i.e.,* the edges of the graph.

Iterative graph recreation. The pickling algorithm carefully sorts the instances paying attention to inter-depencencies.

During Fuel serialization, when a cluster is serialized, the amount of objects of such cluster is stored, as well as the total amount of objects of the whole graph. This means that at materialization time, Fuel know exactly the number of allocations (new objects) needed for each cluster. For example, one Fuel file contains 17 large integers, 5 floats, and 5 symbols, etc. In addition, for variable objects, Fuel also stores the size of such objects. So for example, it does not only know that there are 5 symbols but also that the first symbol is size 4, the second one 20, the third 6, etc.

Therefore, the materialize populates an object table with indices from 1 to N where N is the number of objects in the file. However, it does so in batch, spinning in a loop creating N instances of each class in turn, instead of determining which object to create as it walks a (flattened) input graph, as most of the serializers do.

Once that is done, the objects have been materialized, but updating the references is pending, *i.e.,* which slots refer to which objects. Again the materializer can spin filling in slots from the reference data instead of determining whether to instantiate an object or dereference an object id as it walks the input graph.

This is the main reason why materializing is much faster in Fuel than the other approaches.

Buffered write stream. A serializer usually receives two parameters from the user: the object graph to serialize and the stream where the former should be stored. Most of the time, such stream is a file stream or a socket stream. We know that writing to primary memory is much faster that writing to disc or to a network. Writing into a file stream or a network stream for every single object is terribly slow. Fuel uses an internal buffer to improve performance in that scenario. With a small buffer *e.g.,* 4096 elements, we get almost the same speed writing to a file or socket stream, as if we were writing in memory.

7. Fuel Features

In this section we analyze Fuel in accordance with the concerns and features defined in Section 4.

7.1 Fuel serializer concerns

Performance. We achieved an excellent time performance. This topic is extensively studied and explained in Section 8.

Completeness. We are close to say that Fuel deals with all kinds of objects. Notice the difference between being able to serialize and to get something meaningful while materializing. For example, Fuel can serialize and materialize instances of Socket, Process or FileStream, but it is not sure they will still be valid once they are materialized. For example, the operating system may have given the socket address to another process, the file associated to the file stream may have been removed, etc.

There is no magic. Fuel provides the infrastructure to solve the mentioned problems. For example, there is a hook where a message can be implemented in a particular class and such message will be send once the materialization is done. In such method one can implement the necessary behavior to get a meaningful object. For instance, a new socket may be asked and assigned. Nevertheless sometimes there is nothing to do, *e.g.,* if the file of the file stream was removed by the operating system.

It is worth to note here that some well known special objects are treated as external references, because that is the expected behavior for a serializer. Some examples are Smalltalk, Transcript and Processor.

Portability. As we explained in other sections, portability is not our main focus. Thus, the only portability aspect we are interested in is between Fuel versions. Below we talk about our versioning mechanism.

Versatility and Abstraction capacity. Both concerns are tightly tied. The goal is to be as concrete and exact as possible in the graph recreation, but providing flexible ways to customize abstractions as well as other kinds of substitutions in the graph. This last affirmation still has to be implemented.

Security. Our goal is to give the user the possibility to configure validation rules to be applied over the graph (ideally) before having any secondary effect on the environment. This is not yet implemented.

Atomicity. Unfortunately, Fuel can have problems if the graph changes during the analysis or serialization phase. This is an issue we have to work on in next versions.

7.2 Fuel serializer challenges

In this section, we explain how Fuel implements some of the features previously commented. There are some mentioned challenges we do not include here because Fuel does not support them at the moment.

Maintaining identity. Although it can be disabled, the default behavior when traversing the graph is to recognize some objects as *external references*: Classes registered in Smalltalk, global objects *i.e.,* referenced by global variables, global bindings *i.e.,* included in Smalltalk globals associations, and class variable bindings *i.e.,* included in the classPool of a registered class.

It is worth to mention that this mapping is done at object granularity, *e.g.,* not every instance of Class will be recognized as external. If a class is not in Smalltalk globals or if it has been specified as an *internal class*, it will be traversed and serialized in full detail.

Transient values. Fuel supports this by the class-side hook fuelIgnoredInstanceVariableNames. The user can specify there a list of variable names whose values will not be traversed or serialized. On materialization they will be restored as nil.

Cyclic object graphs and duplicates. Fuel checks that every object of the graph is visited once, supporting both cycle and duplicate detection.

Class shape change tolerance. Fuel stores the list of variable names of the classes that have instances in the graph being written. While recreating an object from the stream, if its class (or anyone in the hierarchy) has changed, then this meta information serves to automatically adapt the stored instances. When a variable does not exist anymore, its value is ignored. If a variable is new, it is restored as nil.

Versioning. This is provided by Fuel through a experimental implementation that wraps the standard Fuel format, appending at the beginning of the stream a version number. When reading, if such number matches with the current version, it is straightforward materialized in the standard way. If it does not match, then another action can be taken depending on the version. For example, suppose the difference between the saved version and the current version is a cluster that has been optimized and so their formats are incompatible. Now, suppose the wrapper has access to the old cluster class and so it configures the materializer to use this implementation instead of the optimized one. Then, it can adapt the way it reads, providing backward compatibility.

Graphical progress update. Fuel has an optional package that is used to show a progress bar while processing either the analysis, the serialization or the materialization. This *GUI* behavior is added via subclassification of Fuel core classes, therefore it does not add overhead to the standard *non-interactive* use case. This implementation is provisional since the future idea is to apply a *strategy design pattern*.

8. Benchmarks

We have developed several benchmarks to compare different serializers. To get meaningful results all benchmarks have to be run in the same environment. In our case we run them in the same Smalltalk dialect, with the same Virtual Machine and same Smalltalk image. Fuel is developed in Pharo, and hence the benchmarks were run in Pharo Smalltalk, image version Pharo-1.3-13257 and Cog Virtual Machine version "VMMaker.oscog-eem.56". The operating system was Mac OS 10.6.7.

In the following benchmarks we have analyzed the serializers: Fuel, SIXX, SmartRefStream, ImageSegment, Magma object database serializer, StOMP and SRP. Such serializers are explained in Section 9.

8.1 Benchmarks Constraints and Characteristics

Benchmarking software as complex as a serializer is difficult because there are multiple functions to measure which are used independently in various real-world use-cases. Moreover, measuring only the speed of a serializer is not complete and it may not even be fair if we do not mention the provided features of each serializer. For example, providing a hook for user-defined reinitialization action after materialization, or supporting class shape changes slows down serializers.

Here is a list of constraints and characteristics we used to get meaningful benchmarks:

All serializers in the same environment. We are not interested in compare speed with serializers that do not run in Pharo.

Use default configuration for all serializers. Some serializers provide customizations to improve performance, *i.e.,* some parameters or settings that the user can set for serializing a particular object graph. Those settings would make the serialization or materialization faster. For example, a serializer can provide a way to do *not* detect cycles. Detecting cycles takes time and memory hence, not detecting them is faster. Consequently, if there is a cycle in the object graph to serialize, there will be a loop and finally a system crash. Nevertheless, in certain scenarios the user may have a graph that he knows that there is no cycles.

Streams. Another important point while measuring serializers performance is which stream to be used. Usually, one can use memory based stream based and file based streams. Both measures are important and there can be significant differences between them.

Distinguish serialization from materialization. It makes sense to consider different benchmarks for the serialization and for the materialization.

Different kind of samples. Benchmark samples are split in two kinds: primitive and large. Samples of primitive objects are samples with lots of objects which are instances of the same class and that class is "primitive". Examples of those classes are Bitmap, Float, SmallInteger, LargePositiveInteger, LargeNegativeInteger, String, Symbol, WideString, Character, ByteArray, etc. Large objects are objects which are composed by other objects which are instances of different classes, generating a large object graph.

Primitive samples are useful to detect whether one serializer is better than the rest while serializing or materializing certain type of object. Large samples are more similar to the expected user provided graphs to serialize and they try to benchmark examples of real life object graphs.

Avoid JIT side effects. In Cog (the VM we used for benchmarks), the first time a method is used, it is executed in the standard way and added to the method cache. The second time the method is used, that means, when it is found in the cache, Cog converts that method to machine code. However, extra time is needed for such task. Only the third time the method will be executed as machine code and without extra effort.

It is not fair to run sometimes with methods that has been converted to machine code and sometimes with methods that have not. Therefore, for the samples we first run twice the same sample without taking into account its execution time to be sure to be always in the same condition. Then the sample is finally executed and its execution time is computed.

8.2 Benchmarks serializing with memory based streams

In this benchmarks we use memory based streams.

Primitive objects serialization. Figure 5 shows the results of primitive objects serialization.

Figure 6 shows the materialization of the serialization done for Figure 5.

The conclusions for serializing primitive objects with memory based streams are:

- We did not include SIXX in the charts because it was so slow that otherwise we were not able to show the differences between the rest of the serializers. This result is expected since SIXX is a text based serializer, which is far slower than a binary one. However, SIXX can be opened and modified by any text editor. This is an usual trade-off between text and binary formats.
- Magma and SmartRefStream serializers seem to be the slowest ones most of the cases.
- StOMP is the fastest one in serialization. Near to them there are Fuel, SRP, and ImageSegment.

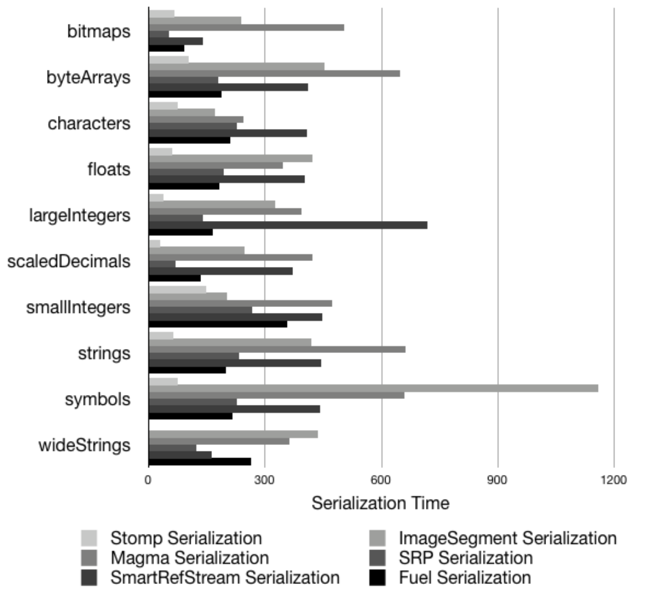


Figure 5. Primitive objects serialization in memory.

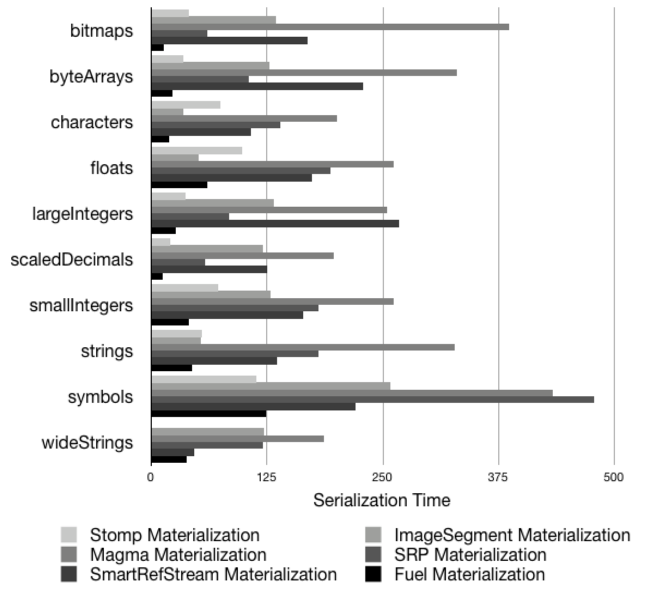


Figure 6. Primitive objects materialization from memory.

- Magma serializer is slow with "raw bytes" objects such as Bitmap and ByteArray, etc.
- Most of the cases, Fuel is faster than ImageSegment, which is even implemented in the Virtual Machine.
- ImageSegment is really slow with Symbol instances. We explain the reason later.
- StOMP has a zero (its color does not even appear) in the WideString sample. That means that cannot serialize those objects.
- In materialization, Fuel is the fastest one. Then after there are StOMP and ImageSegment.

Large objects serialization. As explained, these samples contain objects which are composed by other objects which are instances of different classes, generating a large object graph. Figure 7 shows the results of large objects serialization. Such serialization is also done with memory based stream. Figure 8 presents the materialization results when using the same scenario of Figure 7.

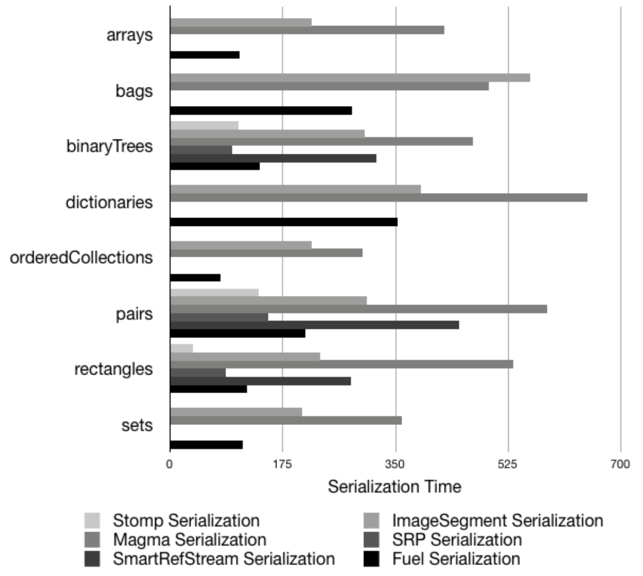


Figure 7. Large objects serialization in memory.

The conclusions for large objects are:

- The differences in speed are similar to the previous benchmarks. This means that whether we serialize graphs of all primitive objects or objects instances of all different classes, Fuel is the fastest one in materialization and one of the best ones in serialization.
- StOMP, SRP, and SmartRefStream cannot serialize the samples for *arrays*, *orderedCollections*, *sets*, etc. This is because those samples contain different kind of objects, included BlockClosure and MethodContext. This demonstrates that the mentioned serializers does not support serialization and materialization of all kind of objects. At least, not out-of-the-box.

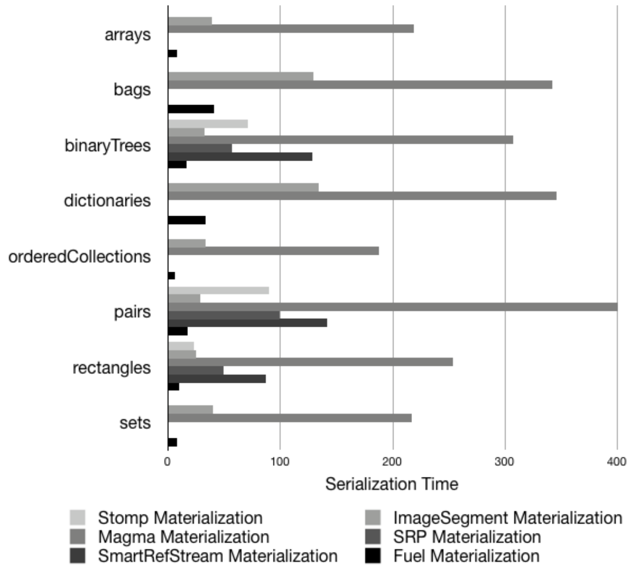


Figure 8. Large objects materialization from memory.

8.3 Benchmarks serializing with file based streams

Now we use file based streams. In fact, the exact stream we use is MultiByteFileStream. Figure 9 shows the results of primitive objects serialization.

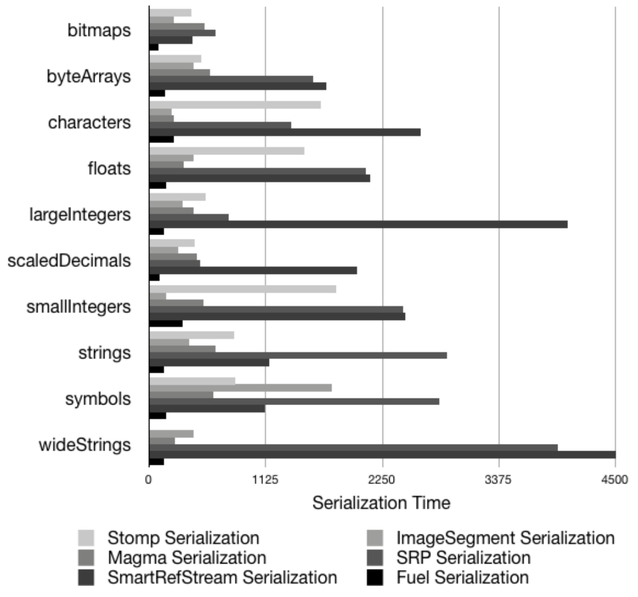


Figure 9. Primitive objects serialization in file.

Figure 10 shows the same scenario of Figure 9 but the results of the materialization.

The conclusions for serialization with file based streams are:

- It is surprising the differences between serializing in memory and in file. In serialization, SmartRefStream is by far the slowest.

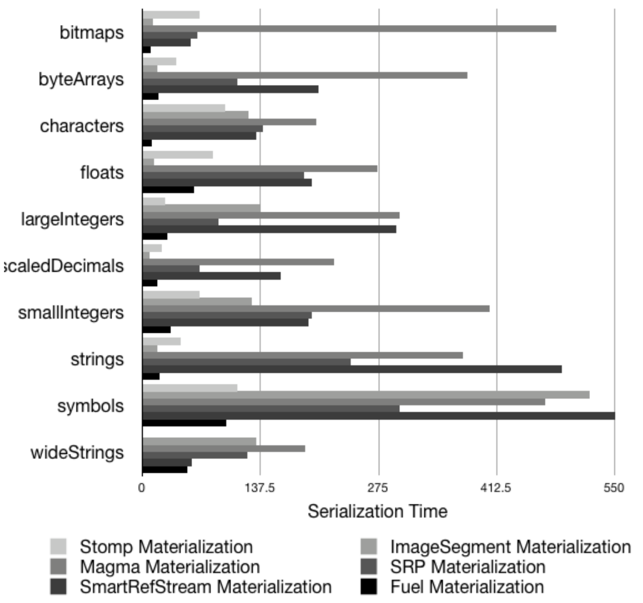


Figure 10. Primitive objects materialization from file.

- SRP and StOMP have good performance when serializing in memory, but not at all when serializing to a file based stream.
- Fuel is the fastest one, taking advantage of its internal buffering technique.
- Magma was one of the slowest in memory based stream but in this case it is much better.
- SmartRefStream and SRP are really slow with WideString instances.
- ImageSegment is slow with Symbol instances.

The conclusions for materialization with file based streams are:

- Again, Magma serializer is slow with "raw bytes" objects such as Bitmap and ByteArray, etc.
- ImageSegment is slow with Symbol instances.

These benchmarks showed that different serializers perform differently when serializing in memory or in files.

8.4 ImageSegment Results Explained

ImageSegment seems to be really fast in certain scenarios. However, it deserves some explanations of how ImageSegment works. Basically, ImageSegment receives the user defined graph and it needs to distinguish between *shared objects* and *inner objects*. Inner objects are those objects inside the subgraph which are *only* referenced from objects inside the subgraph. *Shared objects* are those which are not only referenced from objects inside the subgraph, but also from objects outside.

All *inner objects* are put into a byte array which is finally written into the stream using a primitive implemented in the virtual machine. After, ImageSegment uses SmartRefStream to serialize the *shared objects*. ImageSegment is fast mostly because it is implemented in the virtual machine. However, as we saw in our benchmarks, SmartRefStream is not really fast. The real problem is that it is difficult to control which objects in the system are pointing to objects inside the subgraph. Hence, most of the times there are several *shared objects* in the graph. The result is that the more *shared objects* there are, the slower ImageSegment is because those *shared objects* will be serialized by SmartRefStream.

All the benchmarks we did with primitive objects (all but Symbol) take care to create graphs with zero or few shared objects. That means that we are measuring the fastest possible case ever for ImageSegment. Nevertheless, in the sample of Symbol one can see in Figure 5 that ImageSegment is really slow in serialization, and the same happens with materialization. The reason is that in Smalltalk all instances of Symbol are unique and referenced by a global table. Hence, all Symbol instances are shared and therefore, serialized with SmartRefStream.

We did an experiment where we build an object graph and we increase the percentage of *shared objects*.

Figure 11 shows the results of primitive objects serialization with file based stream. Axis *X* represents the percentage of shared objects inside the graph and the axis *Y* represents the time of the serialization.

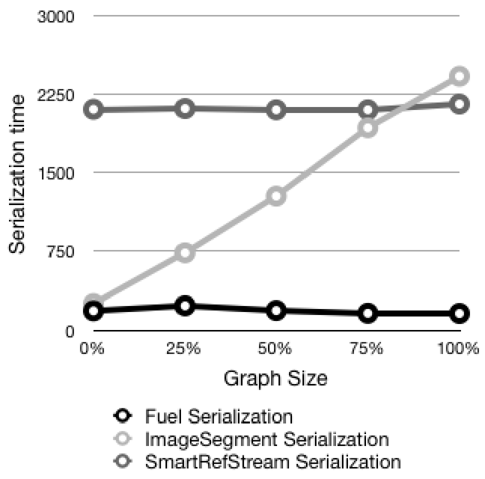


Figure 11. ImageSegment serialization in presence of shared objects.

Figure 12 shows the same scenario of Figure 11 but the results of the materialization.

Conclusions for ImageSegment results

- The more shared objects there are, the more ImageSegment speed is similar to SmartRefStream.
- For materialization, when all are shared objects, ImageSegment and SmartRefStream have almost the same speed.
- For serialization, when all are shared objects, ImageSegment is even slower than SmartRefStream. This is be-

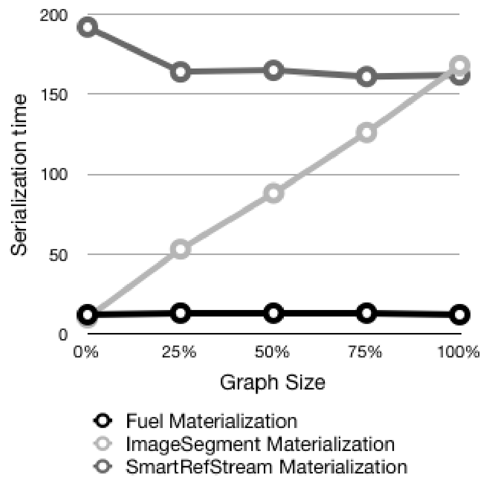


Figure 12. ImageSegment materialization in presence of shared objects.

cause ImageSegment needs to do the whole memory traverse anyway to discover shared objects.

- ImageSegment is unique in the sense that its performance depends in both: 1) the amount of references from outside the subgraph to objects inside; 2) the total amount of objects in the system, since the time to traverse the whole memory depends on that.

8.5 General Benchmarks Conclusions

Magma serializer seems slow when serializing in memory, but it is acceptable taking into account that this serializer is designed for a particular database. Hence, the Magma serializer does extra effort and stores extra information that is needed in a database scenario but may not be necessary for any other usage.

SmartRefSteam provides a good set of hook methods for customizing serialization and materialization. However, it is slow and its code and design are not good from our point of view. ImageSegment is known to be really fast because it is implemented inside the virtual machine. Such fact, together with the problem of *shared objects*, brings a large number of limitations and drawbacks as it has been already explained. Furthermore, with Cog we demonstrate that Fuel is even faster in both, materialization and serialization. Hence, the limitations of ImageSegment are not worth it.

SRP and StOMP are both aimed for portability across Smalltalk dialects. Their performance is good, mostly at writing time, but they are not as fast as they could because of the need of being portable across platforms. In addition, for the same reason, they do not support serialization for all kind of objects.

This paper demonstrates that Fuel is the fastest in materialization and one the fastest ones in serialization. In fact, when serializing to files, which is what usually happens, Fuel is the fastest. Fuel can also serialize any kind of object. Fuel aim is not portability but performance. Hence, all the results make sense from the goals point of view.

9. Related work

The most common example of a serializer is one based on XML like SIXX [17] or JSON [9]. In this case the object graph is exported into a portable text file. The main problem with text-based serialization is encountered with big graphs as it does not have a good performance and it generates huge files. Other alternatives are ReferenceStream or SmartReferenceStream. ReferenceStream is a way of serializing a tree of objects into a binary file. A ReferenceStream can store one or more objects in a persistent form including sharing and cycles. The main problem of ReferenceStream is that it is slow for large graphs.

A much more elaborated approach is Parcel [12] developed in VisualWorks Smalltalk. Fuel is based on Parcel's pickling ideas. Parcel is an atomic deployment mechanism for objects and source code that supports shape changing of classes, method addition, method replacement and partial loading. The key to making this deployment mechanism feasible and fast is a pickling algorithm. Although Parcel supports code and objects, it is more intended to source code than normal objects. It defines a custom format and generates binary files. Parcel has good performance and the assumption is that the user may not have a problem if saving code takes more time, as long as loading is really fast.

The recent StOMP[1] (Smalltalk Objects on MessagePack[2]) and the mature SRP[3] (State Replication Protocol) are binary serializers with similar goals: Smalltalk-dialect portability and space efficiency. They are quite fast and configurable, but they are limited with dialect-dependent objects like BlockClosure and MethodContext. Despite the fact that their main goals differ from ours, we should take into account their designs.

Object serializers are needed and used not only by final users, but also for specific type of applications or tools. What it is interesting is that they can be used outside the scope of their project. Some examples are the object serializers of Monticello2 (a source code version system), Magma object database, Hessian binary web service protocol [7] or Oracle Coherence*Web HTTP session management [13].

Martinez-Peck et al. [11] performed an analysis of ImageSegment (a virtual machine serialization algorithm) and they found that the speed increase in ImageSegment is mainly because it is written in C compared to other frameworks written in Smalltalk. However, ImageSegment is slower when objects in the subgraph to be serialized are externally referenced.

[1] http://www.squeaksource.com/STOMP.html

[2] http://msgpack.org

[3] http://sourceforge.net/projects/srp/

10. Conclusion and Future Work

In this paper, we have looked into the problem of serializing object graphs in object oriented systems. We have analyzed its problems and challenges. What is important is that these steps, together with their problems and challenges, are general and they are independent of the technology.

These object graphs operations are important to support virtual memory, backups, migrations, exportations, etc. Speed is the biggest constraint in these kind of graph operations. Any possible solution has to be fast enough to be actually useful. In addition, this problem of performance is the most common problem among the different solutions. Most of them do not deal properly with it.

We presented Fuel, a general purpose object graph serializer based on a pickling format and algorithm different from typical serializers. The advantage is that the unpickling process is optimized. In one hand, the objects of a particular class are instantiated in bulk since they were carefully sorted when pickling. In the other hand, this is done in an iterative instead of a recursive way, what is common in serializers. The disadvantage is that the pickling process takes an extra time in comparison with other approaches. Besides, we show in detailed benchmarks that we have the best performance in most of the scenarios we are interested in.

We implement and validate this approach in the Pharo Smalltalk environment. We demonstrate that it is possible to build a fast serializer without specific VM support, with a clean object-oriented design, and providing most possible required features for a serializer.

Even if Fuel has an excellent performance and provided hooks, it can still be improved. Regarding the hooks, we would like to provide one that can let the user replace one object by another one, which means that the serialized graph is not exactly the same as the one provided by the user.

Instead of throwing an error, it is our plan to analyze the possibility of create light-weight shadow classes when materializing instances of an inexistent class. Another important issue we would like to work on is in making everything optional, *e.g.,* cycle detection. Partial loading as well as being able to query a serialized graph are concepts we want to work in the future.

To conclude, Fuel is a fast object serializer built with a clean design, easy to extend and customize. New features will be added in the future and several tools will be build on top of it.

Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the CPER 2007-2013.

References

[1] J. K. Bennett. The design and implementation of distributed Smalltalk. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 318–330, Dec. 1987.

[2] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[3] M. D. Bond and K. S. McKinley. Tolerating memory leaks. In G. E. Harris, editor, *OOPSLA: Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 109–126. ACM, 2008.

[4] F. Breg and C. D. Polychronopoulos. Java virtual machine support for object serialization. In *Joint ACM Java Grande - ISCOPE 2001 Conference*, 2001.

[5] P. Butterworth, A. Otis, and J. Stein. The GemStone object database management system. *Commun. ACM*, 34(10):64–77, 1991.

[6] D. Decouchant. Design of a distributed object manager for the Smalltalk-80 system. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 444–452, Nov. 1986.

[7] Hessian. http://hessian.caucho.com.

[8] Java serializer api. http://java.sun.com/developer/technicalArticles/Programming/serialization/.

[9] Json (javascript object notation). http://www.json.org.

[10] T. Kaehler. Virtual memory on a narrow machine for an object-oriented language. *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, 21(11):87–106, Nov. 1986.

[11] M. Martinez Peck, N. Bouraqadi, M. Denker, S. Ducasse, and L. Fabresse. Experiments with a fast object swapper. In *Smalltalks 2010*, 2010.

[12] E. Miranda, D. Leibs, and R. Wuyts. Parcels: a fast and feature-rich binary deployment technology. *Journal of Computer Languages, Systems and Structures*, 31(3-4):165–182, May 2005.

[13] Oracle coherence. http://coherence.oracle.com.

[14] Pickle. http://docs.python.org/library/pickle.html.

[15] Google protocol buffers. http://code.google.com/apis/protocolbuffers/docs/overview.html.

[16] R. Riggs, J. Waldo, A. Wollrath, and K. Bharat. Pickling state in the Java system. *Computing Systems*, 9(4):291–312, 1996.

[17] Sixx (smalltalk instance exchange in xml). http://www.mars.dti.ne.jp/~umejava/smalltalk/sixx/index.html.

[18] D. Ungar. Annotating objects for transport to other worlds. In *Proceedings OOPSLA '95*, pages 73–87, 1995.

[19] D. Wiebe. A distributed repository for immutable persistent objects. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 453–465, Nov. 1986.

Using First-class Contexts to realize Dynamic Software Updates

Erwann Wernli David Gurtner Oscar Nierstrasz
Software Composition Group, University of Bern, Switzerland
http://scg.unibe.ch/

Abstract

Applications that need to be updated but cannot be easily restarted must be updated at run-time. We evaluate the reflective facilities of Smalltalk with respect to dynamic software and the state-of-the-art in this field. We conclude that while fine for debugging, the existing reflective facilities are not appropriate for dynamically updating production systems under constant load. We propose to enable dynamic updates by introducing first-class contexts as a mechanism to allow multiple versions of objects to coexist. Object states can be dynamically migrated from one context to another, and can be kept in sync with the help of bidirectional transformations. We demonstrate our approach with *ActiveContext*, an extension of Smalltalk with first-class contexts. ActiveContext eliminates the need for a system to be quiescent for it to be updated. ActiveContext is realized in Pinocchio, an experimental Smalltalk implementation that fully reifies the VM to enable radical extensions. We illustrate dynamic updates in ActiveContext with a typical use case, present initial benchmarks, and discuss future performance improvements.

1. Introduction

Software needs to be updated: Apart from the need to continuously evolve applications to support new and possibly unanticipated features, there is also a need to fix existing bugs.

Changing the system at run-time for debugging purposes has long been a common practice in dynamic object-oriented languages such as JavaScript, Ruby and Smalltalk. In the case of Smalltalk, as it is fully reflective, there is actually no other way to change the system than to adapt it at run-time, and development in Smalltalk is conducted with this in mind.

In this paper we discuss the requirements for the dynamic update of production systems, and evaluate the reflective capabilities of Smalltalk according to these requirements. We consider three main dimensions during this evaluation: *safety*, the ability to ensure that dynamic updates will not lead to abnormal executions, *timeliness*, the ability to install the update quickly, and *practicality*, the fact that the dynamic software update mechanism must not constrain developers. The outcome of this analysis is that while the existing reflective mechanisms are fine for debugging, they are not adequate to update production systems under constant load, notably because of safety issues.

As a remedy to the problems we have identified, we propose ActiveContext, an extension of Smalltalk with first-class contexts. Contexts have two roles: they dynamically scope software versions, and they mediate access to objects that can be migrated back and forth between versions. As a consequence, the single abstraction of a first-class context enables not only the *isolation* of software versions, but also the *transition* from one version to another.

Making contexts *first-class* empowers the developers with more control over how dynamic updates should happen; it shifts part of the responsibility of the update from the system to the application. This way, it becomes possible to tailor the update scheme to the nature of the application, *e.g.*, rolling out new code on a per-thread basis for an FTP server, or on a per-session basis for a web application.

Entities shared between several versions are mediated by the system and code running entirely in one version is guaranteed to be always consistent. The abstraction of context is intuitive and can be implemented with a reasonable overhead. ActiveContext qualifies as a safe and practical approach to dynamic software update.

As a validation, we have implemented ActiveContext in Pinocchio, an experimental Smalltalk platform that fully reifies the runtime to enable invasive language extensions. We demonstrate a typical use case with a Telnet server.

First we discuss the challenges of dynamic software update and review existing literature in section 2. In section 3 we present our approach with the help of a running example. In section 4 we present the model in more detail, and in section 5 we present an implementation. We discuss future work in section 6 before we conclude in section 7.

2. Why dynamic updates are challenging

The main challenge of dynamic updates is achieving an optimal compromise between safety, timeliness and practicality. Safety means that dynamic updates are guaranteed to not lead to abnormal executions ; timeliness means that updates are installed immediately and instantly; practicality means that the system does not impose additional constraints during development or operation. We resort to common sense for why these characteristics are desirable.

To understand why there are tensions between these properties, let's consider an update that alters a method signature. Once installed, subsequent invocations of the method will use the updated method body that expects the newest list of arguments. It is clear that installing the change immediately is unsafe, as *active methods* on the stack might still presume the old signature, which will lead to type errors [21]. To increase safety, type errors can be prevented with the proper timing of the update using automated checks, but the behavior of the program might still be incorrect depending on the change in the program's logic [23]. Manual assistance to define safe update times is required [12, 22]. This affects negatively both practicality and timeliness.

Obviously, the layout of classes (the set of fields and methods) can change as well, which means the program state (object instances) must be adapted correspondingly during the update. If the update is immediate, active methods might presume the old type and lead to inconsistent accesses to state. Automated checks to delay the update can prevent such type errors, but are not enough. In the case of state, not only should we consider when to migrate the state, but how : existing invariants in the program state must be preserved and safe updates require manual assistance to provide adapters that will migrate the old state to a new, valid, state. This impacts practicality negatively.

A way to reconcile safety and timeliness is to restrict the update to only certain changes, *e.g.*, changes that do not alter types, or changes that are behavior-preserving [19], but this impedes practicality.

Note that transferring the state for large heaps has the same tensions between safety, timeliness and practicality : state transfer in a stop-the-world fashion is safe and practical but compromises timeliness, while lazy state transfer is timely but either unsafe, or less practical [2] depending on the design.

2.1 Assessment of Smalltalk

Now that we have explored the core reasons of these tensions, let's focus on Smalltalk and assess its reflective capabilities according to safety, timeliness and practicality:

Safety. Smalltalk initializes new fields to `nil` and does not allow to customize state transfer such that it maintains existing invariants in the program state.

Run-time errors can occur after an update. When a field is removed, all methods of the class are recompiled. Accesses to the suppressed field return `nil` instead, and assignment to the suppressed field are ignored. Old versions of the method existing on the stack might continue to run, which can lead to severe errors such as the mutation of instance variable at the wrong index, or even the crash of the entire image. Method suppression does not suffer such severe symptoms, as method resolution is dynamic, and at worst raises a `doesNotUnderstand` error.

Smalltalk does not support the atomic installation of co-related changes to multiple classes. An execution that uses a mix of old and new versions of the classes might be incorrect.

Timeliness. Changes are installed immediately. Object instances are migrated in a stop-the-world fashion. Changes on classes that are higher in the class hierarchy might result in the recompilation of many subclasses, as well as the migration of their object instances, which might take long.

Practicality. Arbitrary changes to method signature, method body, class hierarchy or class shape are supported. There is no overhead after the installation of the update.

As this analysis shows, the reflective capabilities of Smalltalk are timely and practical, but not safe, which makes them inadequate to update production systems on the fly. In practice, developers rely on ad-hoc scripts and techniques to update their Smalltalk image in production.

2.2 Other approaches

A large body of research has tackled the dynamic update of applications, but no mechanism resolved all three tensions previously presented. Existing approaches can be classified into three categories:

1. Systems supporting *immediate and global dynamic update* have been devised with various levels of safety and practicality. Dynamic languages other than Smalltalk belong naturally to this category; they are very practical but not safe. Dynamic AOP and meta-object protocols also fit into this category. Systems for Java [5, 8, 11, 16, 24, 26] of this kind have been devised. However, they are less practical and impose restrictions on the kinds of changes supported, due to Java's type system [30]. For example, only method bodies can be updated with HotSwap [8]. Other systems have tried to reconcile practicality and static typing, at the expense of timeliness or safety. For example, some updates will be rejected if they are not provably type-safe [20] or might produce run-time errors [33].
2. Several approaches have tackled the problem of safety by relying on *update points* to define temporal point when it is safe to globally update the application. Such systems have been devised for C [14, 22], and Java [28]. Update points might be hard to reach, especially in multi-

threaded applications [21], and this compromises the timely installation of updates.

3. Some approaches do not carry out the update globally and allow different versions of the entities to coexist at run-time. Different variations of this scheme exist. With dynamic C++ classes [15], the structure of existing objects is not altered, and only new objects have the new structure ; the code is however updated globally which is unsafe. With Gemstone [10], class histories enable different versions of classes to coexist, and objects can be migrated on demand from one version to another ; this is more flexible than an immediate migration but is still unsafe. A strategy that is safe is to adapt the entities back and forth when accessed from different versions of the code using bi-directional transformations. To the best of our knowledge, only two approaches have pursued the latter one: a dynamic update system for C [6], and a type system extended with the notion of run-time version tags that enables hot swapping modules [9].

More generally, dynamic software updating relates to techniques that promote *late binding*. Three main categories of such techniques can be listed: support for virtual classes [18], isolation of software versions (Java class loader [17], ChangeBox [7], and ObjectSpace [4]), and fine-grained scoping of variations (selector namespaces, context-oriented programming [29], classbox [1]). Finally, the migration of instances relates to the problem of *schema evolution* [25] and techniques to convert between types (expanders [32], translation polymorphism [13], implicit conversion [27]). None of these techniques is in itself sufficient to enable dynamic software update though.

3. Our approach — ActiveContext

We believe that dynamic software updates should be addressed explicitly (*i.e.*, reflectively), so as to give developers control over when new code becomes active. Developers should be able to implement an appropriate update scheme for the application, such as roll out new code on a per-thread basis for an FTP server, or on a per-session basis for a web application.

Most approaches for dynamic software updates are either non-reflective, or reflective but lack safety. One notable exception is the work by Duggan [9], which is both reflective and safe. However, it relies on static typing, and the update of a module impacts all modules depending on it: they must all be updated and use the new type, which is not practical. It also fails to support the atomic change of multiple types at once.

ActiveContext is an approach that aims to introduce an ideal dynamic software update mechanism that is explicit and satisfies all requirements established previously. It introduces first-class contexts into the language in a way that reifies software versions. Contexts can be loaded, instantiated, and manipulated explicitly. A context defines a dynamic scope, within which code can be executed, which is just as simple as `aContext do: [ .... ]` .

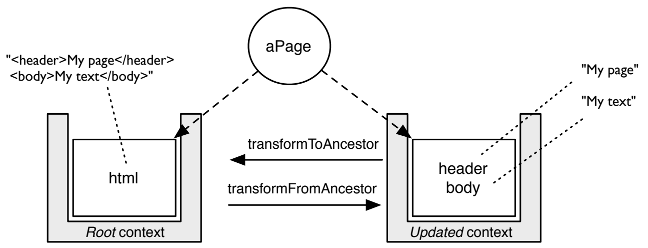


Figure 1. An instance of a `Page` object has different states in different contexts. There are transformation functions between the two contexts.

Contexts also encode bi-directional transformations for objects whose representations differ between contexts, but whose consistency is maintained by the system. Thanks to the mediation of objects using bi-directional transformations, code running in different contexts can coexist at run-time, yet safely operate on shared data. ActiveContext belongs to the third kind of approach described in subsection 2.2.

3.1 ActiveContext examplified

To illustrate our approach to dynamic software updates, let us consider the evolution of a class in a Web Content Management System like Pier[1]. The main entity of the domain model of such a system is the `Page`. It represents the source code of an HTML document. The domain model is held in memory, globally accessible, and shared amongst all threads serving HTTP requests. Let us consider the evolution shown in Figure 1. The `Page` class is refactored, and the data stored originally in the `html` field is now stored in two individual fields `body` and `header`.

Such an evolution cannot be easily achieved with the reflective facilities of Smalltalk: it would require an "intermediate" version of the class with all three fields `html`, `body`, `header` in order to allow the state of the concerned object instances to be migrated incrementally, for instance with `Page allInstances do: [...]`. Only then could the `html` field be removed. Such an update is not only complicated to install, but is also not atomic, possibly leading to consistency issues.

Dynamic software update mechanisms that carry out immediate updates (first category in subsection 2.2) face the risk that some existing thread running old code may access the `html` field which no longer exists. Those which use update points (2nd category in subsection 2.2) would still have to wait until all requests complete prior to a global update of the system state.

The following steps describe how such an update can be installed with ActiveContext while avoiding these issues. First, the application must be adapted so that we can "push"

[1] `http://www.piercms.com`

an update to the system and activate it. Here is how one would typically adapt a Web Content Management System or any server-side software serving requests.

0. *Preparation.* First, a global variable `latestContext` is added to track the latest execution context to be used. Second, an administrative page is added to the Web Content Management System where an administrator can push updates to the system; the uploaded code will be loaded dynamically. Third, the main loop that listens to incoming requests is modified so that when a new thread is spawned to handle the incoming request, the latest execution context is used:

```
latestContext do: [
    [ anIncomingRequest process ] fork.
]
```

After these preliminary modifications the system can be started, and now supports dynamic updates. The lifecycle of the system is now the following:

1. *Bootstrap.* After the system bootstraps, the application runs in a default context named the *Root* context. The global variable `latestContext` refers to the *Root* context. At this stage only one context exists and the system is similar to a non-contextual system.
2. *Offline evolution.* During development, the field `html` is replaced with the two fields `body` and `header`. Figure 1 shows the impact on the state of a page.
3. *Update preparation.* The developer creates a class, say called `UpdatedContext`, that specifies the variations in the program to be rolled out dynamically. This is done by implementing a bidirectional transformation which converts the program state between the *Root* context and the *Updated* context. Objects will be transformed one at a time.

   In our example, the field `html` is split into `body` and `header` in one direction, and the fields `body` and `header` are joined into `html` in the other direction. The class of an object is considered to be part of the object's state and the transfer function also specifies that an updated version of the `Page` class will be used in the *Updated* context.

   Contexts may coexist at run-time for any length of time. It is therefore necessary that the object representations stay globally consistent with one another, which explains the need for a *bidirectional* transformation: if the state of an object is modified in one context, the effect propagates to the representation in the other contexts as well. Only fields that make sense need to be updated though; fields that have been added or removed and have no counterpart in another context can naturally be omitted from the transformations.
4. *Update push.* Using the administrative web interface, the developer uploads the updated `Page` class and the `UpdatedContext` class. The application loads the code dynamically. It detects that one class is a context and instantiates it. This results in the generation of the new representation of all pages in the system. Objects now have two representations in memory. Last, the global variable `latestContext` is updated and refers to the newly created instance of the *Updated* context.

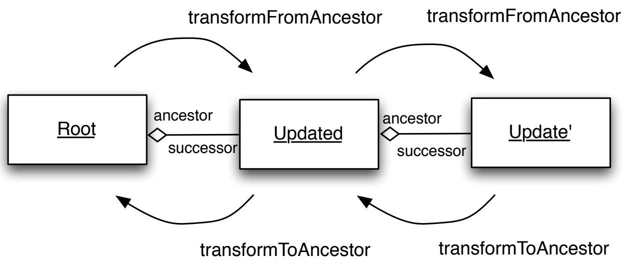


Figure 2. Context instances form a list

5. *Update activation.* When a new incoming request is accepted, the application spawns a new thread to serve the request. The active context that was dynamically changed in the listener thread (see point 0) propagates to the spawned thread. The execution context will be the context referenced in `latestContext`, which is now the *Updated* context.
6. *Stabilization.* This update scheme changes the execution context per thread. Existing threads serving ongoing requests will finish their execution in the *Root* context, while new threads will use the *Updated* context. Assuming that requests always terminate, the system will eventually stabilize. A page can always be accessed safely from one execution context or another as the programming model maintains the consistency of various representations using the bidirectional transformations. This alleviates the need for global, temporally synchronized update points which can be hard to reach in multithreaded systems.

Subsequent updates will be rolled out following the same scheme. For each update a context class is created, and then loaded and instantiated dynamically. Contexts are related to each other with an ancestor/successor relationship. They form a list, with the *Root* context as oldest ancestor, as shown in Figure 2.

7. *Garbage collection.* When no code runs in the oldest context any longer, the context can be removed from the list and be garbage collected, just as the representation of objects in it. (This step is not implemented yet)

4. The ActiveContext Model

We now present the ActiveContext model in more detail. ActiveContext makes a clear distinction between the identity and the *contextual state* of an object. An object can have several representations which remain consistent with one another thanks to state transformations. Behavior can change as well, since the class of an object is part of its state. ActiveContext is a programming model that supports dynamic scoping of state and migration of state between contexts.

4.1 Identity, State and Contexts

An object identifier uniquely identifies an object in any given context (where that object exists). The state of an object may, however, vary from context to context. The contextual state of an object consists of (i) a set of fields and their corresponding values, and (ii) its class, which depends on the context. Of course, the fields of the object must match the fields declared in the (contextual) class description.

An object can have as many states as there are contexts. A context can be seen as a mapping between the global identities of all objects and their corresponding states in that context. A thread can have one *active* context at a time. However, the active context of a thread can be switched any time. Contexts consequently constitute dynamic scopes: `aContext do: [...]`.

The interpreter or virtual machine has an intimate knowledge of contexts like classes or other internal abstractions. Contexts are however explicit in our model and reified as *first-class* entities at the application level. Contexts can be instantiated and manipulated dynamically like other objects. When a new thread is created, it inherits the context of its parent thread, which will be the *active* context for that thread of execution as soon as it starts running.

The class of an object is part of its state. Behavioral variations are therefore achieved by changing the class of the object between contexts, and scoping behavioral changes reduces to a special case of scoping state.

4.2 Transformations

As shown in Figure 2, contexts form a list at run-time. They must have an `ancestor`, and they must implement two methods `transformFromAncestor` and `transformToAncestor` that realize the bidirectional transformation.

The role of the bidirectional transformation is to maintain consistency between several representations of an object in various contexts. A change to an object in a context will propagate to its ancestor and successor—which in turn will propagate it further—so as to keep the representations of an object consistent in all contexts.

As contexts are loaded dynamically in an unanticipated fashion, the transformation is encoded in the newest context and expressed in terms of its ancestor, never in terms of its successor. We have one method to transform *from* the ancestor to the newest context, and another method to transform from the newest context *to* its ancestor. The *Root* context is the only context that does not encode any transformation.

A sample one-way transformation is shown in Figure 3. It corresponds to the transformation from the *Root* context to the *Updated* context of Figure 1. `self` refers to the *Updated* context, and `ancestor` to the *Root* context. Line 5 reads the `html` of the `Page` in the *Root* context. Lines 7–8 split the html into `body` and `header`, and the representation of the `Page` in the *Updated* context is updated accordingly in lines 11–12.

```
1.  transformFromAncestor: id
2.      | cls html body header |
3.      cls := ancestor readClassFor: id.
4.      ( cls = Page ) ifTrue: [
5.          html := ancestor readField: 'html' for: id.
6.          html isNil ifFalse: [
7.              body:= html regex: '<body>(.*)</body>'.
8.              header:= html regex: '<header>(.*)</header>'.
9.          ].
10.         self writeClassFor: id value: Page2.
11.         self writeField: 'body' for: id value: body.
12.         self writeField: 'header' for: id value: header.
13.     ]
14.     ( cls = AnotherClass ) ifTrue: [
15.         ...
16.     ]
17.     ...
```

Figure 3. State transfer—one-to-one mapping between two versions of a class.

4.3 Meta levels

Contexts are meta-objects that are causally connected with the runtime: field writes and object instantiations will be evaluated differently depending on the set of contexts and their corresponding transformations.

Before a context can be used, it must first be registered via `aContext register`. This establishes the causal connection between a context and the runtime. The registration will also create the new representation of all contextual objects in the newly registered context. After this step, the context can be used and only of objects that are created or modified will need to be synchronized later on. This way, all contextual objects have a valid representation in all existing contexts anytime.

Transformations are never called directly by the application, but by the run-time itself because of the causal connection. This corresponds to two distinct meta-levels, that we refer to as the *application* level and the *interpreter* level: user-written code runs at the application level, except for transformations that run at the interpreter level.

4.4 Primitive

Contexts are connected with the run-time, and their state is accessed by the interpreter or virtual machine itself (not only other application objects), which means contexts can't be contextual. If they were, then the code of the interpreter would also be contextual, and it would need to be interpreted by another interpreter. To avoid the infinitive meta-regression, code running at the interpreter level runs outside of any context. As a consequence, some objects in the system must be *primitive*: they have a unique state in the system and are not subject to contextual variations. Context objects are an example.

```
1.  transformFromAncestor: id
2.      | cls holder email body header |
3.      cls := ancestor readClassFor: id.
4.      ( cls = Contact ) ifTrue: [
5.          email := ancestor readField: 'email' for: id.
6.          email isNil ifFalse: [
7.              alias := email regex: '(.*)@'.
8.              host := email regex: '@(.*)'.
9.              self interpret: [
10.                 holder := Email new.
11.                 holder alias: alias.
12.                 holder host: host.
13.                 holder contact: id.
13.             ].
14.         ].
15.         self writeClassFor: id value: Contact2.
16.         self writeField: 'email' for: id value: holder.
17.     ]
18.     ( cls = AnotherClass ) ifTrue: [
19.         ...
20.     ]
21.     ...
```

Figure 4. State transfer—refactoring and usage of the `interpret` keyword to switch between levels.

4.5 Mirror

The fact that transformations run at the interpreter level, outside of any context, implies that one can send messages only to primitive objects in the transformations. Contextual objects must be manipulated *reflectively* via a context with `readClassFor:`, `writeClassFor:value:`, `readField:for:` and `writeField:for:value:` as shown in Figure 3. With these language constructs, a context acts as a mirror [3] that reifies the state of an object in this particular context. This way, the state of an object (class or fields) in an arbitrary context can be updated without subsequent transformations being triggered, and independently of the active context.

4.6 Interpretation

In complex transformations, it can be necessary to evaluate a whole block in a given context to manipulate contextual objects. This can be achieved with `aContext interpret: [...]`, which will evaluate the block in the given context as if it was executed at the application level.

Unlike with mirrors, subsequent transformations will be triggered in this case when contextual objects are modified. This is necessary in particular to instantiate a new contextual object from within a transformation. If subsequent transformations were not triggered, the new object would be missing representations in other contexts.

The evaluation of `do:` and `interpret:` are similar. The difference between `do:` and `interpret:` is that `do:` expects to switch from a current context to another and must be called from the application level, while `interpret:` must be called from the interpreter level, and expects that there is no current context.

Transformations can be more complex than one-to-one mappings and Figure 4 shows the usage of `interpret:` in the case of the refactoring of an email string of the form "alias@host" into a dedicated `Email` holder object with field `alias` and `host` (inspired by a case found in practice [28]). Code between lines 10–13 runs at the application level in the context referred by `self`, and line 10 instantiates a new contextual object.

Keyword and description
// Creates the causal connection between a context and the runtime
`aCtx register`
// Evaluate the block in the given context (used from the interpreter level)
`aCtx interpret: [ ... ]`
// Evaluate the block in the given context (used from the application level)
`aCtx do: [ ... ]`
// Read a value from a field reflectively
`aCtx readField: aFieldName for: aCtxObj`
// Write a value to a field reflectively
`aCtx writeField: aFieldName for: aCtxObj value: aValue`
// Read the class reflectively
`aCtx readClassFor: aCtxObj`
// Write the class reflectively
`aCtx writeClassFor: aCtxObj value: aClass`

Table 1. Keywords to manipulate contexts and contextual objects.

4.7 The big picture

Table 1 recapitulates the syntax to deal with contexts and contextual objects, and Figure 5 visually summarizes the abstractions presented earlier.

5. Implementation

We report on the implementation of ActiveContext in Pinocchio [31][2], an experimental Smalltalk system designed to enable invasive changes to the runtime system. Interpreters are first-class in Pinocchio. The default meta-circular interpreter of Pinocchio can be subclassed to create custom interpreters and experiment with programming language variations. Pinocchio is otherwise similar to conventional Smalltalk systems. It supports in particular the object model and reflective architecture of Smalltalk 80.

5.1 Implementation Details

Pinocchio represents code as abstract syntax trees, or ASTs. Code is evaluated by an interpreter that visits nodes of the AST. The state of a Pinocchio object is stored in slots (first-class fields), which are represented as AST nodes.

Table 2 shows the relevant visitor methods of the interpreter, and indicates which ones were overridden to implement ActiveContext. The ActiveContext interpreter changes the way state and memory is managed, in particular the treatment of slots and message sends, whose corresponding visit methods have been overridden accordingly. Only contextual objects are treated specially. Primitive objects delegate to the native memory management of Pinocchio to avoid any performance overhead. A similar decision was taken for the

[2] `http://scg.unibe.ch/pinocchio`

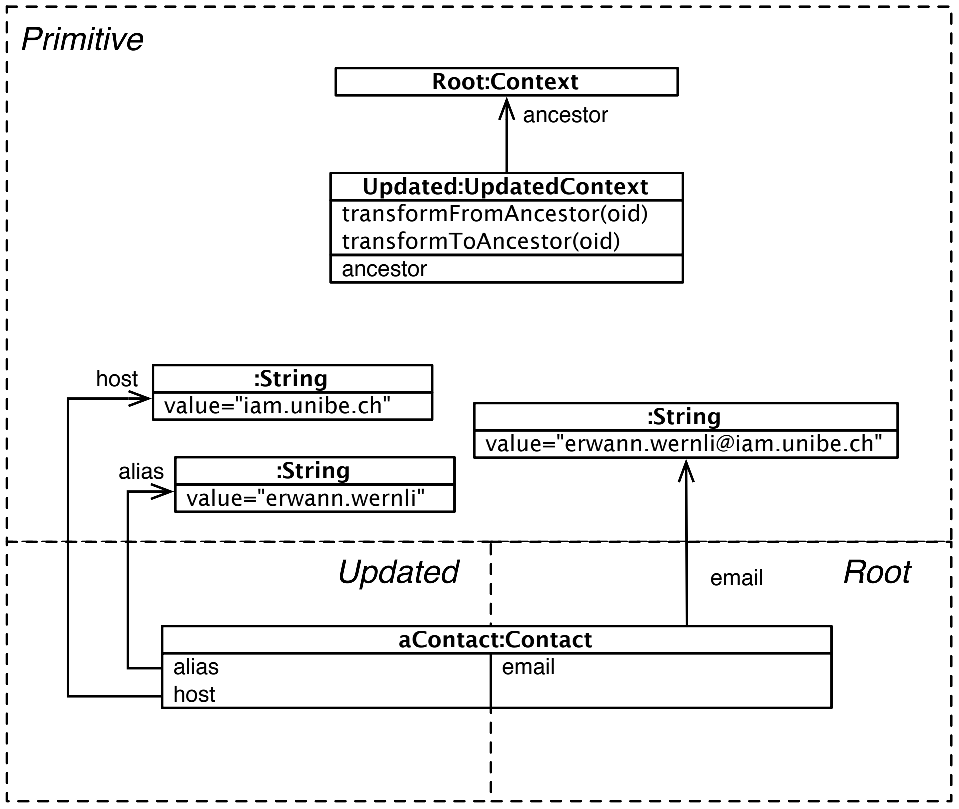


Figure 5. Conceptual overview of the system at run-time. It shows a system with the *Root* context, the *Updated* context, three primitive `String` objects, and one contextual `Contact` object. The memory is conceptually divided into three segments, one for primitive objects, one for the objects' representation in the *Root* context, and one for objects' representation in the *Updated* context.

Visitor method	Overriden
`visitConstant: aConstant`	·
`visitClassReference: aClassReference`	·
`visitVariable: aVariable`	·
`visitSlot: aSlot`	✓
`assignVariable: aVariable to: value`	·
`assignSlot: aSlot to: value`	✓
`visitAssign: anAssign`	·
`visitSelf: aSelf`	·
`visitSend: aSend`	✓
`visitSuper: aSuper`	✓

Table 2. The visitor methods of the interpreter.

Root context, which also delegates to the native memory management even for contextual objects.

Internally, the interpreter uses several `Dictionary` instances to implement the memory model for contextual objects: one dictionary per registered context is created and maps {`object identity, field`} to the corresponding value. A special field `class` is used for the class of the object. This implies one level of indirection to access the state of a contextual object.

The active context is referenced in a field of the interpreter and each thread has a dedicated instance of the interpreter. To distinguish between primitive or contextual objects, we maintain two pools of references represented internally with two `Sets` (this is admittedly a naive approach: tagging the pointer would be faster).

The set of primitive objects was adapted to match the reality of a fully reflective system. `Object`, `Behaviour`, `Class`, `Metaclass` and other special classes needed during bootstrapping cannot be contextual, as they need to exist in order for the `BaseContext` class to be defined, so for them to be contextual would lead to a chicken-and-egg problem. The same also holds true for basic, immutable objects like `nil`, `true` and `false`, as well as numbers, characters, strings and symbols.

The keywords in Table 1 have been implemented with regular message sends that the interpreter handles in a special way.

5.2 Demonstration

To validate our approach, we have implemented a canonical server-side application that illustrates our approach and follows the use case of section 3. The application we im-

		new	**send**	**read**	**write**
Primitive	Metacircular	191	215	137	137
	Root	393	297	215	219
	Updated	492	294	217	283
Contextual	Metacircular	192	146	137	200
	Root	347	229	292	294
	Updated	**935**	341	259	**598**

Table 3. Benchmark—Milliseconds for 500 executions of specific instructions for contextual and primitive objects using the meta-circular interpreter and the ActiveContext interpreter with one context (*Root*) and two contexts (*Root* and *Updated*).

plemented is a Telnet server. A client can connect to the server and run a few simple commands to update a list of contacts stored in memory. While simpler than a web-based Content Management system, such a system exhibits comparable characteristics in term of design and difficulties with respect to dynamic updates.

The telnet server was adapted according to step *Preparation* in section 3. First, a global variable `latestContext` was introduced. Second, the main loop that listens for incoming TCP connections was modified so that when a new connection is accepted, the corresponding thread that is spawned to handle the connection is executed in the latest context. Third, a client connects to the server and usees special commands to upload code and "push" an update.

The system can then be bootstrapped and runs initially in the *Root* context. An administrator can connect to the server and use a special command to upload an update. The classes implementing the logic to process commands can in particular be changed to change the logic of an existing command, or to add new ones. All commands executed as well as their versions are logged in a file. A thread handling a specific client connection keeps running as long as the connection is established. Already connected clients that use the original version are not impacted by the update and multiple clients connected to the server might see different versions of the command-line interface.

When a client disconnects, its server-side thread terminates. The system stabilizes eventually when all clients have disconnected. The log shows the various commands executed over the time and the migration of the server from one version to another.

We benchmarked object creation, message send, field read and field write for primitive and contextual objects under the three configurations of the system that are described in Table 3.

The meta-circular interpreter serves as the baseline for comparison. When running the benchmark with this interpreter, contextual and primitive objects are treated in the same way and results are then similar. When running the benchmark with the ActiveContext interpreter with solely the *Root* context, our implementation delegates to the native memory for primitive and contextual object. Results for both kinds of object are in the same range, but slower than on the meta-circular interpreter due to the overhead of our interpreter. When running the benchmark with the ActiveContext interpreter and two contexts (*Root* and *Updated*), we perceive a small performance drop for primitive objects, but a significative performance drop for contextual objects, notably for operations new and write (in bold in the table). This can be explained easily: (1) send and read operations for contextual objects need to look up data in internal dictionaries, and (2) in addition to the lookup in dictionaries, operations new and write need to trigger transformations to synchronize the data in the *Root* context.

Looking at these results, the worst performance drop is in the range of a factor 5 for contextual object creation (935 ms vs. 191 ms). These results will vary depending on the structure of the objects, the number of objects created and maintained in the pool of references, and the number of registered contexts that need to be synchronized, as well as the complexity of the transformations. This suffices however to estimate the maximum performance degradation to one order of magnitude. We believe this performance degradation can be reduced by a smarter implementation. We consider this to be a validation of the conceptual contribution and a positive feasibility study.

5.3 Assessment of ActiveContext

Let us assess ActiveContext according to the safety, timeliness, and practicality, as we did for vanilla Smalltalk in section 2:

Safety. Custom state transfer can be specified to transition from one version to the other. Code running in a given version will not produce run-time type errors due to dynamic updates. Contexts also help address safety that is beyond typing errors: it provides version consistency. Contexts enable the atomic installation of co-related changes to class, and ensure that code running in a context always corresponds to one precise version of the software.

Timeliness. If the synchronization is performed lazily, the creation of a new software version entails no overhead, and it can be used immediately after creation.

Practicality. Contexts are simple abstractions that are easy to use. They extend the language and do not impose restrictions. Writing the transformations manually is extra work, but it is acceptable if updates are not too frequent, *e.g.*, during maintenance phase. The overhead for synchronizing objects is significant, but it can be dramatically improved by (1) synchronizing only objects that are actually shared, and (2) synchronizing lazily.

ActiveContext extends the reflective architecture with features that enable the update of production system safely.

6. Future work

This paper presents a conceptual model for systems to support dynamic software updates, as well as a prototype to demonstrate the soundness of the approach. Several further points would need to be considered in a full implementation:

Lazy transformation and garbage collection The model that we have presented and implemented uses eager transformations: the state of objects is synchronized after each write. This entails significant overhead for objects whose lifetime is short, and are never accessed from another context than the one in which they were created. This also entails high memory consumption as we keep as many representations for an object as we have contexts. All context instances are connected to each other in a list which prevents them from being garbage collected. With eager transformations, long-lived objects consume more and more memory and become slower and slower to synchronize.

More appealing are *lazy transformations*: instead of synchronizing their state eagerly on write, it is synchronized lazily on read, in a way similar to how caches work. Not only would this reduce the performance overhead, but also reduce memory consumption as only the most up-to-date representation would be kept in memory. There should be a significant overhead only for objects whose structure has changed and has been accessed from several contexts.

Keeping only the most up-to-date representation assumes that the transformation is *lossless*, that is, one representation can be computed out of another one without loss of data. This is not always the case, *e.g.*, in case of field addition or removal with no counterpart in the other context. Such transformations are said to be *lossy*. One idea would be to track which transformations are lossy or not, and only keep multiple versions of objects impacted by lossy transformations.

We plan to implement lazy transformations, to distinguish between lossy and lossless transformations for further optimizations, and to enable garbage collection of unused contexts using weak references in our implementation.

Interdependent class evolution The object graph can be navigated during the transformation, which makes our approach very flexible to support arbitrary forms of evolution and interdependent class evolution, as was shown in the refactoring of Figure 4. Other approaches with similar facilities to navigate the object graph proved to support most scenarios of evolution in practice [2, 20, 28]. Keeping several versions of objects in memory is necessary until an update has been installed completely [2]. This puts memory pressure on the system, regardless of whether the transformations happen lazily or eagerly. One promising aspect of our approach with bi-directional transformations is that the old representation can in principle be recovered at any time; we could avoid keeping multiple representations (at least for objects subject to lossless transformations) and thus relieve the memory pressure.

Versioning of class hierarchies In our current implementation, classes are not contextual objects and this implies that two versions of a class have distinct names across contexts (see line 10 in Figure 3). In a more elaborate implementation, the same class name could be used and would resolve to a different representation of the class. The contextual class state would include `methodDict` and `super`. This would enable the fine-grained evolution of class hierarchies: the superclass of a class could differ in two contexts (without the subclass being modified), and conversely, two versions of a subclass could have different superclasses in two contexts. Metaclasses could possibly also be contextual but some classes would need to be primitive and would not be resolved contextually, for the same reasons that we distinguish between primitive objects and contextual objects (see subsection 4.3).

7. Conclusion

We have presented a novel approach to dynamically update software systems written in dynamic languages. ActiveContext is a programming model that extends the reflective capabilities of a dynamic language with first-class contexts to support the coexistence and synchronization of alternative representations of objects in memory. With ActiveContext, existing threads run to termination in the old context while new threads run in a new context. Program state will eventually migrate from the old to the new context, and during the transition period the state will be synchronized between contexts with the help of bi-directional transformations. We showed that ActiveContext is safe, practical, and timely. It empowers the developer with more control over dynamic updates, and does not require that the system be quiescent to be updated. We have demonstrated how to build a dynamically updatable system with a typical use case. The next step is to introduce lazy transformation and enable garbage collection, which should improve performance and further reduce memory consumption.

Acknowledgments

We would like to thank Mircea Lungu, Adrian Lienhard, Niko Schwarz and Toon Verwaest for kindly reviewing 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] A. Bergel. *Classboxes — Controlling Visibility of Class Extensions*. PhD thesis, University of Bern, Nov. 2005. URL `http://scg.unibe.ch/archive/phd/bergel-phd.pdf`.

[2] C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. *SIGPLAN Not.*, 38(11):403–417, 2003. ISSN 0362-1340. doi: 10.1145/949343.949341. URL `10.1145/949343.949341`.

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

[4] G. Casaccio, D. Pollet, M. Denker, and S. Ducasse. Object spaces for safe image surgery. In *IWST '09: Proceedings of the International Workshop on Smalltalk Technologies*, pages 77–81, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-899-5. doi: 10.1145/1735935.1735948.

[5] S. Cech Previtali and T. R. Gross. Aspect-based dynamic software updating: a model and its empirical evaluation. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 105–116, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0605-8. doi: 10.1145/1960275.1960289. URL `http://doi.acm.org/10.1145/1960275.1960289`.

[6] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew. Polus: A powerful live updating system. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 271–281, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. doi: 10.1109/ICSE.2007.65.

[7] M. Denker, T. Gîrba, A. Lienhard, O. Nierstrasz, L. Renggli, and P. Zumkehr. Encapsulating and exploiting change with Changeboxes. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 25–49. ACM Digital Library, 2007. ISBN 978-1-60558-084-5. doi: 10.1145/1352678.1352681. URL `http://scg.unibe.ch/archive/papers/Denk07cChangeboxes.pdf`.

[8] M. Dmitriev. Towards flexible and safe technology for run-time evolution of Java language applications. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001*, Oct. 2001.

[9] D. Duggan. Type-based hot swapping of running modules. In *Intl. Conf. on Functional Programming*, pages 62–73, 2001.

[10] Gemstone. Gemstone/s programming guide, 2007. URL `http://seaside.gemstone.com/docs/GS64-ProgGuide-2.2.pdf`.

[11] A. R. Gregersen and B. N. Jørgensen. Dynamic update of Java applications — balancing change flexibility vs programming transparency. *J. Softw. Maint. Evol.*, 21:81–112, mar 2009. ISSN 1532-060X. doi: 10.1002/smr.v21:2. URL `http://portal.acm.org/citation.cfm?id=1526497.1526501`.

[12] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Trans. Softw. Eng.*, 22(2):120–131, 1996. ISSN 0098-5589. doi: 10.1109/32.485222. URL `http://portal.acm.org/citation.cfm?id=229583.229586`.

[13] S. Herrmann, S. Herrmann, C. Hundt, C. Hundt, K. Mehner, and K. Mehner. Translation polymorphism in object teams. Technical report, Technical University Berlin, 2004.

[14] M. Hicks and S. Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 27(6): 1049–1096, nov 2005. doi: 10.1145/1108970.1108971.

[15] G. Hjálmtýsson and R. Gray. Dynamic C++ classes: a lightweight mechanism to update code in a running program. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '98, pages 6–6, Berkeley, CA, USA, 1998. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1268256.1268262`.

[16] J. Kabanov. Jrebel tool demo. *Electron. Notes Theor. Comput. Sci.*, 264:51–57, feb 2011. ISSN 1571-0661. doi: 10.1016/j.entcs.2011.02.005. URL `http://dx.doi.org/10.1016/j.entcs.2011.02.005`.

[17] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of OOPSLA '98, ACM SIGPLAN Notices*, pages 36–44, 1998. doi: 10.1145/286936.286945.

[18] O. L. Madsen and B. Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 397–406, Oct. 1989.

[19] K. Makris and R. A. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 31–31, Berkeley, CA, USA, 2009. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1855807.1855838`.

[20] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 337–361. Springer-Verlag, 2000. ISBN 3-540-67660-0. doi: 10.1007/3-540-45102-1_17.

[21] I. Neamtiu and M. Hicks. Safe and timely updates to multi-threaded programs. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 13–24, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1543135.1542479.

[22] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 72–83, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: 10.1145/1133981.1133991. URL `http://doi.acm.org/10.1145/1133981.1133991`.

[23] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. *SIGPLAN Not.*, 43(1):37–49, 2008. ISSN 0362-1340. doi: 10.1145/1328897.1328447.

[24] A. Orso, A. Rao, and M. Harrold. A Technique for Dynamic Updating of Java Software. *Software Maintenance, IEEE International Conference on*, 0:0649+, 2002. doi: 10.1109/ICSM.2002.1167829. URL `http://dx.doi.org/10.1109/ICSM.2002.1167829`.

[25] M. Piccioni, M. Oriol, B. Meyer, and T. Schneider. An ide-based, integrated solution to schema evolution of object-

oriented software. In *ASE*, pages 650–654, 2009.

[26] B. Redmond and V. Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of European Conference on Object-Oriented Programming*, volume 2374, pages 205–230. Springer-Verlag, 2002. doi: 10.1007/3-540-47993-7_9.

[27] scala. The scala programming language. URL `http://lamp.epfl.ch/scala/`. http://lamp.epfl.ch/scala/.

[28] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates: a VM-centric approach. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 1–12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542478. URL `http://doi.acm.org/10.1145/1542476.1542478`.

[29] E. Tanter. Contextual values. In *Proceedings of the 2008 symposium on Dynamic languages*, DLS '08, pages 3:1–3:10, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-270-2. doi: 10.1145/1408681.1408684. URL `http://doi.acm.org/10.1145/1408681.1408684`.

[30] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. Influence of type systems on dynamic software evolution. CW Reports CW415, Department of Computer Science, K.U.Leuven, Leuven, Belgium, 2005. URL `https://lirias.kuleuven.be/handle/123456789/131703`.

[31] T. Verwaest, C. Bruni, D. Gurtner, A. Lienhard, and O. Nierstrasz. Pinocchio: Bringing reflection to life with first-class interpreters. In *OOPSLA Onward! '10*, 2010. doi: 10.1145/1869459.1869522. URL `http://scg.unibe.ch/archive/papers/Verw10aPinocchio.pdf`.

[32] A. Warth, M. Stanojević, and T. Millstein. Statically scoped object adaptation with expanders. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 37–56, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-348-4. doi: 10.1145/1167473.1167477.

[33] T. Würthinger, C. Wimmer, and L. Stadler. Dynamic code evolution for Java. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 10–19, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0269-2. doi: 10.1145/1852761.1852764.

A Smalltalk implementation of Exil, a Component-based Programming Language

Petr Spacek   Christophe Dony
Chouki Tibermacine
LIRMM, CNRS and Montpellier II University
161, rue Ada
34392 Montpellier Cedex 5 France
{spacek,dony,tibermacin}@lirmm.fr

Luc Fabresse
Université Lille Nord de France
941 rue Charles Bourseul
59508 DOUAI Cedex France
luc.fabresse@mines-douai.fr

Abstract

The component-based development paradigm brings various solutions for software reusability and better modular structure of applications. When applied in programming language context it changes the way dependencies and connections between software pieces are expressed. In this paper we present the Smalltalk implementation of "Exil", a component-based architecture description and programming language that makes it possible to use component related concepts (ports, interfaces, services, ...) at design and if wished at programming time. This proposal enables Smalltalk users to develop their applications in the component-oriented style.

Categories and Subject Descriptors D.1.0 [*Programming techniques*]: General—Component oriented programming technique; D.2.11 [*Software Architectures*]: Languages—Component Oriented Language

General Terms Component-based Programming

Keywords Component, Inheritance, Architectures, Programming, Substitutability

1. Introduction

In this work, we consider a software component as a piece of software which is a unit of deployment and composition with contractually specified interfaces and explicit dependencies. A component interacts with other ones only by well declared communication channels.

We have designed a programming language where a component is a basic concept to encapsulate data and functionality in similar way as an object does, but with respect to component design ideas such as independence, explicit requirements and architectures. The goal is to develop a language, in which an expert programmer can develop independent components, design for reuse [5], and a non expert programmer can develop applications by connecting previously developed components, design by reuse [5].

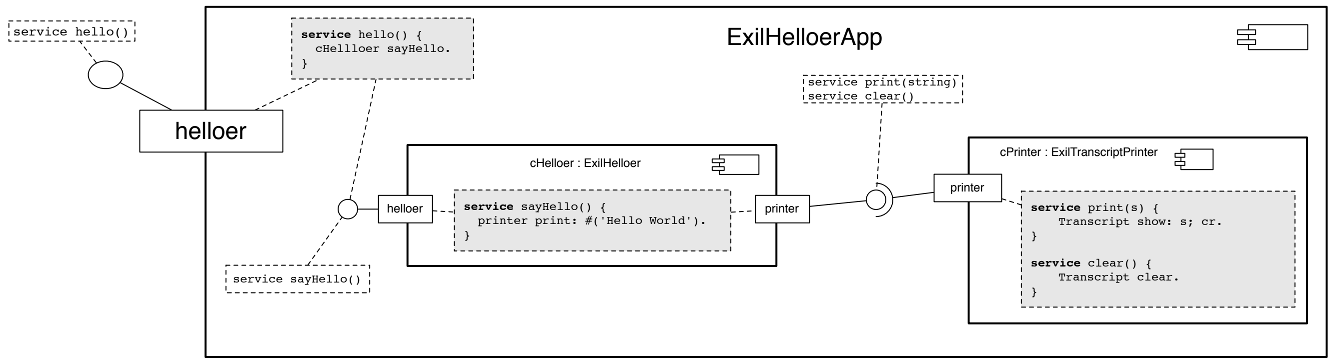
A component can be seen as a black-box which provides functionalities and explicitly expresses what it requires to be able to provide them. With OOP, objects usually require some other objects to be able provide services, for example a compiler class usually requires a parser, see Figure 1. This dependency is expressed by the public attribute p and by Compiler's constructor, see lines 2 and 4 of figure 1). From reuse and deployment point of view, object dependencies are not very well observable from the outside (except by using reflective means or somehow by reading a documentation if there is one). With Component-based programming (CBP), where component dependencies are explicit, it is clear what the deployment environment will need to provide so that the components can function, as illustrated on Figure 2, lines 2 and 14.

```
01 class Compiler {
02   public Parser p;
03
04   public Compiler(Parser p) {}
05   ...
06 }
07 class App {
08   void main(string[] args) {
09     Compiler c = new Compiler();
10     c.p = new SpecialParser();
11     // or Compiler c = new Compiler(new SpecialParser());
12     ...
13   }
14 }
```

Figure 1. The `Compiler` class declares `Parser` attribute, from the black-box viewpoint, this requirement is hidden to the user

When a component is connected to another one, generally to satisfy requirements, this defines a software architecture. A software architecture is an overall design of software system [10]. The design is expressed as a collection of components, connections between the components, and constraints on how the components interact. Describing architecture explicitly at the programming language level can facilitate the implementation and evolution of large software systems and can aid in the specification and analysis of high-level designs. For example, a system's architecture can show which components a module may interact with, help to identify the components involved in a change, and describe system invariants that should be respected during software evolution.

The work we present in this paper aims at proposing a Smalltalk implementation of a dynamically typed component-based programming language named Exil. Exil is based on Scl [4, 5] and extends it towards the explicit and declarative expression of requirements and architectures. It is based on the descriptor/instance dichotomy where components are instances of descriptors. It also provides an original inheritance system [9] which is not the scope of this paper.



```
service hello() {
  cHellloer sayHello.
}
```

```
service sayHello() {
  printer print: #('Hello World').
}
```

```
service print(s) {
    Transcript show: s; cr.
}

service clear() {
    Transcript clear.
}
```

Figure 3. Diagram of the architecture of the *ExilHelloerApp* component

```
component Compiler {
  require Parser p ;
  ...
}
...
component App {
  provide service main(string[] args);

  internalComponents {
    Compiler c;
    Parser p;
  }
  connections {
    connect c.requirements.p to p.default
  }

  service main(string[] args) {
    ...
  }
}
```

Figure 2. Component-oriented languages explicitly express dependencies, i.e. component `Compiler` require a `Parser`

The paper is organized as follows. Section 2 proposes an overview of the Exil component model and programming language. Section 3 gives some basic clues on the Smalltalk implementation. Before concluding and discussing the future work, we briefly present in Section 4 the related work.

2. Exil Overview

This section presents the key concepts and structure of Exil.

In Exil, every component is an instance of a *descriptor*. A descriptor defines the behavior of its instances by *services* and their structure through *ports*, *internal components* and *connections*. The diagram of a simple descriptor is presented on Figure 3. *ExilHelloerApp* is a descriptor of a component saying hello, the Exil code of such a component is shown on Figure 4. It defines that the `ExilHelloerApp` component will maintain only one provided port called `helloer` providing service `hello`. The architecture of this component (field *internalComponents*) declares 2 internal components: `cHelloer` and `cPrinter`, and in the field *internalConnections* it defines how they are connected.

A descriptor may extend another descriptor, such a descriptor is called a *sub-descriptor*. A sub-descriptor inherits all ports, internal components and connection of super-descriptor (its parent), it may add new ports, new internal components and new connections or it may specialize them. A sub-descriptor may specialize an inherited port by modifying its interface. It may specialize an inherited internal component by modifying its default descriptor and finally may specialize connections by combination of `connect` and `disconnect` statements. A sub-descriptor may specialize an inherited service. Using classical mechanism of inheritance we do have common problems such as encapsulation violation between the components. More over, allowing additional requirements in a sub-descriptor the substitutability becomes more complicated [9]. However, modeling assets brought by inheritance are one major cornerstone of the success of object-oriented languages (1) for the ability it gives developers to organize their ideas on the base of concept classification (a list is a kind of collection, such architecture is a kind of visitor, ...) which is itself one key of human abstraction power and (2) for the calculus model introduced by object-oriented programming that makes it possible to execute an inherited code in the correct context (the receiver environment). In the off-the-shelf components context, as pointed by [6], a set of available black-box components cannot cover all possible scenarios of usage, and therefore an adaptation mechanism is needed. Inheritance can be the mechanism, which enables programmers to easily extend or specialize such components.

Services A descriptor may introduce services to specify functionality of its instances. When a service is listed in a provided port description, then the service is public. Each service has a signature given by the following template: `<service-name> (<argument1-name>, <argument2-name>, ... )`. A definition of a service consist of the `service` keyword followed by the *service signature* and a source code written in brackets after the signature, for example service `hello` in Figure 4.

The syntax of Exil is a mix of java-like syntax, used for specifying descriptors, and Smalltalk syntax used for service bodies implementation - this dichotomy is motivated by the fact that our language is currently implemented in the Pharo-Smalltalk environment [1], but we consider java-like syntax more readable and expressive for structural descriptions.

Components communicate by *service invocations* through their ports. A service invocation consists of a port, a selector (the name of the requested service) and arguments. Arguments are treated as in Scl, by temporary connections between arguments and ports temporary created for each argument. In case of argument incompatibility an exception is thrown.

A component c_1 can invoke a service of a component c_2 if a provided port p_2 of c_2 is connected to a required port p_1 of c_1. In this case, the service invocation is emitted via p_1 of c_1 and received via p_2 of c_2. When a component sends a service invocation i via one of its required ports r, the component checks if port r is connected to provided port p and if yes, then it transmits i via p, else a *does-not-understood* error is thrown.

```
component ExilHelloerApp {
  provide { helloer->{hello()} }
  internalComponents {
    cHelloer->ExilHelloer;
    cPrinter->ExilTranscriptPrinter;
  }

  internalConnections {
    connect cHelloer.printer to cPrinter.printer
  }

  service hello() { cHelloer sayHello }
}
```

Figure 4. Exil code of the `ExilHelloerApp` descriptor providing and implementing the `hello` service and having four internal components `cHelloer` and `cPrinter` inter-connected by connections specified in the *internalConnections* field

Interface An interface in Exil is a named list of service signatures. An interface is created by a statement with the following template: `interface <interface-name> { <service-signature-1>; <service-signature-2>; ...}`. We have introduced interfaces in Exil compared to Scl for convenience and reuse purposes. That means, we want to be able to reuse a list of services, used as a contract description by one component, as a contract description of another component.

Port A port is an unidirectional communication point described by a list of services signatures or by an interface reference (which makes it possible to reuse such a description) and by a role (*required* or *provided*). A provided port describes what is offered by a component. A required port describes what is demanded by a component. For example, a definition of two provided ports named `A` and `B` looks like: `provide {A->{service1()}; B->ISecond}`, where the `A` port provides a service called `service1` and the `B` port is described by an interface `ISecond`.

Connection A connection between two components is performed by a connection between their ports. A descriptor lists all connections in the field *internalConnections*. A connection is specified by a statement described by the template
`connect <an-emittor-port-address> to <a-receiver-port-address>`. By *the port-address* it is meant the expression `<component-name>.<port-name>`, for example see the field `internalConnections` in Figure 4. A component can act as an adapter between various other components and then, it is called a *connector*.

Internal component A component can own internal components. Such a component is then called *a composite*. The owning component references an internal component by a variable.

A list of internal components is defined in the descriptor's field *internalComponents*, see Figure 4. The Exil code of the `cHelloer` internal component is in Figure 5. Internal components are initialized during instantiation of the owning composite. By default all internal components are initialized with `NilComponent` component, a developer should implement an `init` service or optionally may specify a default descriptor in the internal components list, i.e. use the following statement `internalComponents {cPrinter->Printer}`, which is equivalent to the `cPrinter := Printer new.` line in the `init` service.

An internal component is encapsulated by the owning composite and it is not accessible from outside of the composite. Services defined by a composite can use internal components to implement a desired behavior (service invocation redirect).

```
component ExilHelloer {
  provide { helloer->{sayHello()} }
  require { printer->{print(string); clear()} }

  service sayHello() {
    printer print: #('Hello World').
  }
}
```

Figure 5. Exil code of the `ExilHelloer` descriptor providing and implementing the `sayHello` service and requiring services `print` and `clear` via required port `printer`.

3. Implementation

Exil is implemented in the Pharo Smalltalk environment [1] as an extension of Scl. We chose Smalltalk because of its reflective capabilities, which are necessary for mechanisms like the service invocation mechanism. And we chose Pharo environment for its rich set of support tools and frameworks like PetitParser [8] framework or Mondrian [7], which we use or will use for the Exil implementation.

The Exil implementation contains core classes representing Exil component model, then there are parser and compiler classes responsible for source code processing and classes implementing Exil GUI shown on Figures 6 and 7.

3.1 Parser & Compiler

Exil has custom syntax and therefore a special parser is required. We use PetitParser framework base for our parser which is represented by the `ExilParser` class, which inherits `ExilGrammar` class and extends it with AST building code. We have chosen PetitParser framework, because it allows us to maintain Exil grammar as easily as common Smalltalk code and because we can smoothly compose it with the PetitSmalltalk parser. The PetitSmalltalk parser is used for service bodies parsing. The result of our parser is an AST made from subclasses of the `ExilParseNode` class.

`ExilCompiler` transform the AST made by the parser to Exil core classes representation. The compiler is designed as a visitor of AST nodes.

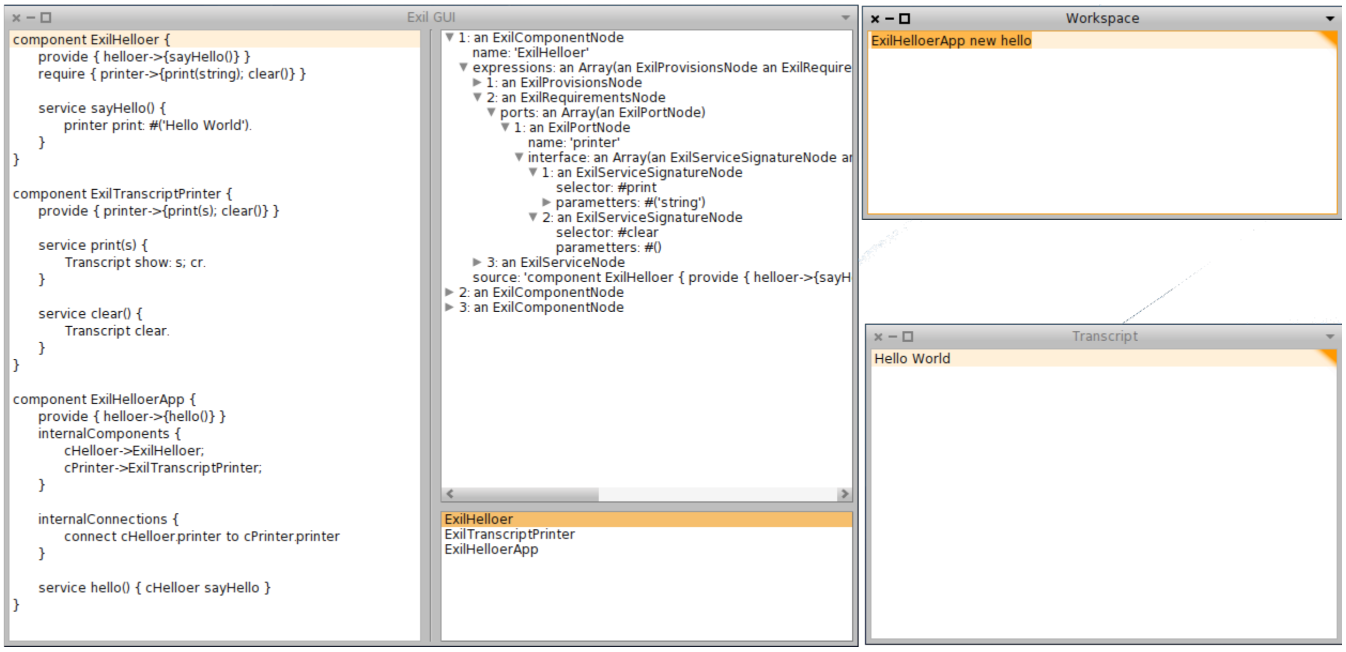
3.2 Core

In our proposal, we use Smalltalk classes to implement component descriptors. All classes implementing component descriptors are subclasses of the base class called `ExilComponent`, the base class the mechanism to store information about provided and required ports, internal components and their interconnections contains in the class-side. This information is used at instantiation time by descriptors (initialization methods) to create components (descriptors instances).

Internally, for each port and each internal component, an instance variable is created in the class implementing the descriptor to hold references to port instances and internal components instances. Ports are implemented as objects. There is one class hierarchy for provided ports and one for required ports, all of them are subclasses of `ExilPort` base class. Ports are described by interfaces, which are implemented by arrays or by classes.

An interface is implemented as an array of service signatures. When an interface is defined as the named interface, for example `interface IMemory { load(); save(data); }`, then the `IMemory` sub-class of the class `ExilInterface` with methods `load` and `save_data:` is created. These methods are having empty bodies. A sub-interface is then implemented as a sub-class of the class representing its parent.

Services are represented by methods. An automatic (and transparent for the user) mapping of a service signature from Exil to



```
component ExilHelloer {
    provide { helloer->{sayHello()} }
    require { printer->{print(string); clear()} }

    service sayHello() {
        printer print: #('Hello World').
    }
}

component ExilTranscriptPrinter {
    provide { printer->{print(s); clear()} }

    service print(s) {
        Transcript show: s; cr.
    }

    service clear() {
        Transcript clear.
    }
}

component ExilHelloerApp {
    provide { helloer->{hello()} }
    internalComponents {
        cHelloer->ExilHelloer;
        cPrinter->ExilTranscriptPrinter;
    }

    internalConnections {
        connect cHelloer.printer to cPrinter.printer
    }

    service hello() { cHelloer sayHello }
}
```

```
▼ 1: an ExilComponentNode
    name: 'ExilHelloer'
  ▼ expressions: an Array(an ExilProvisionsNode an ExilRequire
    ▶ 1: an ExilProvisionsNode
    ▼ 2: an ExilRequirementsNode
      ▼ ports: an Array(an ExilPortNode)
        ▼ 1: an ExilPortNode
            name: 'printer'
          ▼ interface: an Array(an ExilServiceSignatureNode ar
            ▼ 1: an ExilServiceSignatureNode
                selector: #print
              ▶ parametters: #('string')
            ▼ 2: an ExilServiceSignatureNode
                selector: #clear
                parametters: #()
    ▶ 3: an ExilServiceNode
    source: 'component ExilHelloer { provide { helloer->{sayH
▶ 2: an ExilComponentNode
▶ 3: an ExilComponentNode
```

```
ExilHelloer
ExilTranscriptPrinter
ExilHelloerApp
```

```
ExilHelloerApp new hello
```

```
Hello World
```

Figure 6. Exil GUI

a Smalltalk's method selector is based on naming convention, for example service signature `sum(numberA,numberB)` is mapped into a Smalltalk method with selector `sum_numberA:numberB:`. Since service bodies are written in Smalltalk, a call of the `sum` service performed in a body of an another service looks like `<receiver> sum:{ 1. 2 }`. The call is automatically dispatched to `sum_numberA:numberB:` method, according to parameters arity.

For each internal component and port there is an instance variable having the same name. Information about ports and associated interfaces or about default descriptors of internal components are stored as class side methods which return an array of associations, i.e. pairs of port name (resp. internal component name) and interface (resp. descriptor). Connections are stored similarly, as a class-side method which returns an array of associations. An association is a pair of port-addresses. We call these methods *the description methods*. For example the `ExilHelloerApp` descriptor shown in Figure 4 is implemented as a sub-class of the `ExilComponent` class named `ExilHelloerApp` having 3 instance variables, one named `helloer` representing the port *helloer* and two others representing internal components *cHelloer, cPrinter* and named in the same way as the internal components. The `ExilHelloerApp` metaclass implements four description methods called `providedPortsDescription`, `requiredPortsDescription`, `internalComponentsDescription` and `connectionsDescription`. Source code of the class is in Figure 8.

3.3 Inheritance implementation

A sub-descriptor is implemented as a sub-class of the class representing its super-descriptor. All these specializations are implemented as modifications of the description methods.

A service specialization is equal to the method overriding in Smalltalk. When a sub-descriptor specialize an inherited service, the corresponding method is then overridden in the sub-class realizing the sub-descriptor.

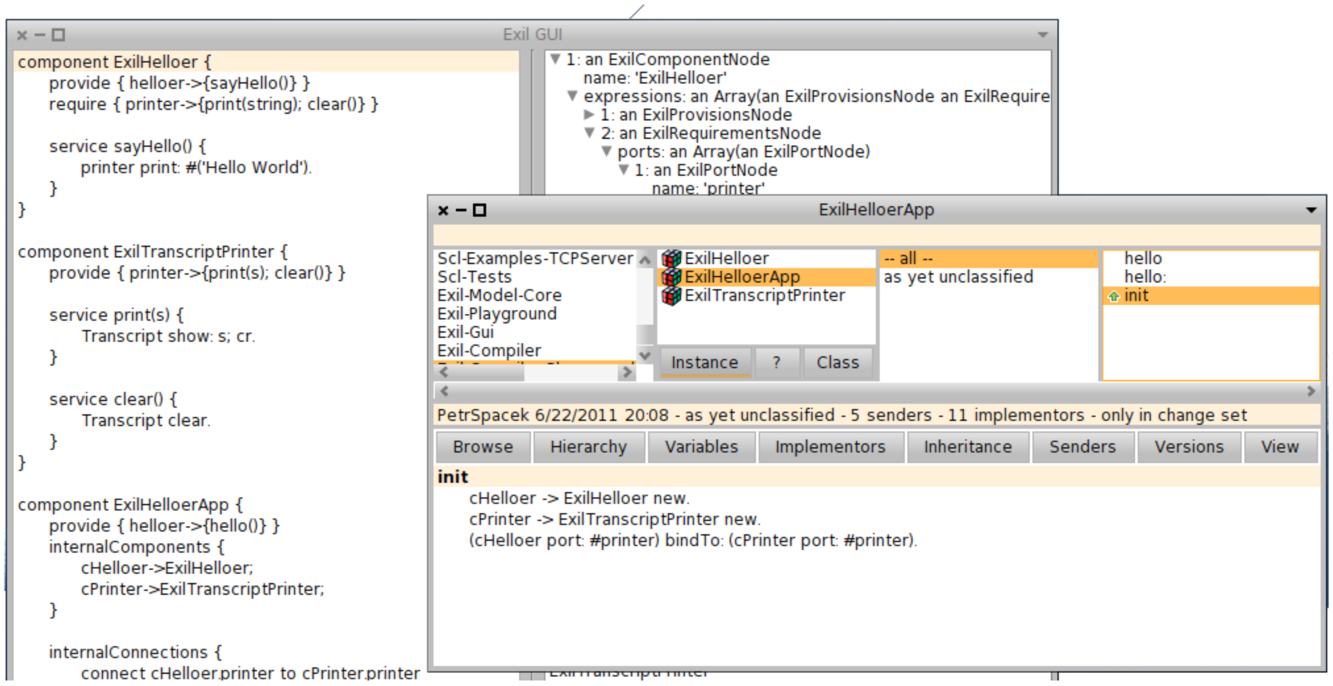
```
ExilComponent subclass: #ExilHelloerApp
  instanceVariableNames: 'helloer cHelloer cPrinter'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Exil-Examples-Helloer'.
!

ArithEvaluator class>>providedPortsDescription
  ^ #( #helloer->#(#hello) ).
!
ArithEvaluator class>>requiredPortsDescription
  ^ #().
!
ArithEvaluator class>>internalComponentsDescription
  ^ #( #cHelloer->ExilHelloer #cPrinter->ExilTranscriptPrinter ).
!
ArithEvaluator class>>connectionsDescription
  ^ #((#cHelloer->#printer)->(#cPrinter->#printer).
!
```

Figure 8. The ExilHelloerApp class implementing the `ExilHelloerApp` descriptor showed in Figure 4.

Service invocations Service invocations, in the context of inheritance, fully benefits from Smalltalk's message sending system. When a port receives a service invocation which is valid according to the contract specified by the interface of the port, it translates the service signature into a Smalltalk selector and the standard *method look-up* is performed. Since descriptors and sub-descriptors are realized by classes and subclasses, not extra mechanism is needed, the standard *method look-up* works perfectly.

Substitution and initialization/compatibility support Exil users are responsible for the `init` method implementation of a descriptor to achieve dependency injection, that is to say, for initializing variables referencing internal components used in the internal architecture. The first support comes in case when a sub-descriptor has an additional required port. Then our inheritance system automatically generates two new class instantiation methods (and two corresponding `init` methods, not described here), one `newCompatible` without parameter and `newCompatible:` having as unique param-



```
component ExilHelloer {
    provide { helloer->{sayHello()} }
    require { printer->{print(string); clear()} }

    service sayHello() {
        printer print: #('Hello World').
    }
}

component ExilTranscriptPrinter {
    provide { printer->{print(s); clear()} }

    service print(s) {
        Transcript show: s; cr.
    }

    service clear() {
        Transcript clear.
    }
}

component ExilHelloerApp {
    provide { helloer->{hello()} }
    internalComponents {
        cHelloer->ExilHelloer;
        cPrinter->ExilTranscriptPrinter;
    }

    internalConnections {
        connect cHelloer.printer to cPrinter.printer
```

```
▼ 1: an ExilComponentNode
    name: 'ExilHelloer'
  ▼ expressions: an Array(an ExilProvisionsNode an ExilRequire
    ▶ 1: an ExilProvisionsNode
    ▼ 2: an ExilRequirementsNode
      ▼ ports: an Array(an ExilPortNode)
        ▼ 1: an ExilPortNode
            name: 'printer'
```

```
Scl-Examples-TCPServer    ExilHelloer             -- all --               hello
Scl-Tests                 ExilHelloerApp          as yet unclassified     hello:
Exil-Model-Core           ExilTranscriptPrinter                           init
Exil-Playground
Exil-Gui
Exil-Compiler

PetrSpacek 6/22/2011 20:08 - as yet unclassified - 5 senders - 11 implementors - only in change set

init
    cHelloer -> ExilHelloer new.
    cPrinter -> ExilTranscriptPrinter new.
    (cHelloer port: #printer) bindTo: (cPrinter port: #printer).
```

Figure 7. Component classes in browser

eter an array of pairs *port-component*. The first one is able to create an instance, compatible for substitution with instances of super-descriptors, for which all additional requirements are satisfied by connections to default instances of the required component. The second one does the same thing but uses its argument for connecting additional required ports.

The second support is a method to achieve substitutions safely. Substitution is achieved by a method `replace:with:` on instance-side of the base class `ExilComponent`. This method takes two arguments, the first one is the name of the internal component variable referencing the component which should be replaced and the second argument is the replacement component. `replace:with:` checks for original and replacement components descriptors compatibility then it checks if all requirements would be satisfied after substitution. If the descriptor of the new component is compatible with the descriptor of the original one and if all requirements of the new component are about to be satisfied, the replacement is performed otherwise an exception is thrown. `replace:with:` reconnects all ports of the original component to corresponding ports of the new component and change internal component reference of the composite to reference the new component.

Readers can download a Pharo image of Exil implementation here: http://www.lirmm.fr/∼spacek/exil/

4. Related work

We give in this section an overview of existing component models implemented on top of Smalltalk, and discuss their limitations.

CLIC Clic [2], an extension of Smalltalk to support full-fledged components, which provides component features such as ports, attributes, or architecture. From the implementation point of view, it fully relied on Smalltalk reflective capabilities. Thus, from the Smalltalk virtual machine point of view, CLIC components are objects and their descriptors are extended Smalltalk classes. Because of this symbiosis between CLIC and Smalltalk, the use of CLIC allows taking benefit from modularity and reusability of components without sacrifice performance. CLIC model allows components to have only one provided port. The idea of a single provided port is based on the observation that developers do not know beforehand, which services will be specified by each required port of client component. Therefore it is hard to split component functionality over multiple ports. CLIC also support explicit architecture description and inheritance. It does not need any additional parser or compiler.

FracTalk FracTalk[1] is a Smalltalk implementation of the Fractal hierarchical component model [3]. In FracTalk, a primitive component is implemented as a plain object. As in Exil, every port is implemented as a single object in order to ensure that every port allow invoking only declared operations. Therefore, a single component materializes as multiple objets. In opposite to Exil, the description of a component is scattered over multiple classes. A component in FracTalk is described by implementation class and factory class. Another limitation of FracTalk is the the difficulty to make use of Smalltalk libraries. Smalltalk objects aren't full fledged components since they do not have a membrane and then does not provide expected non-functional ports. Therefore, the only mean to use a Smalltalk object in a FracTalk application is to encapsulate it in the content of some component.

5. Conclusions

In this paper, we propose a Smalltalk implementation for a dynamically-typed component-based language. The language brings benefits of the component-paradigm closer to the Smalltalk users and it also provides solid soil for experiments in component software area. Exil allows programmers to express architectural structure and then seamlessly fill in the implementation with Smalltalk code, resulting in a program structure that more closely matches the designer's conceptual architecture. Thus, Exil helps to promote

[1] http://vst.ensm-douai.fr/FracTalk

effective architecture-based design, implementation, program understanding, and evolution.

We plan to work in the near future on the integration of our previous work on architecture constraint specification and architecture description structural validation [11, 12]. In this way, we can specify conditions on the internal structure of a component (its internal components and connections between them) or on its ports. This will help developers in better designing their systems by making more precise architecture descriptions. There are here some interesting issues that we foresee to study, as for example, architecture constraint inheritance.

In the future, we would like to switch from Smalltalk syntax used for services implementation to Ruby syntax, which is more similar to the syntax used for component structure description. For this purposes we would like to port SmallRuby[2] project into Pharo and develop Ruby parser using PetitParser [8] framework. We are also interested in visual programming and we plan to use the Mondrian [7] framework to enhance our user interface with auto-generated component/architecture diagrams.

References

[1] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, 2009. ISBN 978-3-9523341-4-0. URL `http://pharobyexample.org`.

[2] N. Bouraqadi and L. Fabresse. Clic: a component model symbiotic with smalltalk. In *Proceedings of the International Workshop on Smalltalk Technologies*, IWST '09, pages 114–119, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-899-5.

[3] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36: 1257–1284, September 2006. ISSN 0038-0644. doi: 10.1002/spe.v36:11/12. URL `http://portal.acm.org/citation.cfm?id=1152333.1152345`.

[4] L. Fabresse. *From decoupling to unanticipated assembly of components: design and implementation of the component-oriented language Scl*. PhD thesis, Montpellier II University, Montpellier, France, December 2007.

[5] L. Fabresse, C. Dony, and M. Huchard. Foundations of a simple and unified component-oriented language. *Comput. Lang. Syst. Struct.*, 34:130–149, July 2008. ISSN 1477-8424. doi: 10.1016/j.cl.2007.05.002. URL `http://portal.acm.org/citation.cfm?id=1327541.1327717`.

[6] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995. ISBN 978-0-201-63361-0.

[7] M. Meyer and T. Gîrba. Mondrian: Scripting visualizations. European Smalltalk User Group 2006 Technology Innovation Awards, Aug. 2006. URL `http://scg.unibe.ch/archive/reports/Meye06cMondrian.pdf`. It received the 2nd prize.

[8] L. Renggli, S. Ducasse, T. Gîrba, and O. Nierstrasz. Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, Malaga, Spain, June 2010. URL `http://scg.unibe.ch/archive/papers/Reng10cDynamicGrammars.pdf`.

[9] P. Spacek, C. Dony, C. Tibermacine, and L. Fabresse. Reuse-oriented inheritance in a dynamically-typed component-based programming language. Technical report, LIRMM, University of Montpellier 2, May 2011.

[10] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002. ISBN 0201745720.

[11] C. Tibermacine, R. Fleurquin, and S. Sadou. A family of languages for architecture constraint specification. *In the Journal of Systems and Software (JSS), Elsevier*, 83(5):815–831, 2010.

[12] C. Tibermacine, S. Sadou, C. Dony, and L. Fabresse. Component-based specification of software architecture constraints. In *Proceedings of the 14th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE'11)*, Boulder, Colorado, USA, June 2011. ACM Press.

[2] https://swing.fit.cvut.cz/projects/smallruby

Klotz: An Agile 3D Visualization Engine

Ricardo Jacas   Alexandre Bergel
Pleiad Lab, Department of Computer Science (DCC),
University of Chile, Santiago, Chile

ricardo.jacas@gmail.com   http://bergel.eu

ABSTRACT

Klotz is an agile 3D visualization engine. Visualizations are produced from an arbitrary model described in terms of objects and interconnections. Any arbitrary model may be visualized. Klotz uses a semi-descriptive scripting language to easily and interactively build visualizations.

Klotz, on its current version, offers four layouts to easily exploit the third dimension when visualizing data.

Klotz is entirely implemented in Pharo. The engine is fully based on the facilities offered by Morphic.

1. INTRODUCTION

Visual displays allow the human brain to study multiple aspects of complex problems in parallel. Visualization, "allows for a higher level of abstract, a closer mapping to the problem domain" [3].

Numerous frameworks have been offered by the Smalltalk community to visualize data. Mondrian[1] [1], one of them, is a flexible and agile visualization engine that uses a two dimensional representation to visualize data.

We are building on the experience we gained with Mondrian by proposing a new visualization engine that adds a third dimension to the graphical representation. Klotz applies the main features of Mondrian, namely the scripting language and the interactive easel, to a new rendering engine.

Contrary to other Smalltalk 3D visualization engines (Lumière[2], Jun[2]), Klotz does not rely on OpenGL or any external libraries. The generation of 3D graphics is solely based on Morph facilities. The benefits are numerous, including ease of installation and multi-platform support.

The paper is structured as follows: Section 2 presents the essential characteristics of Klotz. It progressively presents Klotz' features by giving short and concise illustrative scripts. Section 3 briefly describes the main points of Klotz implementation and gives some benchmarks. Section 5 concludes.

[1]http://www.moosetechnology.org/seaside/pier/tools/mondrian
[2]http://aokilab.kyoto-su.ac.jp/jun/index.html

2. KLOTZ

2.1 Klotz in a nutshell

Klotz is an agile 3D visualization engine. Visualizations are made of cubes and lines. Contrary to other 3D visualization frameworks available on Smalltalk, Klotz maps each graphical element to an object that belongs to an user-defined domain. The visual dimensions of a graphical element is the result of applying metrics on the visualized domain.

Klotz' objective is to offer a flexible and agile tool to visualize any arbitrary domain expressed in terms of objects and relations without any prior preparation. Visualizations are described by means of a scripting language. Consider the illustrative script:

```
1 | subclasses |
2 subclasses := Magnitude subclasses.
3 view nodes: subclasses.
4 view applyLayout: KLSphereLayout new.
5
6 view node: KLEaselCommand using: (KLCube new
     fillColor: Color green).
7 view edges: subclasses from: #yourself to:
     #superclass.
```

Line 1 defines a temporary variable `subclasses`. The variable is initialized in Line 2 with the all the subclasses of the class `Magnitude`. Line 3 adds the objects referenced by `subclasses` into the view. Each of the subclass of `Magnitude` is represented by a cube. Line 4 positions each cube on an invisible sphere.

Line 6 adds a new node, `Magnitude`, the root of the class hierarchy. The node is colored in green. Line 7 adds as many edges there are elements in the variable `subclasses`. For each subclass, an edge is drawn from the subclass to its superclass, `Magnitude`. The result is depicted in Figure 1.

Klotz is intended to be suplementary 3D version of Mondrian, its graphic engine is not based on it tought, but built from the ground. The interface is as similar as it can be to Mondrian's, but its intention is not to bring the same visualizations to 3D figures, but to add new means to analyse code on another perspective, with another dimention to add more information on the object representation itself.

Agility of Klotz is expressed via an easel to interactively "compose" a visualization (Figure 2). The lower part contains the scripts what is entered by the user. The upper part contains the visualization generated by the script interpretation.

2.2 Scripting visualizations with Klotz

The Klotz scripting language is plain Smalltalk code. Each script is based on 4 principles:

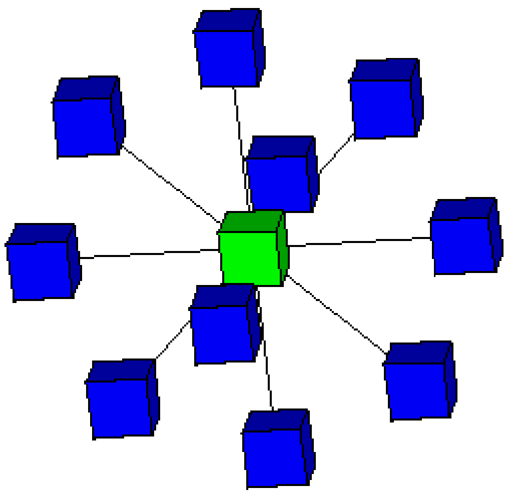


Figure 1: Magnitude subclasses.

- Elements composing the visualization are represented as *nodes*. Each node may have a shape and color that reflect some characteristics of the represented node.
- Relations between elements are represented with *edges*. The color and width of a shape depend on some arbitrary characteristics of connected objects.
- Elements may be ordered using *layouts*. A layout may use edges to direct the ordering.
- Containment is expressed with a *view*, an object that enables the construction of the visualization by offering numerous methods.

The scripting language supports 4 different ways of defining nodes. The message `node:` creates an individual node using some default visual properties (colored in blue, thin black border line). A variant of it is `nodes:` to add multiple nodes in one single instruction. The script

```
view node: Magnitude.
view node: Number.
view node: Time
```

is equivalent to

```
view nodes: { Magnitude. Number . Time}
```

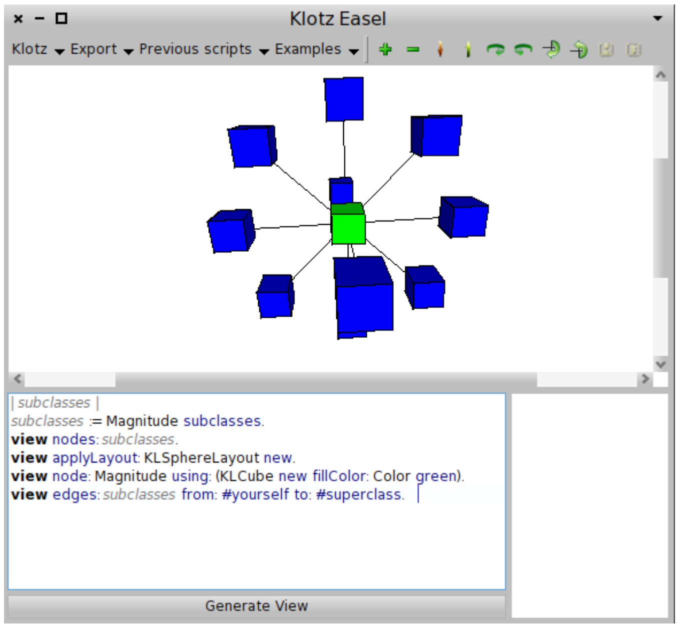
The result is shown in Figure 3.

The visual representation may be particularized according to some characteristics of the provided nodes. The message `node:using:` and `nodes:using:` allow graphical cube to be customized with metrics computed on the represented model. Consider the example, depicted in Figure 4:

```
view nodes: Magnitude subclasses using: (KLCube
    new height: #numberOfInstanceVariables)
```

The message `numberOfInstanceVariables` is sent to each of `Magnitude`'s subclasses. The result of `numberOfInstance-Variables` defines the height of the node.

The methods `edge:from:to:` and `edges:from:to:` use default black and thin line shape offered by the view. Figure 5 shows the result of the script:



```
| subclasses |
subclasses := Magnitude subclasses.
view nodes: subclasses.
view applyLayout: KLSphereLayout new.
view node: Magnitude using: (KLCube new fillColor: Color green).
view edges: subclasses from: #yourself to: #superclass.
```

Figure 2: The Klotz Easel

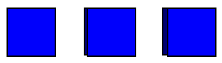


Figure 3: `Magnitude`, `Number` and `Time`.

```
view nodes: Magnitude subclasses.
view edges: (Magnitude subclasses) from: #yourself
     to: MetacelloVersionNumber
```

The message `edges:` *domain* `from:` *fromSelectorOrBlock* `to:` *toSelectorOrBlock* constructs an edge for each element of the *domain*. The source code is the result of evaluating *fromSelectorOrBlock* on the considered node and the target is obtained by evaluating *toSelectorOrBlock*. One-arg blocks and symbols are equally accepted.

2.3 The third dimension

Visualizations in third dimensions convey a "feeling of immersion" that Klotz is intensively exploiting. A number of tools and options are offered by either the visualization or the easel.

Light intensity.

A visualization contains one unique light, a white light located at the same position than the camera. The light intensity on a face is at its maximum when the face is orthogonal to the camera. When a surface of the face is close to be lined up with a light ray, the face is dark.

Emphasizing the perspective.

Perspective is the way a solid object is drawn on a two-dimensional surface so as to give the right impression of their height, width and depth. Our experience shows that it is difficult to precisely compare element positions when closely

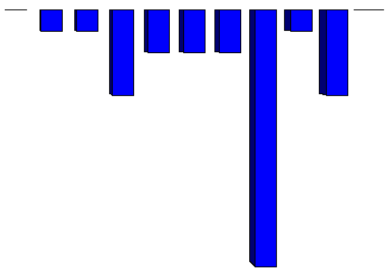


Figure 4: Magnitude subclasses, the heights shows the number of instance variables.

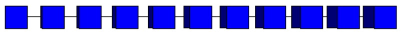


Figure 5: Basic edges

located from each other. Perspective may be emphasized thanks to an increase and decrease perspective commands offered by the easel.

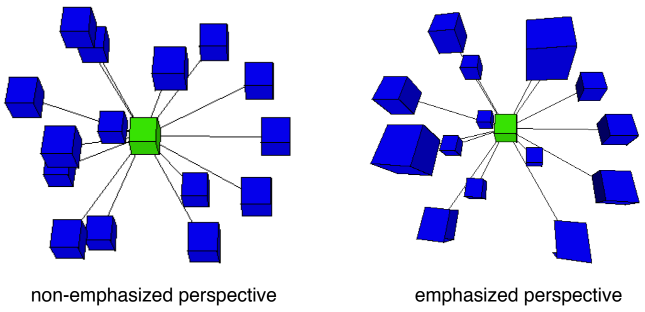


Figure 6: Emphasizing the perspective

Figure 6 illustrates this situation with a slight variation in the perspective.

Controlling the camera.

A user looks at a visualization through the view camera. The easel offers six commands to rotate and move the camera along every axis.

Layout.

Nodes are ordered using a layout. The default layout that is used when no other is specified is the horizontal line layout. A layout is specified using the message `applyLayout:`. Four additional layouts are available: cube layout, sphere layout, block layout, and scatterplot layout.

Cube Layout: This *layout* orders the nodes in a three-dimensional cube.

```
view
   nodes: Magnitude withAllSubclasses
   using: (KLCube new width:
    #numberOfInstanceVariables).
view applyLayout: (KLCubeLayout new).
```

Sphere Layout: nodes are located on the surface of a sphere, centered on the center of the view. The following example places all the subclasses of `Magnitude` on a sphere (Figure 8):

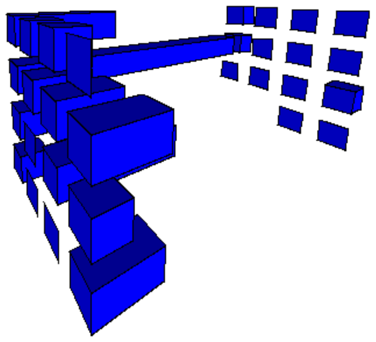


Figure 7: Cube Layout

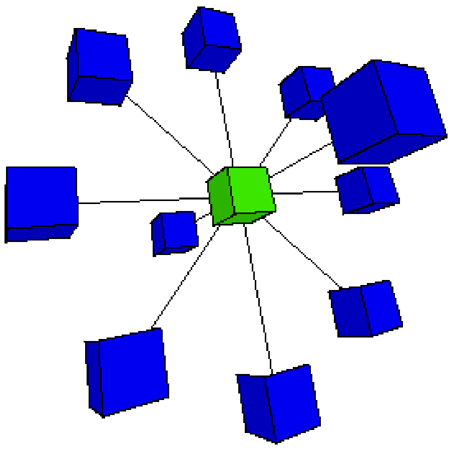


Figure 8: Sphere Layout

```
view nodes: Magnitude subclasses.
view applyLayout: KLSphereLayout new.

view
   node: Magnitude
   using: (KLCube new fillColor: Color green).
view
   edges: Magnitude subclasses
   from: #yourself
   to: #superclass.
```

Block Layout: nodes are hierarchically grouped and organized on a surface. Each group is uniquely colored. The assigned color is randomly chosen if none is specified in the shape.

The following script visualizes the structure of Klotz (Figure 9):

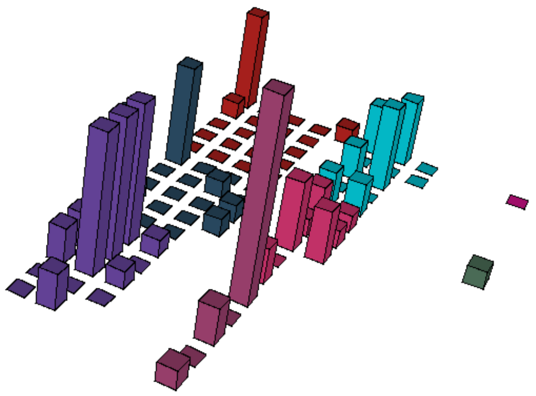


Figure 9: Block Layout

```
| shape packages block |
shape := KLCube new
            height: #numberOfInstanceVariables.
packages :=
   PackageInfo allPackages
      select: [:pak | pak packageName
                  matches: 'Klotz-*'].

block := KLBlockLayout new.
packages do: [ :pak |
   block with:
      {view nodes: pak classes using: shape } ].
view applyLayout: block
```

The script gives a randomly chosen color to each package of Klotz. The color is used to paint the classes of each package. Each node is a colored class. The height represents the number of attributes.

Scatterplot layout: nodes may be located on a three dimensional Cartesian. Each node has a 3d coordinate that is determined from applying three metric on the represented model. The following script plots each class of the Klotz-Kernel package along its number of attributes, number of methods (Figure 10):

```
view nodes:
   ((PackageInfo named:'Klotz-Kernel') classes).

view applyLayout:
(KLScatterCubeLayout new
    "blue line"
    x: [:cls | cls numberOfLinesOfCode / 1000];
    y: #numberOfInstanceVariables; "red line"
    z: #numberOfMethods) "green line"
```

3. IMPLEMENTATION

Klotz is freely available from `http://squeaksource.com/Klotz.html`

The current version of *Klotz*, yet not optimized, provides support up to 1000 *nodes* on screen, as well as, 1000 *edges* between these *nodes*.

As shown in Figure 11, the time taken for the easel to render a scene with the default horizontal line layout is almost proportional with the quantity of nodes and edges.

On Figure 12 illustrates the linear resource taken by the layouts.

The Klotz graphic engine its based on the construction of polygons that are later rendered as regular 2D Morphs. These polygons vertices are calculated using tridimentional vectors with absolute coordinates, and using matrices to produce each transformation (from zooming to the perspective transformation into 2D points). The 3D shapes are also basically optimised, the hidden faces (calculated to some point by need) are not rendered. These calculations, easy as they sound, are often troublesome since calculation for coordinates requires decent, and fast, calculations that sometimes can show perfectly reasonable results in theory (like a point located on the infinity) that must be taken care with proper aproximations, wich must also fix the not accurate aproximations made by the Integer/Float classes.

The Klotz graphic generation solely uses Morph facilities. The most common approach to visualize 3D graphics is to use OpenGL, a reference in the field. We deliberately decided to not use OpenGL for a number of practical reasons:

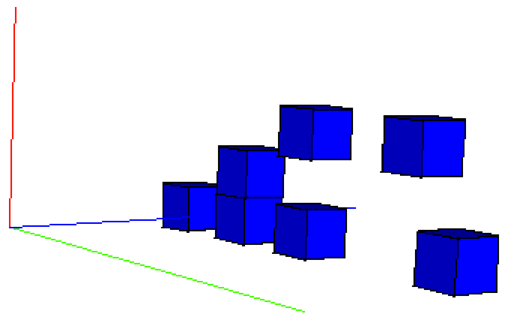


Figure 10: The Klotz Kernel Package

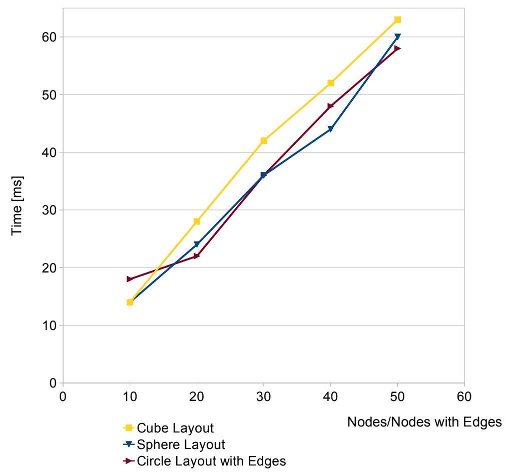


Figure 11: Benchmark (quantity vs time[ms]) for few *nodes*/*edges*

- OpenGL is distributed as a set of natives libraries, depending on the operating system. Libraries are accessed within Pharo using FFI or Alien, two technologies that interoperate with native libraries. Unfortunately, the recent advances with the Pharo virtual machine significantly reduced the usability of accessing external libraries.
- The visualizations produced with Mondrian rarely go

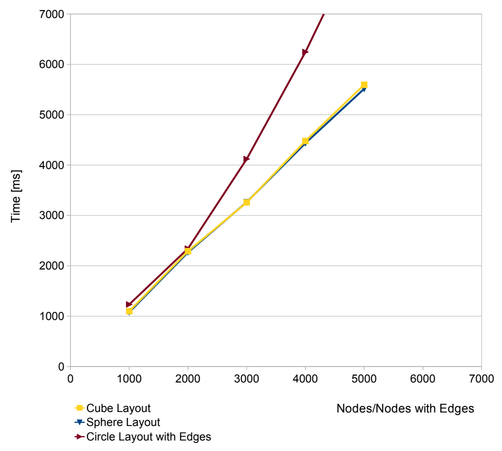


Figure 12: Benchmark (quantity vs time[ms]) for lots of *nodes/edges*

over 2,000 nodes and 1,000 edges. It is reasonable to expect similar figure as the upper limit for Klotz. OpenGL enables sophisticated rendering, including a high number of rendered polygons and advanced light composition. We do not expect to have such a need in a close future.

Basing Klotz on OpenGL is clearly on our agenda. For this, Alien needs to gain stability, especially with the Pharo JIT virtual machine (Cog).

4. RELATED WORK

4.1 Lumiere

Lumiere [2] is a 3D Framework,that applies a stage metaphor. This metaphor implies that all graphics are produced by cameras taking pictures of 3D shapes lit by the lights of the stage. A stage provides the setting for taking pictures of a composition of visual objects we call micro-worlds. Lumiere's objective is to implement a 3D graphical support to Pharo programmers, at a high level of abstraction. This framework uses OpenGL. As mentioned before, this libraries are entirely dependent of the OS and the right native OpenGL packages within the machine itself, to be used.

On the other hand, Klotz do not represent a competitor on that matter, Klotz is a 3D code visualization engine, its goal is not focus on the actual quality of the graphic interface, but on the expressiveness of the representation in order to achieve a better comprehension of the code. Right now Klotz graphical interface is a small engine that is completely based on the Morph engine. This does not mean to be the final core, eventually the system will need an external graphic engine, probably based on OpenGL, and that can also be Lumiere itself.

4.2 CodeCity

CodeCity [4] is a full fledged city metaphor environment, for code analisys. Its visualization is based on this metaphor, and its metrics are entirely defined (and chosen carefully) to faithfully explain software code. This concrete approach intends to focus not only on the visualization itself, but in the analysis of software evolution. It also support reverse engineering.

CodeCity is programmed in VisualWorks Smalltalk on top of the Moose platform. Just like Lumiere, it uses OpenGL for rendering.

When it comes to Klotz, not been as expressive as Codecity, eventually, and depending on the programer's skills, it can implement most aspect of its model.

5. CONCLUSION

Klotz is an agile three-dimensional visualization engine. Klotz visualizes a graph of objects, without any preparation of the objects. Klotz modeling system allows one to change a graph definition easily, making it simple and fast to adjust a desired visualization. Been based on any object as the items represented on the node it gives great dynamism to the kind of visualization that it can provide. As future work, we plan to:

- Implement drag-and-drop support to manage the nodes easily.
- Add an interface to see the information within the node on mouse focus, and change it dynamically.
- Optimize and improve the graphical libraries, if possible, change to a stable, OpenGL based library.
- Add lots of new Layouts, and with time, other figures to use as nodes.

Acknowledgment.
We thank Patricio Plaza for his effort and participation on an early version of Klotz.

6. REFERENCES

[1] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis'06)*, pages 135–144, New York, NY, USA, 2006. ACM Press.
[2] Fernando Olivero, Michele Lanza, and Romain Robbes. Lumiére: A novel framework for rendering 3d graphics in smalltalk. In *Proceedings of IWST 2009 (1st International Workshop on Smalltalk Technologies)*, pages 20–28. ACM Press, 2009.
[3] Marian Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, June 1995.
[4] Richard Wettel and Michele Lanza. Codecity: 3d visualization of large-scale software. In *ICSE Companion '08: Companion of the 30th ACM/IEEE International Conference on Software Engineering*, pages 921–922. ACM, 2008.

A programming environment supporting a prototype-based introduction to OOP

Carla Griggio[†] Germán Leiva[‡†] Guillermo Polito[‡†] Gisela Decuzzi[†] Nicolás Passerini[‡†]

[†]Universidad Tecnológica Nacional (UTN) – Argentina [‡]Universidad Nacional de Quilmes (UNQ) – Argentina

{carla.griggio | leivagerman | guillermopolito | giseladecuzzi | npasserini}@gmail.com

Abstract

This paper describes the features that a programming environment should have in order to help learning the object-oriented programming (OOP) paradigm and let students get the skills needed to build software using objects very quickly. This proposal is centered on providing graphical tools to help understand the concepts of the paradigm and let students create objects before they are presented the class concept [14]. The object, message and reference concepts are considered of primary importance during the teaching process, allowing quick acquisition of both theory and practice of concepts such as delegation, polymorphism and composition [7]. Additionally, a current implementation of the proposed software and the experience gained so far using it for teaching at universities and work trainings. Finally, we describe possible extensions to the proposed software that are currently under study.

Categories and Subject Descriptors K.3.2 [*Computer and Information Science Education*]: computer science education, information systems education; D.3.2 [*Language Classificationss*]: object-oriented languages; D.3.3 [*Language Constructs and Features*]: classes and objects, inheritance, polymorphism

General Terms Experimentation, Human Factors, Languages

Keywords Educational programming environments, object-oriented programming, teaching methodologies, prototype-based, objects visualization

1. Introduction

Frequently, in introductory courses to OOP, students have prior experience in structured programming. This is often counterproductive when understanding some of the basic concepts of the OOP paradigm, such as the relationship between a class and its instances, the difference between object and reference, delegation and polymorphism [16]. In order to minimize this difficulty, a possible strategy is to postpone the introduction of the class concept. This reduces the set of concepts needed to build programs [7].

Similiar difficulties appear in students who do not have prior knowledge in programming at the time of learning OOP, and specially in those cases it is convinient to bring down any complexity that a language might have in order to understand the ideas the paradigm proposes [15].

We propose to provide the student a reduced and graphical programming environment in which the object and the message are the central concepts instead of defining classes and then instantiate them. Moreover, it has to offer facilities to understand the concepts of environment, state and garbage collection. This tool is meant to make the first steps before getting into a full featured language and IDE, where concepts like class and inheritance can be properly learned once the basics are settled down.

2. Proposed programming environment

In this section we describe the features for a programming environment which helps our objectives, stating which common learning difficulties have to be prevented when introducing the OOP paradigm to students.

2.1 Multiple object environments - lessons

In order to introduce the concept of object environment we propose using lessons: an object environment associated to a group of objects, the references pointing to them, a workspace to interact with them and an object diagram. Lessons should be independent from each other; none of the elements mentioned above should be shared between different lessons.

The use of lessons also aims to help students organize their work and let the teacher offer environments with pre-existing objects as scenarios for excercises.

2.2 Defining and using objects

We believe that a visual tool to create objects allows the student to build programs focusing on object use instead of object creation at the first stages of the learning process. We propose to work with a simplified prototype oriented environment [8, 17], where the objects contain their own methods without the need of classes; and the creation of objects, attributes and methods is performed through a visual approach. This simplified object-oriented programming environment allows students to build programs in a very early stage of the course with many collaborating objects and making use of polymorphism.

Since the proposed environment lacks the concepts of classes or inheritance, the reuse of code is achieved by cloning objects. The cloning mechanism is simpler than those found in Self [13] or Javascript [17]. In those languages, tipically a prototypical object is divided in two parts, one containing the behavioral slots, which will be shared as a *parent object* among all the cloned objects, and another part containing the "state" slots, which must be cloned every time a new object is created. In our proposal, the same object servers the two purposes:

- the new object has the same internal structure as the original one.
- the new object has the cloned one as delegate. This means that the new object *inherits* all of the behaviour of the original one, but also it can *override* its behaviour by defining methods with the same name.

This simplified prototype approach enables code sharing mechanisms and facilitate the introduction of classes and inheritance in a later stage of the course based on a more classical view of the object-oriented paradigm.

To improve the student's programming experience, the environment should provide ready-to-use basic prototype objects such as numbers, booleans, collections and dates; in order to do more complex exercises the environment could also include networking, persistency and graphical user interface objects.

2.3 Objects and References

Students often confuse objects and references, believing they are the same. References are taught as a directed relation between two objects, or -as an exception- a workspace and an object. We emphasize these relations are mutable using diagrams to show the interaction between objects.

In order to make the difference clearer, our proposal is to separate the addition of a new object to the environment in two steps. The first step is to create an object. In the second step we associate a reference to it; this can be done by creating a new reference or by redirecting an existing reference. The association between the object and the reference could be done graphically using the object diagram.

This explicit separation between the created object and a reference pointing to it, improves the understanding of the difference between both concepts. Once established that difference, assignments could be used by the students without fear.

2.4 Object Diagram

A lesson should provide its own object diagram, where one can appreciate visually the relationship between living objects in that lesson's environment. This tool makes it easier to get a clearer distinction between the concept of object and reference, and helps to comprehend the state of the environment of the lesson at a given moment.

When the student interacts with the objects from a workspace, the diagram shows the state changes while the program executes. This provides a live vision of what happens in the object environment after each message-send.

The visual representation of the objects and references in the environment and the ability to follow their changes along with the program execution improves the understanding of some important concepts of the paradigm: like references and object identity.

3. Implementations

The first implementation of a tool based on the proposed style was an add-on for the Dolphin Smalltalk[1] environment which allowed the creation of objects without using classes and had a workspace to interact with them [7]. We used that first implementation to put in practice the idea of delaying the introduction of the class concept, and it was also useful as a model for the next implementations.

Nowadays, there is a new version of that tool built on top of Pharo Smalltalk[2] named LOOP (Learning Object Oriented Programming) implementing the first versions of the features described above [7].

The main menu of LOOP is a Lesson Browser (fig. 1), where lessons can be created, opened, deleted, exported to a file for sharing and imported back in the Pharo image. Exporting and importing a lesson is very useful for the teacher to evaluate exercises done by the students and also give them prebuilt lessons.

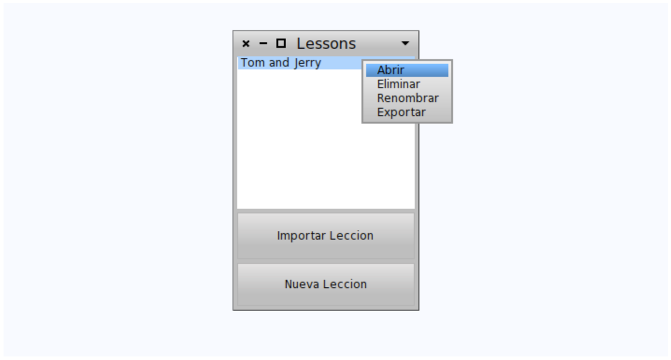
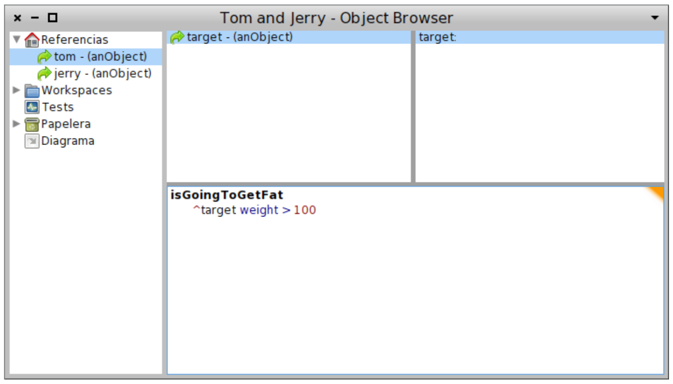


Figure 1. Lesson Browser

To create objects and references inside a lesson, the user has to use the object browser, which shows every reference and object created in the lesson environment. Selecting a reference from the menu brings up the object inspection window for the object that it points to, where the user can browse and define its attributes and methods (fig. 2).



```
isGoingToGetFat
    ^target weight > 100
```

Figure 2. Object Browser

A live object diagram shows the state of a lesson's environment and it is updated after every action that affects the environment state, i.e. addition or deletion of attributes of an object, message sends with side effects, creation of new objects, garbage collection, etc. (fig. 3).

The user can define many workspaces with different scenarios of interaction with the objects within the lesson (fig. 4).

An explicit garbage collection mechanism is illustrated with a Garbage Bin metaphore. Candidates for collection can be easily found in the object diagram because they would have no arrows pointing at them, and the Garbage Bin lists those same unreferenced objects (fig. 5). When the Garbage Bin is emptied, those unreferenced objects are deleted from the environment and dissappear from the object diagram (fig. 6, 7).

4. Experiences

LOOP was used in university courses and job trainings to put in practice the concepts of polymorphism, object composition and

[1] Object Arts Dolphin Smalltalk: http://www.object-arts.com/

[2] Pharo Smalltalk: http://www.pharo-project.org/

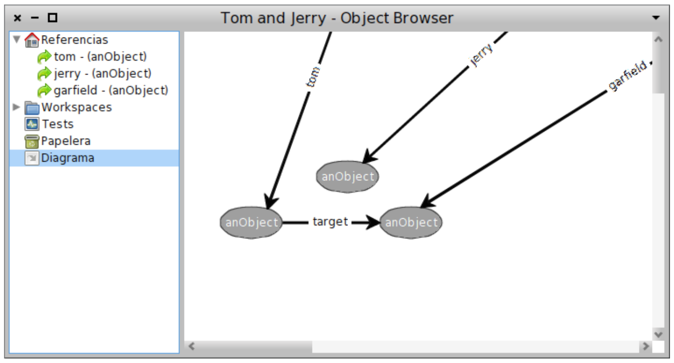
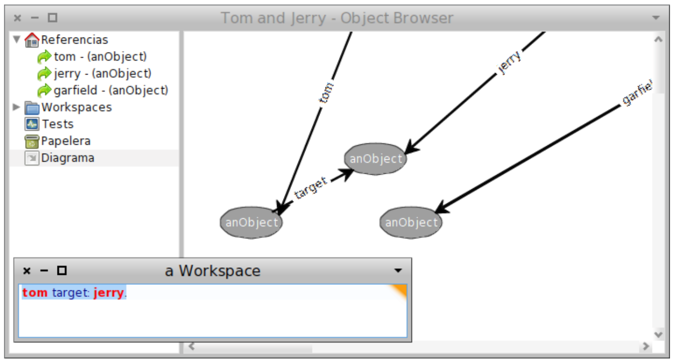
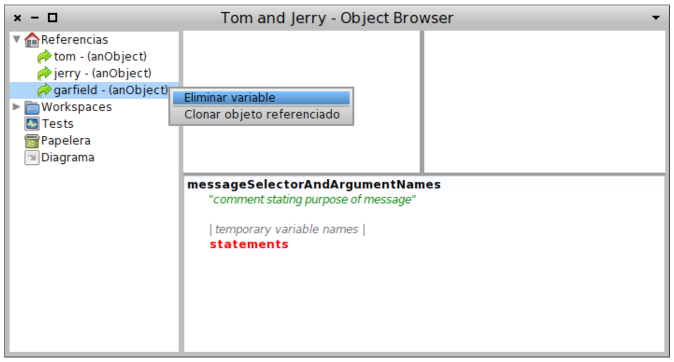


Figure 3. Interacting with objects from a workspace



```
tom target: jerry.
```

Figure 4. Object Diagram



```
messageSelectorAndArgumentNames
    "comment stating purpose of message"

    | temporary variable names |
    statements
```

Figure 5. Deleting a reference

delegation from the start. Afterwards, the concepts tought in the class was introduced as an alternative to build objects and share and extend their behavior without difficulties.

In object oriented job trainings for technologies like Smalltalk or Java, most of the trainees had few or no programming experience. Those courses demanded high quality training in a short time. Using LOOP intensively in the first lessons to introduce the paradigm, the transition to an specific programming language was faster than in previous courses. Also, the aspirants who used LOOP, showed a higher learning curve for other object-oriented technologies.

In UTN and UNQ object oriented courses where LOOP was used the students were already experienced with structured pro-

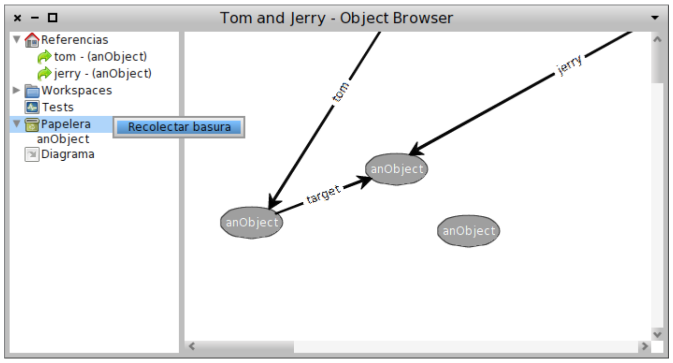


Figure 6. Object environment before garbage collection

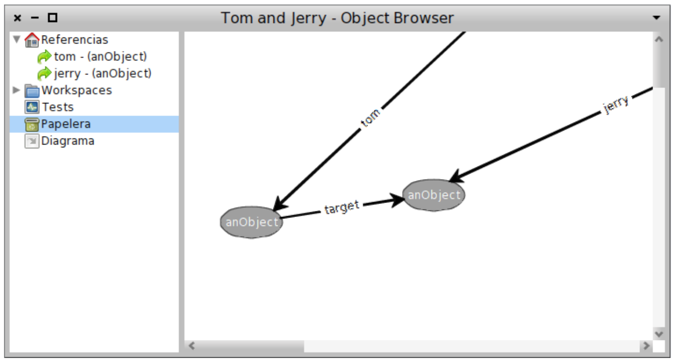


Figure 7. Object environment after garbage collection

gramming. The visual environment helped them to face the learning process without trying to just match their previous knowledge.

In Table 1 we present the last 5 years results of the OOP exams from the Programming Paradigms course given by Fernando Dodino at UTN. Fernando used the actual implementation of LOOP in 2011 Q1, having the most successful pass rate in 5 years.

Quarter	Pass Rate
2011 Q1	84,62%
2010 Q2	68.42%
2010 Q1	69.76%
2009 Q2	80.95%
2008 Q2	66.67%
2008 Q1	74.07%
2007 Q2	73.33%
2006 Q2	75.00%

Table 1. Pass Rates

5. Discussion and Related Work

LOOP is presented as a visual environment to teach OOP using a reduced set of language constructions and a prototype approach to create objects. It presents the main concepts of object, message and reference in a specialized tool with a visual representation of the object environment. Several visual tools to teach programming already exists, like ObjectKarel[2], Scratch[14] and Etoys[4].

ObjectKarel presents a visual tool based on the abstraction of robots to teach OOP, using a map where the robots-the objects-move when messages are sent to them. LOOP does not center on a specific abstraction like a robot: it allows the student to create any other abstraction. Scratch and Etoys, are aimed to teach the basics of programming to children, using visual objects and scripts to play with them. These projects are mainly oriented to teach the basics of programming to novices or children, while LOOP focuses on teaching professional programming to people who wants to enter in the software industry.

LOOP's prototype approach is mainly based on the ideas of Self [13] and Javascript [17], differing in some crucial points. LOOP's model is more restrictive than the existing in Self, allowing only one parent for each object, focusing on the concepts the tools wants to show -object, message, relations, polymorphism- instead of other more complex ones -classes, inheritance, traits, mixins- existing in a full featured language. The idea of parent slots/prototypes in LOOP is completely handled by the tool, whithout letting the student manage them.

6. Conclusions

Our experience using LOOP shows that students learn the object oriented programming paradigm more easily when we incorporate a programming environment offering visual tools for creating objects and interacting with them. Also, defining our own programming environment, allows us to select the programming concepts we want to introduce at each step of the learning process, providing an excellent ground for a gradual introduction of those concepts. The programming environment proves to be very useful for students, with or without previous programming knowledge, because it allows them to focus on the most important concepts of the programming paradigm, avoiding technology-specific distracting elements. A solid knowledge of those concepts facilitates a later transition to any other object-oriented programming language.

7. Further work

The current implementation of LOOP is based on Smalltalk and the syntax used when programming is the syntax of the Smalltalk language. We think this syntax is the best choice for an introductory course, because of its simplicity and is resemblance of the natural language. Also it is meant for courses that, after an introduction based on LOOP, can continue learning object-oriented programming in a real Smalltalk environment. Nevertheless, we consider that a future implementation of LOOP should include a configurable syntax, allowing the teacher to choose the most similar option to that of the language he is planing to use in the later stages of the course. For example, if the course is going to continue using the Java language, a C-like syntax could be considered for being used on LOOP. This would allow us to take the most of LOOP also in courses based on other languages different from Smalltalk. Besides, the tool could allow the teacher to configure its own syntax.

We also want to include a configurable type system. We think that explicit type definitions take the focus away from the most important concepts of the paradigm and should be preferably avoided in introductory courses. Nevertheless, since many object-oriented languages make heavy use of static type-systems with explicit type definitions, a configurable type-system should also be considered.

The current implementation of LOOP offers a limited support for developing unit tests. This part of the tool should be improved in order to facilitate the use of test-driven development (TDD) from the begining of the course. Since the nature of LOOP programming environment imposes some specific difficulties to build tests without side-effect, a concrete implementation of TDD inside of LOOP is still to be analyzed in depth.

We also consider improving both appearance and functionality of the graphical diagrams of LOOP. Object diagrams should be interactive, allowing the creation of new objects or sending messages from the diagram itself, as an alternative to the workspace and the reference-panel. Also, sequence and collaboration diagrams would be useful for the comprehension of the dynamic relationships between objects. This kind of diagrams could be inferred from the evaluation of any piece of code, even the execution of tests.

Another subject of research is a "debugger" for the tool [1]. We think that a live and powerful debugger *à la Smalltalk* is a rich tool for the understanding of the whole environment behaviour. After a message is sent, a debugger view can be used like a video player, with play, forward and backward buttons to navigate the message stack and see how the state changes after each message send in the object diagram.

Finally, there are some improvements to be made to the user interface, such as shortcuts, code completion, improved menus or internationalization. Currently the tool is only available in spanish, we want to make it configurable to add more languages as necessary.

Acknowledgments

Carlos Lombardi and Leonardo Cesario collaborated in the first implementation of the programming environment. Also, many of the teachers and students of the subjects of *Programming Paradigms* at UTN and *Computers II* at UNQ gave us great help by testing our tools, reporting bugs and proposing new ideas. Gabriela Arévalo, Victoria Griggio and Débora Fortini helped us in the writing of this paper. Fernando Dodino provided us his course statistics.

References

[1] J. Bennedsen and C. Schulte. Bluej visual debugger for learning the execution of object-oriented programs? *Trans. Comput. Educ.*, 10:8:1–8:22, June 2010. ISSN 1946-6226. doi: http://doi.acm.org/10.1145/1789934.1789938. URL `http://doi.acm.org/10.1145/1789934.1789938`.

[2] R. Findler, John. Clements, Cormac. Flanagan, Matthew. Flatt, Shriram. Krishnamurthi, Paul. Steckler and Matthias. Felleisen DrScheme: a programming environment for Scheme *J. Functional Programming* 12: 159–182, March 2002. URL `www.cs.cmu.edu/ rwh/courses/refinements/papers/Findleretal02/jfp.pdf` - .

[3] M. Hertz and E. D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 313–326, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: http://doi.acm.org/10.1145/1094811.1094836. URL `http://doi.acm.org/10.1145/1094811.1094836`. http://www.squeakland.org/resources/articles/article.jsp?id=1008

[4] A. Kay Squeak Etoys authoring 2005-01-01. URL `http://www.vpri.org/pdf/rn2005001_learning.pdf`.

[5] S. Kouznetsova. Using bluej and blackjack to teach object-oriented design concepts in cs1. *J. Comput. Small Coll.*, 22:49–55, April 2007. ISSN 1937-4771. URL `http://portal.acm.org/citation.cfm?id=1229637.1229646`.

[6] N. Liberman, C. Beeri, and Y. Ben-David Kolikant. Difficulties in learning inheritance and polymorphism. *Trans. Comput. Educ.*, 11:4:1–4:23, February 2011. ISSN 1946-6226. doi: http://doi.acm.org/1921607.1921611. URL `http://doi.acm.org/1921607.1921611`.

[7] C. Lombardi, N. Passerini, and L. Cesario. Instances and classes in the introduction of object oriented programming. *Smalltalks 2007 – Primera Conferencia Argentina de Smalltalk*, 2007.

[8] O. Madsen. Strategic research directions in object-oriented programming. *ACM Comput. Surv.*, 28, December 1996. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/242224.242424. URL `http://doi.acm.org/10.1145/242224.242424`.

[9] I. Michiels, A. Fernández, J. Börstler, and M. Prieto. Tools and environments for understanding object-oriented concepts. In *Proceedings of the Workshops, Panels, and Posters on*

Object-Oriented Technology, ECOOP '00, pages 65–77, London, UK, 2000. Springer-Verlag. ISBN 3-540-41513-0. URL `http://portal.acm.org/citation.cfm?id=646780.705783`.

[10] M. Satratzemi, S. Xinogalos and V. Dagdilelis. An Environment for Teaching Object-Oriented Programming: ObjectKarel In *Proceedings of the The 3rd IEEE International Conference on Advanced Learning Technologies* , ICALT'03 ISBN 0-7695-1967-9 URL `http://portal.acm.org/citation.cfm?id=961590`.

[11] V. Shanmugasundaram, P. Juell, and C. Hill. Knowledge building using visualizations. In *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, ITICSE '06, pages 23–27, New York, NY, USA, 2006. ACM. ISBN 1-59593-055-8. doi: http://doi.acm.org/10.1145/1140124.1140134. URL `http://doi.acm.org/10.1145/1140124.1140134`.

[12] V. Shcherbina, P. Vortman, and G. Zodik. A visual object-oriented development environment (voode). In *Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '95, pages 57–. IBM Press, 1995. URL `http://portal.acm.org/citation.cfm?id=781915.781972`.

[13] D. Ungar and R. B. Smith. Self. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 9–1–9–50, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-7. doi: http://doi.acm.org/10.1145/1238844.1238853. URL `http://doi.acm.org/10.1145/1238844.1238853`.

[14] I. Utting, S. Cooper, M. Kölling, J. Maloney, and M. Resnick. Alice, greenfoot, and scratch – a discussion. *Trans. Comput. Educ.*, 10:17:1–17:11, November 2010. ISSN 1946-6226. doi: http://doi.acm.org/10.1145/1868358.1868364. URL `http://doi.acm.org/10.1145/1868358.1868364`.

[15] P. Ventura and B. Ramamurthy. Wanted: Cs1 students. no experience required. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, SIGCSE '04, pages 240–244, New York, NY, USA, 2004. ACM. ISBN 1-58113-798-2. doi: http://doi.acm.org/10.1145/971300.971387. URL `http://doi.acm.org/10.1145/971300.971387`.

[16] G. R. S. Weir, T. Vilner, A. J. Mendes, and M. Nordström. Difficulties teaching java in cs1 and how we aim to solve them. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, ITiCSE '05, pages 344–345, New York, NY, USA, 2005. ACM. ISBN 1-59593-024-8. doi: http://doi.acm.org/10.1145/1067445.1067543. URL `http://doi.acm.org/10.1145/1067445.1067543`.

[17] L. Wilkens. Objects with prototype-based mechanisms. *J. Comput. Small Coll.*, 17:131–140, February 2002. ISSN 1937-4771. URL `http://portal.acm.org/citation.cfm?id=772636.772659`.

Memoization Aspects: a Case Study

Santiago Vidal
ISISTAN Research Institute, Faculty of Sciences, UNICEN University, Campus Universitario, Tandil, Buenos Aires, Argentina, Also CONICET
svidal@exa.unicen.edu.ar

Claudia Marcos
ISISTAN Research Institute, Faculty of Sciences, UNICEN University, Campus Universitario, Tandil, Buenos Aires, Argentina, Also CIC
cmarcos@exa.unicen.edu.ar

Alexandre Bergel
PLEIAD Lab, Department of Computer Science (DCC), University of Chile, Santiago, Chile
http://bergel.eu

Gabriela Arévalo
Universidad Nacional de Quilmes, Bernal, Buenos Aires, Argentina, Also CONICET
garevalo@unq.edu.ar

Abstract

Mondrian, an open-source visualization engine, uses caching mechanism to store precalculated values avoiding calculating them when they are needed in the application. During evolution phases of any software, unanticipated changes are difficult to predict. Particularly in the case of Mondrian, in the planned extensions of this tool, we have noticed that the caches sometimes are not used. Using aspect-oriented programming, we have refactored these caches into a well defined aspect to solve the mentioned problem. We have achieved it without paying the price of runtime problems.

1. Introduction

Dealing with emerging requirements in a target application is probably one of the most difficult challenges in software engineering [13].

This paper presents a solution to a maintenance problem we recently faced with while developing the Mondrian application. Mondrian is an agile visualization engine implemented in Pharo, and is used in more than a dozen projects[1]. As in many software developments, new requirements of several increasing number of clients impact on design decisions that were hold for years.

[1] http://moosetechnology.org/tools

Mondrian uses simple two-dimensions rendering to graphically visualize a target domain. Mondrian is almost exclusively used to visualize software metrics and lets the user produce a wide range of visual representations[2]. One of the strong design decision that Mondrian holds is the structure of its multiple cache mechanisms.

Mondrian has 9 caches spread over the graphical element hierarchy. The caches aim to quickly render two dimensional widgets, graphically composed of rectangle and line shapes. Mondrian caches are instances of the memoization technique[3]. Sending twice the same message returns the same value if there is no side effect that impacts on the computation.

Unfortunately, the new requirements of Mondrian defeats the purpose of some caches. One example is the bounds computation to obtain the circumscribed rectangle of a two-dimensional graphical element. This cache is senseless in a 3D setting. Bypassing the cache results in a complex extension of Mondrian.

We have first identified where the caches are implemented and how they interact with the rest of the application. For each cache, we marked methods that initialize and reset the cache. We have subsequently undertaken a major refactoring of Mondrian core: we have implemented a prototyping version of Mondrian, in which caches are externalized from the base code. We implement our refactoring with a customized aspect-based mechanism. We were able to modularize the cache while preserving the overall architecture and Mondrian performances were not affected with the refactoring process.

The contributions of this paper are: (i) identification of memoizing cross-cutting concern and (ii) refactorization of

[2] http://www.moosetechnology.org/docs/visualhall

[3] http://www.tfeb.org/lisp/hax.html#MEMOIZE

these cross-cutting concerns into modular and pluggable aspects.

The paper is structured as follows. Section 2 shows the problem we faced with when trying to evolve Mondrian. Section 3 describes the aspect-based solution we adopted. Section 4 presents the impacts of our solution on Mondrian. Section 5 briefly summarizes the related work. Section 6 presents some conclusions.

2. Making Mondrian Evolve

This section details a maintenance problem we have faced with when developing Mondrian.

2.1 Turning Mondrian into a framework

Mondrian[4] is an agile visualization library [12]. A domain specific language is provided to easily define interactive visualizations. Visualizations are structured along a graph structure, made of nested nodes and edges. Mondrian is a crucial component, used in more than a dozen independent projects. To meet clients performance requirements, Mondrian authors are focused on providing a fast and scalable rendering. To that purpose, Mondrian contains a number of caches to avoid redundant code executions.

Mondrian is on the way to become a visualization engine framework more than a library as it is currently. It is now used in situations that were not originally planned. For example, it has been used to visualize the real-time behavior of animated robots[5], 3D visualizations[6], whereas it has been originally designed to visualize software source code using plain 2D drawing [9]. The caches that are intensively used when visualizing software are not useful and may even be a source of slowdown and complexity when visualizing animated robots.

2.2 Memoization

Memoization is an optimization technique used to speed up an application by making calls avoid repeating the similar previous computation. Consider the method `absoluteBounds` that any Mondrian element can answer to. This method determines the circumscribed rectangle of the graphical element:

```
MOGraphElement>>absoluteBounds
   absoluteBoundsCache
      ifNotNil: [ ^ absoluteBoundsCache ].
   ^ absoluteBoundsCache :=
      self shape absoluteBoundsFor: self
```

The method `absoluteBoundsFor:` implements a heavy computation to determine the smallest rectangle that contains all the nested elements. Since this method does not perform any global side effect, the class `MOGraphElement` defines an instance variable called `absoluteBoundsCache` which is initialized at the first invocation of `absoluteBounds`. Subsequent invocations will therefore use the result previously computed.

Obviously, the variable `absoluteBoundsCache` needs to be set to `nil` when the bounds of the element are modified (*e.g.,* adding a new nested node, drag and dropping).

2.3 Problem

Mondrian intensively uses memoization for most of its computation. A user-performed interaction that leads to an update invalidates the visualization, since the cache need to be recomputed. These memoizations were gradually introduced over the development of Mondrian (which started in 2006). Each unpredictable usage, such as for example visualization of several inner nodes, leaded to a performance problem that has been solved using a new memoization. There are about 32 memoizations in the current version of Mondrian.

These caches have been modified along the common usage of Mondrian. Visualizations produced are *all* static and employ colored geometrical objects.

Extending the range of applications for Mondrian turns some of the caches senseless. For example `absoluteBoundsCache` has no meaning in the three-dimensional version of Mondrian since the circumscribed rectangle is meaningful only with two dimensions.

Using delegation. We first tried to address this problem by relying only on explicit objects, one for each cache. This object would offer the needed operations for accessing and resetting a cache.

As exemplified with the method `absoluteBounds` given above, the caches are implemented by means of dedicated instance variables defined in the `Cache` class. That is to say, each cache is associated with an instance variable. In this way, a variable of the Cache class, called `generalCache`, is defined in the `MOGraphElement` class. Through this variable the different caches can be accesed with the method `cacheAt:( key)` where `key` is a string with the name of the cache.

Figure 1 illustrates this situation where a graph element has one instance of the `Cache` class, itself referencing to many instances of `CacheableItem`, one for each cache.

Below we show how the method `absoluteBounds` is written following this approach:

```
MOGraphElement>>absoluteBounds
   (generalCache cacheAt: 'absoluteBoundsCache')
     ifCacheNil: [
   (generalCache cacheAt: 'absoluteBoundsCache')
    putElement: (self shape absoluteBoundsFor: self)].
   ^ (generalCache cacheAt: 'absoluteBoundsCache')
     getInternalCache.
```

As we observe, with this approach the different instance variables related with the caches are replaced by a unique variable called `generalCache`. On the other hand, the legibility of the method is deteriorated as well as the performance.

Significant overhead. This modularization solely based on delegating messages has a significant overhead at execution

[4] http://www.moosetechnology.org/tools/mondrian

[5] http://www.squeaksource.com/Calder.html

[6] http://www.squeaksource.com/Klotz.html

time due to the additional indirection. The separation of this concern is not a trivial problem. Specifically, when we use this solution, the caches mechanism was 3 to 10 times slower, with the delay proportional to the number of elements.

2.4 Requirement for refactoring

Refactoring Mondrian is a task to be performed carefully. In particular, the refactoring has the following constraints:

- All cache accesses have to be identified. This is essential to have all the caches equally considered.
- No cost of performance must be paid, else it defeats the whole purpose of the work.
- Readability must not be reduced.

3. Aspect-based Refactoring

The goal of the refactoring is the separation of the *Cache Concern* from the four essential classes of *Mondrian*: *MOGraphElement* and its subclasses (*MOEdge*, *MONode*, and *MORoot*). These classes have 235 methods and more than 1000 number of lines of codes in total.

3.1 Identifying caches

The first step of the refactoring is identifying the caches. This initial identification of the caches is done with the information provided by the developers of Mondrian and is based on knowing the variables related to the caches and the places where they are used. The caches are mostly identified by browsing the methods in which the caches variables are referenced and accessed. Nine different caches are found in Mondrian: *cacheShapeBounds*, *cacheForm*, *boundsCache*, *absoluteBoundsCache*, *elementsToDisplayCache*, *lookupNodeCache*, *cacheFromPoint*, *cacheToPoint*, and *cacheBounds*. Each of them has a different internal structure according to what is stored: *boundsCache* will hold an instance of the class `Rectangle` and *cacheForm* an instance of a bitmap `Forms`, for example.

After this initial identification, the fragment of codes in which the caches are used are grouped together based on the purpose of its use (*e.g.,* saving information, obtaining the data stored). Each group is associated with different activities:

- Initialize and reset the cache: the fragments of code in this group initialize or reset a cache variable putting them in nil or creating an instance of an object.
- Retrieve the cache value: this group obtains the information that is saved in a cache.
- Store data in the cache: the code fragments grouped here store information into a cache variable.

These groups allow the identification of code patterns that are repeated when using the caches. An aspect refactoring is associated for each found pattern [8]. These code patterns are described in the following subsections.

3.2 Pattern description

We identified 5 code patterns based on Mondrian source code and are described below. Each pattern is described with a relevant typical occurrence, the number of occurrences we found in Mondrian and an illustration.

Reset Cache. A cache has to be invalidated when its content has to be updated. We refer to this action as reset. The code to express a reset is *cache:=resetValue* where *resetValue* and the initial value the cache should have. Typically, the *resetValue* depends on the type of the stored value. It could be `nil`, an empty dictionary, or a particular value (*e.g.,* `0@0`). Eighteen occurrences of this pattern are found in Mondrian. We found that in some occurrences the reset of the caches is performed before the logic of the method, and other methods in which the reset must be done after. For example, the method `MOGraphElement>>shapeBoundsAt:put:` resets the caches *absoluteBoundsCache* and *boundsCache* before modifying the cache cacheShapeBounds. In contrast, the method `MONode >>translateBy:bounded:` resets the caches *boundsCache* and *absoluteBoundsCache* after executing most of the statements of the method.

Consider the method `MOGraphElement>>resetCache`. This method is called whenever the user drags and drops a graphical element. In this method, the *Reset Cache* pattern is repeated in four occasions to reset the caches *boundsCache*, *absoluteBoundsCache*, *cacheShapeBounds*, and *elementsToDisplayCache*. In this case, the reset of the caches can be done before or after the execution of the methods *resetElementsToLookup* and *resetMetricCaches*.

```
MOGraphElement>>resetCache
   self resetElementsToLookup.
   boundsCache := nil.
   absoluteBoundsCache := nil.
   cacheShapeBounds :=SmallDictionary new.
   elementsToDisplayCache := nil.
   self resetMetricCaches
```

Lazy Initialization. In some situations it is not relevant to initialize the cache before it is actually needed. This happens when a graphical element is outside the scrollbar visual frame: no cache initialization is required for a graphical element if the element is not displayed. These caches are relevant only when the user actually sees the element by scrolling the visualization. Typically, the structure of this pattern is: *^ cache ifNil:[cache:=newValue]*. Mondrian contains five occurrences of a lazy cache initialization. Consider the method `bounds`:

```
MOEdge>>bounds
   ^ boundsCache ifNil:[boundsCache:= self shape
     computeBoundsFor: self ].
```

The circumscribed rectangle is returned by `computeBoundsFor:` and is performed only when an edge is actually visible (`bounds` is used in `drawOn:`, the rendering method).

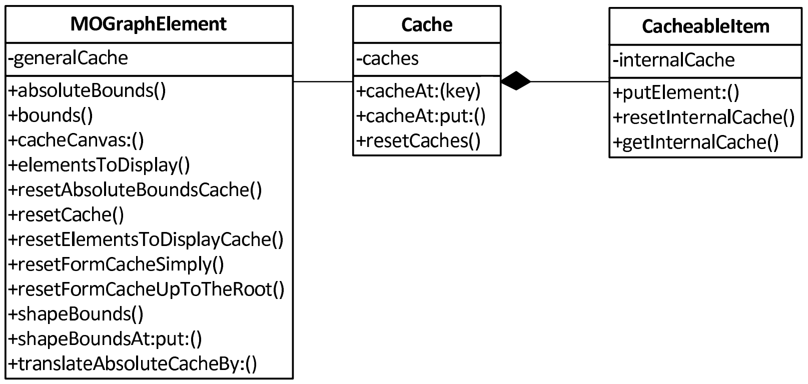


Figure 1. Cache behavior delegation.

Cache Initialization. This pattern represents a situation in which a value is assigned to a cache. The structure of the pattern is only an assignment: *cache := aValue*. This pattern is found in three occasions. Consider the method `cacheCanvas:`

```
MOGraphElement>>cacheCanvas: aCanvas
   cacheForm:= aCanvas form
     copy: ((self bounds origin + aCanvas origin-(1@1))
            extent: (self bounds extent + (2@2))).
```

The method `cacheCanvas:` is invoked only during testing in order to verify some characteristics of the caches such as their effectiveness.

Return Cache. This pattern shows the situation in which a cache is accessed. The structure of the pattern is the return of the cache: *return cache*. This pattern is found in four occasions. Next, the method `shapeBounds` is presented as an example in which *cacheShapeBounds* is accessed.

```
MOGraphElement>>shapeBounds
   ^ cacheShapeBounds
```

Cache Loaded. This pattern checks whether one cache or more are initialized or conversely, if they are not nil. So, the structure of the pattern for a single cache is *cache != nil*. This pattern is found in two occasions. Next the method *isCacheLoaded* is presented as an example of this pattern.

```
MOGraphElement>>isCacheLoaded
   ^cacheForm notNil.
```

Additionally, Table 1 gives the occurrences of each pattern in the `MOGraphElement` hierarchy, the methods involved in each pattern, and the caches related with a pattern.

Figure 2 shows the distribution of the caches over the main Mondrian classes, methods in which the caches are used, and the classes where each cache is defined. As we observe, the caches are used and defined across the whole class hierarchy.

3.3 Cache concerns as aspects

Once the code patterns are identified, we set up strategies to refactor them. The goal of the refactorization is the separation of these patterns from the main code without changing the overall behavior, enforced by an extended set of unit tests.

The refactoring is performed by encapsulating each of the nine caches into an aspect. Aspect definition weaving is achieved via a customized AOP mechanism based on code annotation and source code manipulation.

The refactoring strategy used is: for each method that involves a cache, the part of the method that directly deals with the cache is removed and the method is annotated. The annotation is defined along the cache pattern associated to the cache access removed from the method. The annotation structure is *<patternCodeName: cacheName>* where *cacheName* indicates the name of the cache to be considered and *patternCodeName* indicates the pattern code to be generated. For example, the annotation *<LazyInitializationPattern: #absoluteBoundsCache>* indicates that the *Lazy Initialization* pattern will be "weaved" for the cache *absoluteBoundsCache* in the method in which the annotation is defined.

The weaving is done via a customized code injection mechanism. For each annotation a method may have, the code injector performs the needed source code transformation to use the cache. Specifically, the weaving is achieved through the following steps:

1. A new method is created with the same name that the method that contains the annotation but with the prefix "compute" plus the name of the class in which is defined. For example, given the following method:

```
MOGraphElement>>absoluteBounds
   <LazyInitializationPattern: #absoluteBoundsCache>
   ^ self shape absoluteBoundsFor: self
```

   a new method called `computeMOGraphElementAbsoluteBounds` is created.

2. The code of the original method is copied into the new one.

```
MOGraphElement>>computeMOGraphElementAbsoluteBounds
   ^ self shape absoluteBoundsFor: self
```

Cache	Occurrences	Methods involved	Caches involved
Reset Cache	18	10	boundsCache, absoluteBoundsCache, cacheShapeBounds, elementsToDisplayCache, cacheForm, cacheFromPoint, cacheToPoint
Lazy Initialization	5	5	elementsToDisplayCache, absoluteBoundsCache, boundsCache, cacheBounds
Cache Initialization	3	3	cacheForm, cacheFromPoint, cacheToPoint
Return Cache	4	4	cacheShapeBounds, cacheForm, cacheFromPoint, cacheToPoint
Cache Loaded	2	2	cacheForm, cacheFromPoint, cacheToPoint
Total	32	24	

Table 1. Cache Concern scattering summary.

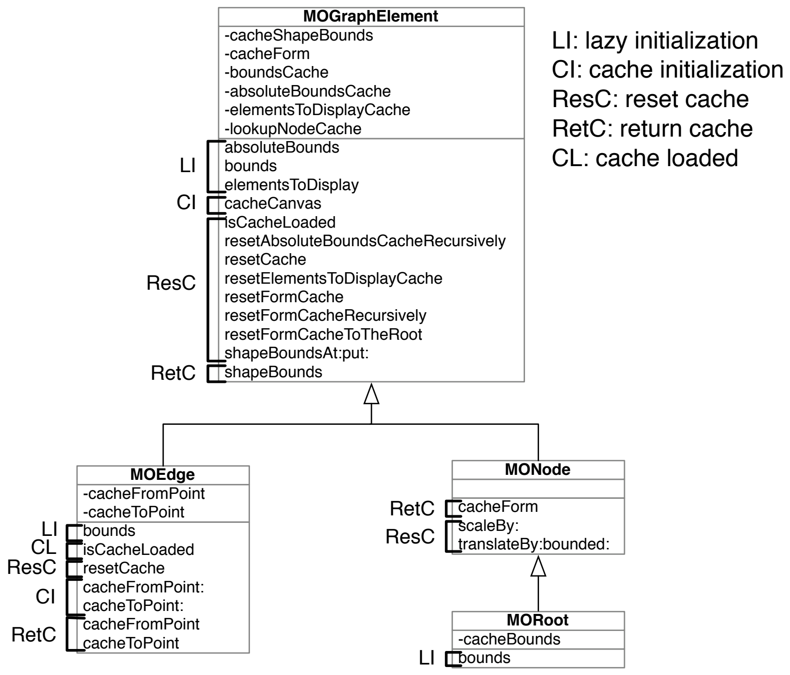


Figure 2. Pattern locations in the MOGraphElement hierarchy

3. The code inside the original method is replaced by the code automatically generated according to the pattern defined in the annotation. This generated method contains a call to the new one of the Step 1.

```
MOGraphElement>>absoluteBounds
   absoluteBoundsCache
      ifNotNil: [ ^ absoluteBoundsCache].
   ^ absoluteBoundsCache:=
      (self computeMOGraphElementAbsoluteBounds)
```

In order to automatically generate the code to be injected, the code weaver uses a class hierarchy (Figure 3), rooted in the abstract `CachePattern` class. `CachePattern` class contains the methods needed to process annotations (called pragmas in the Pharo terminology). Each subclass overrides the method `generateMethodWith:` to perform the source code manipulation.

Next, we present the refactorings applied to each code pattern.

Reset Cache. In order to refactor this pattern each statement that resets a cache was extracted using an annotation. The annotation contains the cache to be resetted. Since in some cases the resets are done at the beginning of a method and others at the end, a hierarchy of Reset Cache pattern is created. Figure 3 shows this hierarchy, which is composed of the classes *AbstractResetCachePattern*, *BeforeResetCachePattern*, and *AfterResetCachePattern*. The annotations are defined in the classes at the bottom of the hierarchy as <BeforeResetCachePattern: cacheName> and <AfterResetCachePattern: cacheName> respectively. For example, in the case presented in Section 3.2 of the method *resetCache*, an annotation is defined for each reset of a cache leaving a cleaner code in the method. In this case, all the resets are done before the method call, so the used annotations are the ones defined by *BeforeResetCachePattern*. Even though the order of calls is changed (in comparison with the original method), the method behavior is not modified. The code to be generated will reset the cache defined in the annotation. Following, the refactored code is presented:

```
MOGraphElement>>resetCache
   <BeforeResetCachePattern: #absoluteBoundsCache>
   <BeforeResetCachePattern: #elementsToDisplayCache>
   <BeforeResetCachePattern: #boundsCache>
   <BeforeResetCachePattern: #cacheShapeBounds>
   self resetElementsToLookup.
   self resetMetricCaches
```

The methods *resetElementsToLookup* and *resetMetricCaches* perform additional actions that do not involve the cache variables. For this reason they remain in the method *resetCache*.

After the code injection, the method *resetCache* is transformed into:

```
MOGraphElement>>resetCache
   absoluteBoundsCache:=nil.
   elementsToDisplayCache:=nil.
   boundsCache:=nil.
   cacheShapeBounds:=SmallDictionary new.
   self computeMOGraphElementresetCache
```

where the method `computeMOGraphElementresetCache` is:

```
MOGraphElement>>computeMOGraphElementresetCache
   self resetElementsToLookup.
   self resetMetricCaches
```

This mechanism of injection of the generated code is the same for the rest of the patterns.

Lazy Initialization. To refactor this pattern the precondition checking is contained into an annotation defined as <LazyInitializationPattern: cacheName>. Given that the cache is initialized with a value when the precondition fails, the original method is modified to return this value. For example, in the case of the method *bounds* introduced in the previous section, the code related to the cache is extracted using the annotation and only the value to initialize the cache remains in the method as shown the code below:

```
MOEdge>>bounds
   <LazyInitializationPattern: #boundsCache>
   self shape computeBoundsFor: self.
```

Thus, the code to be generated in this example will be *boundsCache ifNotNil: [^ boundsCache]. ^ boundsCache:= computeMOEdgeBounds.*

Cache Initialization. The refactorization of this cache is similar to the last one. Given that the structure of the pattern is an assignment, the first section of the assignment (*cacheName:=*) will be generated automatically by the weaver using an annotation <CacheInitializationPattern: cacheName>. The value at which the cache is initialized constitutes the method body. In the case of the example presented in Section 3.2, the refactored code is shown below:

```
MOGraphElement>>cacheCanvas: aCanvas
   <CacheInitializationPattern: #cacheForm>
   (aCanvas form copy: ((self bounds origin + aCanvas
     origin
   - (1@1)) extent: (self bounds extent + (2@2)))).
```

Return Cache. In this refactorization the entire return clause is encapsulated by the annotation. The annotation is defined as <ReturnCachePattern: cacheName>. Following, the refactored code of the example shown in the last section is presented:

```
 MOGraphElement>>shapeBounds
   <ReturnCachePattern: #cacheShapeBounds>
```

Cache Loaded. In order to refactor this pattern the cache checking is encapsulated by an annotation defined as <CacheLoadedPattern: cacheName>. The code generated contains a sentence in which the checking is done for all the caches defined in the annotations of this pattern contained in a method. In the case of the example presented in Section 3.2, the refactored code is shown below:

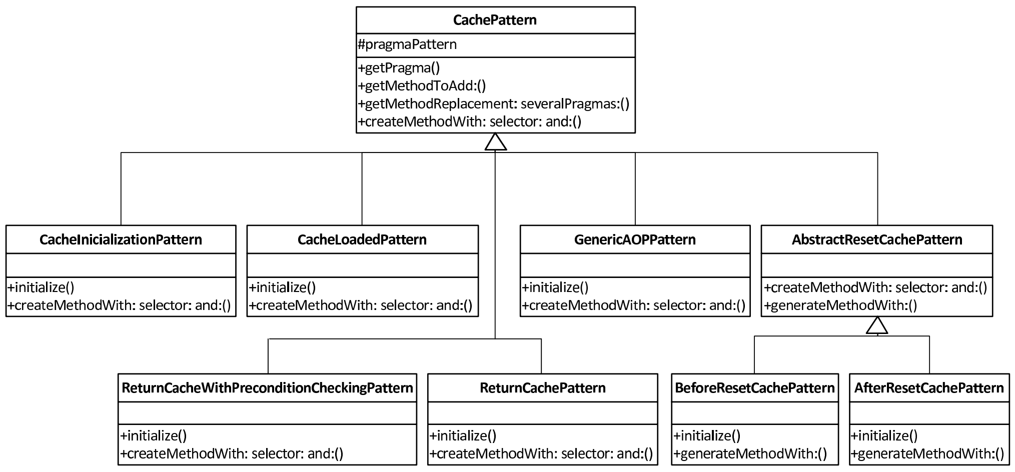

CachePattern
#pragmaPattern
+getPragma()
+getMethodToAdd:()
+getMethodReplacement: severalPragmas:()
+createMethodWith: selector: and:()
CacheInicializationPattern
+initialize()
+createMethodWith: selector: and:()
CacheLoadedPattern
+initialize()
+createMethodWith: selector: and:()
GenericAOPPattern
+initialize()
+createMethodWith: selector: and:()
AbstractResetCachePattern
+createMethodWith: selector: and:()
+generateMethodWith:()
ReturnCacheWithPreconditionCheckingPattern
+initialize()
+createMethodWith: selector: and:()
ReturnCachePattern
+initialize()
+createMethodWith: selector: and:()
BeforeResetCachePattern
+initialize()
+generateMethodWith:()
AfterResetCachePattern
+initialize()
+generateMethodWith:()

Figure 3. Pattern hierarchy.

```
MOGraphElement>>isCacheLoaded
   <CacheLoadedPattern: #cacheForm>
```

Using restructurings based on the patterns, the *Cache Concern* is refactorized properly in more than 85% of the methods of the `MOGraphElement` hierarchy that uses one or more caches. Some uses of the caches are not encapsulated by means of cache patterns due to that (1) the code belongs to a cache pattern but the code related with the cache is tangled with the main concern, or (2) the code does not match with any of the described patterns. For example, the following method

```
MOGraphElement>>nodeWith: anObject ifAbsent: aBlock
   | nodeLookedUp |
   lookupNodeCache ifNil: [ lookupNodeCache :=
     IdentityDictionary new ].
   lookupNodeCache at: anObject ifPresent: [ :v | ^ v ].
   nodeLookedUp := self nodes detect: [:each | each
     model = anObject ] ifNone: aBlock.
   lookupNodeCache at: anObject put: nodeLookedUp.
   ^ nodeLookedUp
```

could not be refactored because the cache *lookupNodeCache* is used to make different computations across the whole method by which is closely tied to the main concern. These uses of the caches that are not encapsulated by using the described patterns are also refactored by means of annotations. For these cases a *Generic AOP* pattern is used. The used annotations have the structure $<$*cache: cacheName before: beforeCode after: afterCode* $>$ where *cache* indicates the name of the cache to be injected. The before and after clauses indicate the source code that will be injected and when it will be injected in regard to the execution of the method. That is to say, the code inside the original method will be replaced by the code pointed out in the before clause of the annotation, a call to the new method will be added, and the code contained in the after clause of the annotation will be added at the end. For example, the refactorization of the method presented previously is

```
MOGraphElement>>nodeWith: anObject ifAbsent: aBlock
   <cache: #lookupNodeCache before:'   lookupNodeCache
     ifNil: [lookupNodeCache := IdentityDictionary new ].

   lookupNodeCache at: anObject ifPresent: [ :v | ^ v ].
   ^lookupNodeCache at: anObject put: (' after: ' )'>
   | nodeLookedUp |
   nodeLookedUp := self nodes detect: [:each | each
     model = anObject ] ifNone: aBlock.
   ^ nodeLookedUp
```

As we see, all the code with references to the cache *lookupNodeCache* are encapsulated into the before clause of the annotation.

4. Results

The use of the presented patterns is used to compose the caches behavior improving the maintenance of the system. In this line, the contribution of the approach is twofold. First, the mechanism of encapsulation and injection could be used to refactor the current Mondrian caches (and also those ones that may be introduced in the future) improving the code reuse. Second, the code legibility is increased because the *Cache Concern* is extracted from the main concern leaving a cleaner code.

The cache composition is achieved during the injection phase. As the different pieces of code that are related to the cache are encapsulated by means of the patterns restructurings, an implicit process of division of the complexity of the caches behavior is achieved. That is to say, this kind of approach helps the developer by splitting the caches behavior in small fragments of code. These fragments of code are encapsulated by the patterns restructurings and they are finally composed during the injection phase. For example, the functionality related to the cache *absoluteBoundsCache* is refactored by the patterns *Reset Cache, Lazy Initialization*, and *Cache Initialization*.

One of the main priorities of the refactoring is to not affect the performance of the system. For this reason a group of benchmarks were measured in order to evaluate the cache performance when a set of nodes and edges are displayed. The variations of performance between the system before and after applying refactorings that we observe are not significant. That is because, in general, the code after the injection of the caches is the same that the original code before the Mondrian refactoring. There were only minor changes such as the reorder of statements in some methods (without changes in the behavior) and the deletion of methods with repeated code. Figure 4 shows the details of the benchmarks results, in which the time execution to the nodes and edges visualization were calculated. The results of both benchmarks were average over a total of 10 samples. As we see, as was expected, there are not remarkable variations during these displaying.

Using cache in the main logic. This experience has been the opportunity to think again on the implementation of Mondrian. We found one occurrence where a cache variable is not solely used as a cache, but as part of main logic of Mondrian. The method *bounds* contains an access to `boundsCache`:

```
MOGraphElement >>bounds
   ...
   self shapeBoundsAt: self shape ifPresent: [ :b | ^
     boundsCache := b ].
   ...
```

```
MOGraphElement >>translateAbsoluteCacheBy: aPoint
   absoluteBoundsCache ifNil: [ ^ self ].
   absoluteBoundsCache := absoluteBoundsCache
     translateBy: aPoint
```

The core of Mondrian is not independent of the cache implementation. The logic of Mondrian relies on the cache to implement its semantics. This is obviously wrong and this is situation is marked as a defect[7].

Singularity of *`#displayOn:`* Displaying a node uses all the defined caches to have a fast rendering. We were not able to define `displayOn:` as the result of an automatic composition. The main problem is that this method uses intensively the cache to load and save data during its execution. For this reason, the code related to the cache is very scattered across the method making the restructuration by mean of cache patterns almost unviable. So, this method was restructured using the Generic AOP pattern.

Reordering. The injection mechanism may reorder statements in the instrumented method. This is the case of the reset method (which was presented in the previous section). As shown, in this case the caches are resetted at the beginning of the method and after that the methods *resetElementsToLookup* and *resetMetricCaches* are invoked in contrast with the original method in which the former was invoked at the beginning and the former at the end. Even though the order of calls is changed, the behavior of the method is not modified. The consistent behavior was manually and automatically checked.

5. Related Work

Our approach is not particularly tied to our code weaver. An approach called AspectS has been proposed for Squeak Smalltalk [7]. AspectS is a general purpose AOP language with dynamic weaving. Unfortunately, it does not work on Pharo, the language in which Mondrian is written. A new aspect language for Pharo is emerging[8], we plan to use it in the future.

Several approaches have been presented in order to refactor and migrate object-oriented systems to aspect-oriented ones. Some of these approaches use a low level of granularity focusing on the refactorization of simple languages elements such as methods or fields [1, 3, 5, 14, 16]. On the other hand, other approaches are focused on a high level of granularity. This kind of approaches tries to encapsulate into an aspect an architectural pattern that represents a cross-cutting concern. That is, these approaches are focused on the refactorization of a specific type of concern. Our work is under this category.

Others works that deal with the refactorization in a high level of granularity are discussed next. Da Silva et al. [4] present an approach of metaphor-driven heuristics and associated refactorings. The refactorization of the code proposed is applicable on two concerns metaphors. A heuristic represents a pattern of code that is repeated for a specific concern and it is encapsulated into an aspect by means of a set of fixed refactorings.

Van der Rijst et al. [11, 15] propose a migration strategy based on crosscutting concern sorts. With this approach the cross-cutting concerns are described by means of concern sorts. In order to refactor the code, each specific CCC sort indicates what refactorings should be applied to encapsulate it into an aspect.

Hannemman et al. [6] present a role-based refactoring approach. Toward this goal the cross-cutting concerns are described using abstract roles. In this case the refactorings that are going to be used to encapsulate a role are chosen by the developer in each case. Finally, AOP has been used for some mechanisms of cache in the past. Bouchenak et al. [2] present a dynamic web caching content approach based on AOP. In order to achieve this goal, a set of weaving rules are specified using AspectJ as aspect-oriented language. In this same line, Loughran and Rashid [10] propose a web cache to evaluate an aspect-oriented approach based on XML annotations.

[7] `http://code.google.com/p/moose-technology/issues/detail?id=501`

[8] `http://pleiad.cl/phantom`

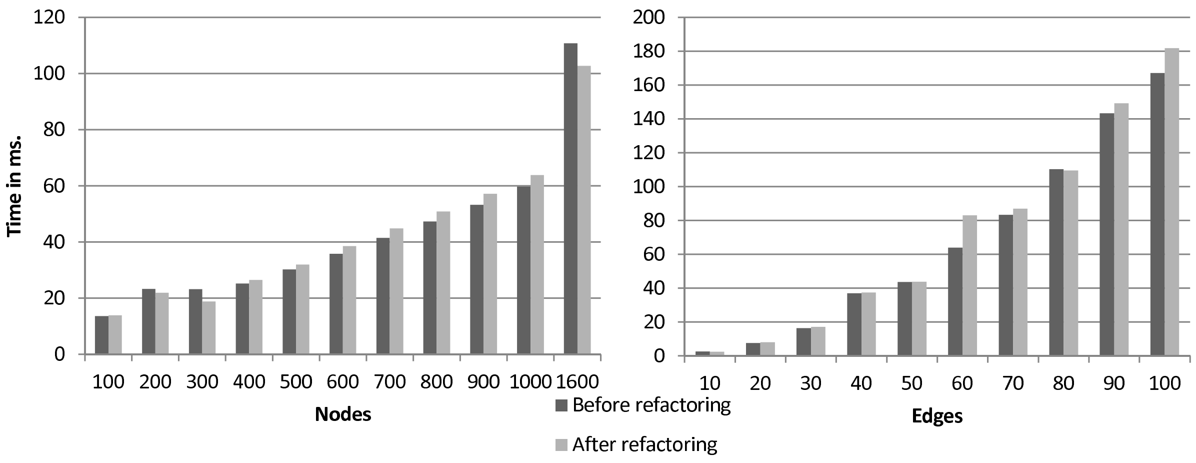


Figure 4. Benchmark of performance.

6. Conclusion

This paper presents a software evolution problem in which early made decisions become less relevant. We have solved this problem by using aspects to encapsulate and separate problematic code from the base business code. The refactoring has been realized without a performance cost. All Mondrian memoization implementations have been refactored into a dedicated aspect.

References

[1] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella. Automated refactoring of object oriented code into aspects. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 27–36, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2368-4. doi: http://dx.doi.org/10.1109/ICSM.2005.27.

[2] S. Bouchenak, A. L. Cox, S. G. Dropsho, S. Mittal, and W. Zwaenepoel. Caching dynamic web content: Designing and analysing an aspect-oriented solution. In M. van Steen and M. Henning, editors, *Middleware*, volume 4290 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2006. ISBN 3-540-49023-X. URL `http://dblp.uni-trier.de/db/conf/middleware/middleware2006.html#BouchenakCDMZ06`.

[3] M. Ceccato. Automatic support for the migration towards aspects. In *CSMR '08: Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, pages 298–301, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-1-4244-2157-2. doi: http://dx.doi.org/10.1109/CSMR.2008.4493331.

[4] B. C. da Silva, E. Figueiredo, A. Garcia, and D. Nunes. Refactoring of crosscutting concerns with metaphor-based heuristics. *Electron. Notes Theor. Comput. Sci.*, 233:105–125, 2009. ISSN 1571-0661. doi: http://dx.doi.org/10.1016/j.entcs.2009.02.064.

[5] J. Hannemann, T. Fritz, and G. C. Murphy. Refactoring to aspects: an interactive approach. In *eclipse '03: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 74–78, New York, NY, USA, 2003. ACM. doi: http://doi.acm.org/10.1145/965660.965676.

[6] J. Hannemann, G. C. Murphy, and G. Kiczales. Role-based refactoring of crosscutting concerns. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 135–146, New York, NY, USA, 2005. ACM. ISBN 1-59593-042-6. doi: http://doi.acm.org/10.1145/1052898.1052910.

[7] R. Hirschfeld. Aspects - aspect-oriented programming with squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *NetObjectDays*, volume 2591 of *Lecture Notes in Computer Science*, pages 216–232. Springer, 2002. ISBN 3-540-00737-7. URL `http://dblp.uni-trier.de/db/conf/jit/netobject2002.html#Hirschfeld02`.

[8] A. Kellens, K. Mens, and P. Tonella. A survey of automated code-level aspect mining techniques. *Transactions on Aspect-Oriented Software Development (TAOSD)*, IV (Special Issue on Software Evolution):Springer-Verlag, 2007.

[9] M. Lanza and S. Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, Sept. 2003. doi: 10.1109/TSE.2003.1232284. URL `http://scg.unibe.ch/archive/papers/Lanz03dTSEPolymetric.pdf`.

[10] N. Loughran and A. Rashid. Framed aspects: Supporting variability and configurability for aop. In *ICSR*, volume 3107 of *Lecture Notes in Computer Science*, pages 127–140. Springer, 2004. ISBN 3-540-22335-5. URL `http://dblp.uni-trier.de/db/conf/icsr/icsr2004.html#LoughranR04`.

[11] M. Marin, A. Deursen, L. Moonen, and R. Rijst. An integrated crosscutting concern migration strategy and its semi-automated application to jhotdraw. *Automated Software Engg.*, 16(2):323–356, 2009. ISSN 0928-8910. doi: http://dx.doi.org/10.1007/s10515-009-0051-2.

[12] M. Meyer, T. Gîrba, and M. Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis'06)*, pages 135–144, New York, NY, USA, 2006. ACM Press. doi: 10.1145/1148493.

1148513. URL http://scg.unibe.ch/archive/papers/Meye06aMondrian.pdf.

[13] I. Sommerville. *Software Engineering*. Addison Wesley, sixth edition, 2000.

[14] P. Tonella and M. Ceccato. Refactoring the aspectizable interfaces: An empirical assessment. *IEEE Transactions on Software Engineering*, 31(10):819–832, 2005. doi: http://doi.ieeecomputersociety.org/10.1109/TSE.2005.115. URL http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/TSE.2005.115.

[15] R. van der Rijst, M. Marin, and A. van Deursen. Sort-based refactoring of crosscutting concerns to aspects. In *LATE '08: Proceedings of the 2008 AOSD workshop on Linking aspect technology and evolution*, pages 1–5, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-147-7. doi: http://doi.acm.org/10.1145/1404953.1404957.

[16] A. van Deursen, M. Marin, and L. Moonen. A systematic aspect-oriented refactoring and testing strategy, and its application to jhotdraw. *CoRR*, abs/cs/0503015, 2005.

MDE-based FPGA Physical Design

Fast Model-Driven Prototyping with Smalltalk

Ciprian Teodorov
Lab-STICC MOCS, CNRS UMR 3192
ciprian.teodorov@univ-brest.fr

Loïc Lagadec
Lab-STICC MOCS, CNRS UMR 3192
loic.lagadec@univ-brest.fr

Abstract

The integrated circuit industry continues to progress rapidly deepening the gap in between the technological breakthroughs and the electronic design automation industry. This gap is even more problematic in the context of physical design, the last automation level between applications and the technology. The challenges of meeting the physical and performance constraints needs innovation at the algorithmic level, and at the methodological level.

This study presents a methodological approach to physical design automation relying on model-driven engineering. Relying on the flexibility, and adaptability of the Smalltalk environment we propose an agile framework enabling fast physical design tool-flow prototyping. We illustrate our approach by using the Madeo FPGA toolkit as a legacy codebase that is incrementally changed to adopt this model-driven development strategy.

Some pragmatic achievements are presented to illustrate the principal axes of this approach: algorithmic improvements through plug-and-play routines, domain-model extension for emerging technologies, as well as model evolution towards a meta-described environment.

Categories and Subject Descriptors D.2.11 [*Software Architectures*]

General Terms Design, Management

Keywords Model-Driven Engineering, Smalltalk, Physical Design,

1. Introduction

Object-oriented design enabled, for years, the development of feature-reach software environments, which propose flexible and extensible solutions for complex problem spaces, such as electronic design automation (EDA). Madeo framework[21, 24] is such an environment, automating the complete design flow of FPGA-based[1] application development, from high-level application synthesis to the physical design of the circuit layout. Result of 15 years of proactive research focusing on different aspects of the FPGA CAD[2] flow, the Madeo infrastructure cannot be described as a single software solution. Instead, it is a compilation of a number of different projects, which correspond to the research topics under investigation at different times during its existence.

The high-complexity of today's reconfigurable architectures poses huge constraints in terms of automation. To tame this complexity, the FPGA application synthesis is decomposed in a number of relatively independent problems, that are to be solved and the results integrated into the final solution. Most of these problems are NP-complete and typically solved heuristically. Moreover the optimization objectives vary and the literature is very rich in solutions highly optimized for specific configurations.

To survive in such a demanding context, a software solution has to be very adaptable, flexible and it must evolve over time. Exploiting the advantages of Smalltalk language and environment, the Madeo infrastructure provides an one stop shopping-point for FPGA application synthesis and related problems[10, 27]. From an evolution perspective, three principal evolution axes were identified:

- Extending the domain model to integrate new domain-specific concepts;
- Adding missing functionality to improve the domain coverage and the automation level;
- Adding new functions, similar to the ones already present, to give the user a choice, especially when we speak about heuristic solutions.

The OO design methodology[47] provide a largely accepted mechanism for extending the domain model, the inheritance. The design of the Madeo infrastructure, as a framework, exploits inheritance for easily adding new domain concepts while maintaining a high-degree of cohesion

IWST'11 23rd August 2011, Edinburgh.

[1] Field Programmable Gate Array

[2] Computer Aided Design

and code sharing. However in terms of adding new behavior, the OO design methodology is not clear on the preferred way of functionally extending a system. Moreover into a research context, the underlining idea is to rapidly produce prototypes focused on specific research needs. In consequence, after years of evolution and improvement, our software environment got bloated with many different projects (at different maturity levels). And even more, some parts of the system that got completely lost during the numerous integration steps.

From a software engineering perspective, our principal goal is to preserve our code-base, which gives us a strategic advantage over other research groups. This legacy code-base drives the innovation in our group, by providing high-end solutions for many practical EDA problems. This enables us to focus on novel and challenging aspects of the integrated circuit design and automation. But at the same time we need a methodological solution to solve the functional evolution aspects by isolating the core functionality and exposing clear interfaces. Moreover, we believe that such a methodological solution will open the toolbox making it accessible to a larger user base.

During the last years, model-driven engineering (MDE) and component-based development have become some of the most promising methodological approaches for tackling the software complexity and evolution problems. Based on the success stories of the MDE community, we focused our attention on finding an evolution methodology fitting our constraints, especially in terms of legacy-code reuse. Even though UML[3] became the de-facto standard for model-based methodologies, in the case of our software projects, largely based on Smalltalk philosophy and tools, we believe that the Smalltalk community is still providing the right answers. The success of the Moose platform[36] for software and data analysis provided us with inspiration and insight for creating the software development and evolution infrastructure presented in this work.

In this study, we present some conceptual, as well as practical, results in terms of software development methodology for tackling the embarrassingly complex problems of integrated circuit (IC) physical design automation. To the best of our knowledge, except our preliminary study in[50], this proposition is the first MDE-based approach addressing the physical design automation problem. The principal contributions of this study are:

- *A novel conceptual framework*, based on the MDE methodology, for the physical design aspects of the EDA flow. This framework stresses on the need of decoupled software artifacts, and isolated concern-oriented software modules. Moreover, from the practical point of view, this methodology doesn't impose any constraint on the implementation language or formalism of the core algorithms and heuristics needed, which is very important given the computational- and memory-intensive solutions needed in our context.
- *An abstract vocabulary* for structurally describing the applicative and architectural artifacts involved in the flow. The use of this abstraction level enables the creation of a common API that can be used for transversal features, such as domain agnostic manipulations, querying, model serialization, etc.
- *A novel way for conceptually describing, characterizing and implementing physical design algorithms* (e.g. placement, routing) using the transformation metaphor. These algorithms are viewed as model-to-model composite transformations organized as hierarchical directed-acyclic graphs. This view helps the algorithm designer to focus on the specific problem to be solved while enabling fine-grained reuse of software.

The remaining of this study dives into the details of this work as follows. Section 2 presents the state of the art in terms of MDE and circuit design automation. Section 3 starts by presenting the enabling technologies and experiences before showing the principal characteristics of our methodology. Section 4 further describe our approach focusing on the domain-specific aspects of FPGA physical synthesis. Section 5 shows practical experiences of our agile approach to legacy refactoring. This process was used to progressively adopt the presented MDE methodology in our framework, without losing functionality at any step during the evolution process. Section 6 reviews the principal ideas presented in this study, before concluding with some future directions.

2. Related Works

This section has the dual role of introducing model-driven engineering and circuit design automation to the readers as well as briefly presenting the state of the art in both fields.

2.1 Model-driven development

Model-driven approach provides a methodology to tackle the complexity of software development using abstraction, problem decomposition, and separation of concerns. This methodology places models and transformations at the center of the software system development. A model can be seen as an abstract and simplified way to describe and understand complex systems. According to [33] a **transformation** is ”the automatic generation of one(or more) target model(s) from one(or more) source model(s), according to a transformation definition. A **transformation definition** is a set of rules that together describe how a model in the source language can be transformed into a model in the target language.”

The massive adoption of the model-driven methodology was favored principally by the standardization of Unified Modeling Language (UML) and Model-Driven Architec-

[3] Unified Modeling Language

ture(MDA) by the Object Management Group (OMG)[8]. Modeling languages such MOF, EMOF[38] or Ecore[46] as well as the development of environments like Eclipse Metamodeling Framework[46] have been largely adopted as the *de-facto* standard for model-driven software engineering. Efforts, like Kermeta[35], proposed ways to ”breathe life into (your) metamodels”, by adding an executable meta-language, for operational semantics specification, to the contemplative (structural) side of model-driven development.

But the OMG is not the only actor improving on the state of the art of this field. The orthogonal classification architecture, proposed by Atkinson et al. [1, 2], provides a new insight into some possible future metamodeling environments, able to solve some of the current limitations.

The works from the Software Composition Group, proposing the use of Smalltalk as a ”reflexive executable meta-language”[11] for ”meta-modeling at runtime”[19], address some more pragmatic issues around model-driven engineering, e.g. the use of a single language – Smalltalk in this case – for all the modeling aspects; the possibility to dynamically change the meta-model at runtime; the need to have executable meta-descriptions that alleviates the need for code generation.

In the context of this study, we have adopted a hybrid approach based on the FAME meta-model for domain modeling, and an ad-hoc Smalltalk-based behavior and algorithm specification approach, relying on the ”transformation” metaphor, implemented using the visitor design pattern and isolated software components.

2.2 Electronic Design Automation

One of the principal problems addressed by electronic design automation (EDA) tools is the compilation of behavioral system specifications, in languages like C, Matlab to hardware systems[28]. From this perspective the EDA flow is typically decomposed in two compilation steps: high-level synthesis, and physical synthesis. The high-level synthesis (HLS) tools automatically convert behavioral descriptions of the system into optimized hardware designs, described using languages like VHDL or Verilog, which are then compiled to technology-specific logic netlists. The physical-design tools are used to map the technology-mapped netlists to electrically correct physical circuits. In the context of this work we are looking at FPGA physical synthesis flow, which is composed of the following principal steps: partitioning, floorplanning, placement, and routing.

2.2.1 HLS

UML extensions (profiles) such as SysML[51], Embedded UML[31], and MARTE[52] have been proposed for the embedded system design. These profiles allow the specification of systems using high-level models. Mapping UML concepts to the target language syntax low-level (i.e. VHDL) code is generated. In [14, 43], for example, the authors propose a HLS and design space exploration flow based on the MDE methodology, which based on successive refinements of a high-level model generates VHDL hardware accelerators.

Moreover, concerns such as interchange or debug are, mainly, considered during HLS. As an example, RedPill[23] supports probe-based validation of running applications. Probes are inserted in the high-level code, but also appear in the generated code, and bring controllability to the hardware under execution. RedPill makes use of domain modeling (in that case, the application) through Platypus[41], a STEP/EXPRESS framework that offers both a standard way of modeling the domain, and interchange facilities. In particular, Platypus has been used in the scope of the Morpheus FP6 Project[26] to support multi-target and cross-environment synthesis/compilation. The output was a netlist to be further processed by low-level tools (physical design).

2.2.2 FPGA Physical Design

The Field Programmable Gate Array (FPGA) is a flexible computing architecture that bridges the gap between Application Specific Integrated Circuits (ASIC) and General Purpose Processors (GPP) by providing a customizable hardware fabric capable of implementing wide range of applications. FPGA designs trade the performance of ASICs for flexibility and fabrication costs[20].

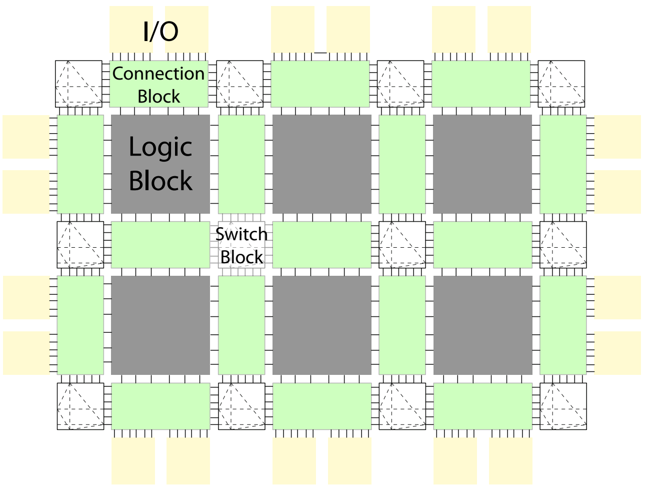


Figure 1. The typical structure of a island-style FPGA.

The basic structure of a FPGA is a regular mesh of logic blocks interconnected by a reconfigurable routing architecture. In the case of Island-style FPGAs the routing resources are evenly distributed throughout the mesh, the logic block usually having routing channels all around, see Figure 1. [4]

For FPGA physical-synthesis, a number of tools like VPR[3] and Madeo[21] aim at providing generic frameworks targeting reconfigurable architecture. They are based on high-level models, which ease the architecture and application description (based on specific DSLs[37]), and using

[4] In the scope of this paper when we say FPGA we are actually meaning Island-style FPGA

generic heuristics (simulated annealing[18] - placement) and algorithms (Pathfinder[32] - routing) they automate the application mapping on the FPGA architectures. The principal limitation of these tools is the difficulty to extend them functionally by plugging-in new algorithms.

2.2.3 Madeo and software evolution problems

Madeo[24] is a design suite for the exploration of reconfigurable architectures. It includes a modeling environment that supports multi-grained, heterogeneous architectures with irregular topologies. Madeo framework initially allows to model FPGA architectures. The architecture characteristics are represented as a common abstract model. Once the architecture is defined, the Madeo CAD tools can be used to map a target netlist on the architecture. Madeo embeds placement and routing algorithms (the same as VPR[3]), a bitstream generator, a netlist simulator, and a physical layout generator. It supports architectural prospection and very fast FPGA prototyping. Several FPGAs have been modeled, including some commercial architectures (such as Xilinx Virtex family), and prospective ones (such as STMicro LPPGA). Based on Madeo infrastructure further research projects emerged such as DRAGE[40], that virtualizes the hardware platform and produces physical layouts as VHDL descriptions.

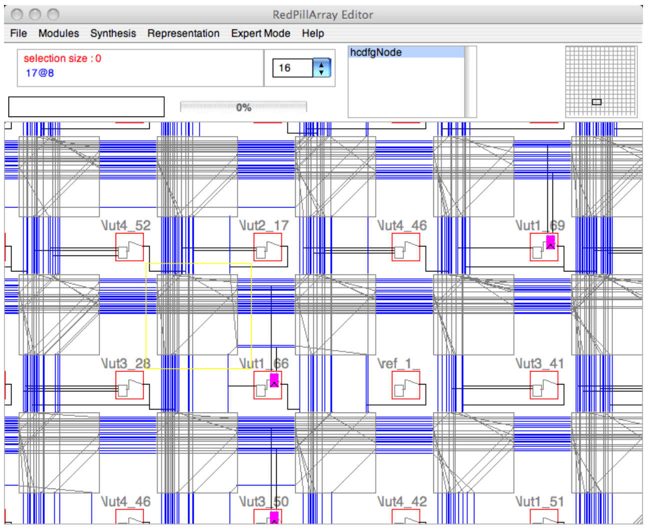


Figure 2. Island style FPGA in Madeo

The Madeo infrastructure has three parts that interact closely (bottom-up):

- *Domain model and its associated generic tools*. The representation of practical architectures on a generic model enables sharing of basic tools such as place and route (P&R), allocation, circuit edition[21]. Figure 2 illustrates MADEO on an island style FPGA. Specific atomic resources, such as operators or emerging technologies, can be merged with logic, since the framework is extensible.
- *High-level logic compiler (HLLC)*. This compiler produces circuit netlists associated to high-level functionalities mapped to specific technology models. Leveraging object-oriented programming flexibility in terms of operators and types, the HLLC produces primitives for arbitrary arithmetic or symbolic computing.
- *System and architecture modeling*.The framework enables the description of static and dynamic aspects specific to different computing architectures, like: logic primitives, memory, processes, hardware-platform management, and system activity.

The compiler uses logic generation to produce configurations, binds them to registers or memories, and produces a configured application. The control over the place and route tools enables building complex networks of fine or medium grain elements.

3. Model-based EDA in Smalltalk

In 1996, when the first developments of the Madeo framework began, object oriented (OO) software for EDA could not be imagined. The executing environments provided by virtual machines was considered too slow for solving the hard combinatorial optimization problems of circuit design. The journey of creating a OO CAD framework for FPGA design automation, using the Smalltalk environment, was challenging. On the way we showed that dynamically typed languages can serve for creating high density logic designs[9, 42], we proposed the first virtual FPGA prototyping environment[25] and we proved that, harnessing the power of OO software design, we could create a flexible yet competitive EDA toolkit that even today enables breakthrough research in the field[22, 40]. Meanwhile OO design became widely accepted by the EDA community through languages like C++(i.e. systemC[39]) and Java(i.e. JHDL[16]). But OO design suffers from a number of problems, common in software engineering, especially from a software evolution perspective. As the target domain evolves the software tools used modeling, simulation and reasoning about the domain have to evolve accordingly. And without a methodological approach for evolution, systems become unmaintainable and, even more, functionality is lost through numerous integration cycles. Model-driven engineering promises a number of solutions for this problem by abstracting and decoupling the different software artifacts, most notably through the use of aspect-driven programming and component-based design. In consequence, today we assist at yet another step in the evolution of EDA industry, the move towards model-driven and component based design, especially in the context of embedded system development and high-level synthesis.

During the last years Smalltalk world has undergone tremendous evolutions, through truly open environment like Pharo[4], language innovations such as the adoption traits for fine-grained reuse[12], language boxes for embedded DSLs[44] and runtime meta-modeling frameworks like

FAME[19]. Moose project experience report[13], as well as our experience with Platypus[26] led us to create an agile MDE-based prototype, targeting FPGA physical design automation, using the Smalltalk environment. Our goal is to create an evolution-aware platform, relying on our legacy code-base, that can evolve to accommodate new technological requirements.

The most important aspects that need to be addressed by such a toolkit are: domain modeling, domain-specific languages, code reuse, legacy and fast prototyping, and external tools integration. In the following paragraphs we will review each of these aspects and briefly present some Smalltalk technologies that can address each issue.

Domain modeling is at the core of any EDA toolkit. It enables the expression of domain specific aspects by creating a common vocabulary that will then be exploited to model different systems. In the context of rapidly changing IC technology, the domain model has to evolve rapidly, eventually changing its internal structure. For addressing this problem the FAME meta-modeling library[19] proposes a solution by offering run-time access to the meta-information, appearing in a meta-model. By creating a bidirectional causal connection between the meta-level and the implementation-level the model updates can be reflected in-between the two levels at runtime. In Section 3.1 we will present in details a FAME-based abstract meta-model that is used throughout our framework as a common base for creating the different domain models needed.

Domain-specific languages are used extensively in the context of circuit design automation. Their principal roles are: describing the IC architecture (Hardware Description Languages (HDL)), describing the different physical, technological, and design constraints, describing the functional requirements, specifying test cases and objectives, describing the application netlists, etc. In the context of our framework, currently, we have implemented a dozen different parsers for interfacing with other tools and we have a proprietary HDL used for FPGA architecture description and instantiation. The SmaCC-based[6] parser descriptions are difficult to modify according to the evolving needs of the EDA field. The parser-combinator libraries such as PetitParser along with the laguage-box concept implemented in the Helvetia DSL development environment[44] provide a new solution to this problem. Helvetia enables the seamless creation of embedded DSLs, while PetitParser brings inheritance to grammar specifications, thus offering the possibility to isolate the grammar specifications from the abstract syntax tree creation. These developments provide a smart way for defining a concrete text-based syntax for the instantiation of our domain models. It should be noted that due to the high number of hardware modules included into a typical architecture, visual languages are not well suited for the architecture description. Moreover IC designers are very proficient using textual description languages.

Code reuse is the most important concern from a software evolution point of view. OO methodology provides inheritance and polymorphism as *de-facto* solutions for enabling large-scale code reuse. With the adoption of traits[12], in Smalltalk dialects such as Squeak and Pharo, the OO toolbox for reuse gained a new tool. Traits are a mechanism for fine-grained method reuse that decouples functional method implementation from the object state.

Legacy and Fast prototyping As stated in the introduction, a big issue lies in the reuse of our legacy code-base. Moreover, any evolution methodology has to enable incremental development, such as to be able to go on using our environment during the whole evolution process. We bootstrapped the prototype by reusing the MADEO project infrastructure, and incrementally moved towards our new MDE-based framework. In Section 5 some of the evolution steps are presented as to illustrate our approach.

External tools integration is a high-level reuse mechanism by which tools developed by third-parties can be integrated into the toolkit. Historically we have been using inter-process communication (via UnixProcess class) and FFI (such as DLLCC) for executing external programs and interfacing with our toolkit. The Alien FFI, proposed for Newspeak language[5] and adopted by Pharo, provides a new OO-friendly way of interfacing Smalltalk environment with the external world. Moreover, the transformation metaphor, proposed for algorithm design, opens the toolkit even more by enabling a fine-grained external tool reuse, via the *External* atomic transformation.

The remaining part of this section presents our practical approach for abstract domain modeling and a novel conceptual framework for designing and implementing algorithms based on the transformation metaphor.

3.1 Fame-based Domain Modeling

From a modeling point of view most of the EDA tools are structured around hierarchical models appearing at different levels of abstraction. These models are used to represent the two principal axes of EDA:

- *Application* The applications are typically seen as composition of operators, that can be further broke-down into simpler constructs towards elementary systems primitives. The behavior specifications can be treated at different abstraction levels: i.e. control data flow graphs, with hierarchies representing processes and/or different functions used to implement the application, combinatorial or sequential logic, used to describe the system behavior as composition of simple Boolean functions, etc.
- *Hardware* The hardware designers rely heavily on hierarchical descriptions to simplify the integrated circuit development. As for the applications, the hardware is described hierarchically at different abstraction levels. At the system level, for example, an IC can be seen as an ar-

ray of blocks, each of which implements a specific functionality (i.e. processors, GPUs[5], network modules, etc.). Each of these blocks can then be decomposed into its own building blocks (memory, logic, interconnection, etc.). At the logic level, a digital circuit can be seen as a composition of logic blocks, implementing different functions according to the Boolean equations characterizing the blocks. At the circuit level the system is again specified hierarchically, using transistors, diodes, etc. as primitives.

Based on these observations, we created an abstract meta-model that is used to structurally describe our domain as a hierarchical composition of primitive elements interconnected together. This abstraction describes mainly a hierarchical annotated port-graph. Using the FAME metamodeling framework was straightforward creating a Smalltalk implementation of this high-level domain-specific modeling language.

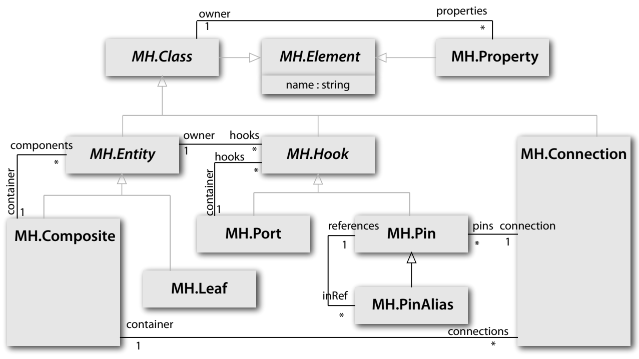


Figure 3. A view of the core structure of the proposed meta-model

In the proposed meta-model, see Figure 3, the domain structure is captured as a *Composite* that contains one or more *Entity* instances (called ”entities” from now on). *Entity* is an abstract construct specialized as *Leaf* and *Composite*. The *Leaf* instances are domain primitives which are viewed as the indivisible building blocks of more sophisticated structures. The entities have hooks (*Hook*) which provide an external interface through which they can be interconnected together. The abstract *Hook* concept is specialized as *Pin* and *Port*. The *Pin* element allows connection between entities. It can be a physical contact, or a logical interface depending on the specification level or on the refinement degree. The role of *Port* instances is to group together and structure the *Pin* instances, they can be seen as namespaces structuring the access to particular pins. The *Connection* purpose is to glue together the entities of a particular system by linking pins together. These connections can be interpreted as wires, communication channels, or simply as relations between entities.

All these concepts are refinements of the *Class* element which owns *Property* instances. The *Property* represent different attributes particular to the modeled domain. Some examples of properties are: the position of an entity on a layout, the capacity or resistance of a wire modeled as a Connection instance, the logical value of a circuit pin represented as a *Pin* instance.

The meta-model, presented in Figure 3, represents a common abstraction which can be used to specify any kind of interconnected system having any number of hierarchical components that present the system at different abstraction levels. This core structure is refined in Section 4.1 to describe an island-style FPGA architecture and in Section 4.2 to capture the characteristics of a combinatorial circuit.

3.2 Transformation Metaphor for Algorithm Design

Besides structural domain modeling, algorithm design is the most important aspect of any EDA toolkit. Since almost all the optimization problems encountered in electronic CAD tools are NP-hard in terms of complexity, most of the practical solutions rely on heuristics. Hence, the main concern of EDA tools users and designers is the heuristic performance in terms of execution time, memory requirements, and solution optimality. It is commonly accepted for an EDA tool to run for days or weeks on high-end machines having huge memory resources. In terms of implementation, these optimization heuristics are very complex. Most of them are implemented in highly optimized C, for performance issues.

The Madeo framework itself relies on external tools, mainly for logic synthesis and technology mapping. But it implements generic heuristics for the physical design problem to assure the flexibility of the mapping. The idea is that these generic heuristics can be used in the context of different FPGA architecture with the minimum human intervention for optimization goals parametrization.

To address the algorithm design problem in this study we propose a technique that we call ”Transformation metaphor”. This technique appears as a conceptual framework; the algorithm designer now looks at the algorithm implementation as it was a model transformation problem.

This approach mainly reifies the implicit tight dependency between algorithms and domain models, through explicit transformations, isolating their respective concerns, thus increasing the flexibility and the expressivity of our toolkit.

In the transformation metaphor each algorithm (or heuristic) is seen as a hierarchical composite transformation. Figure 4 presents the different types of transformations proposed by this approach. The primitive transformation types being: *concern extraction*, and *atomic transformations*.

The *concern extraction* represents the mapping from a domain model to simpler models required by the algorithm being implemented. The purpose of the concern extraction is to decouple the domain specific details of the model from the algorithm-specific details.

From an implementation perspective, the *concern extraction* is nothing more than implementing a visitor (accord-

[5] graphical processing unit

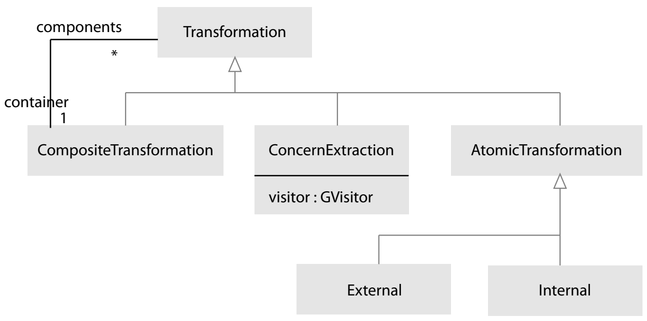


Figure 4. Transformation metaphor model

ing to the visitor design pattern[15]) that will iterate over the domain model extracting information and instantiating an algorithm-specific model.

The *atomic transformations* represent actual algorithms (or heuristic) needed to solve the problem. According to the specific needs it can further decomposed in more fine grain modules and composed as using the transformation metaphor or it can be directly implemented in a programming language on demand. The only requirement is to export clear interfaces so that it can be integrated as an *atomic transformation* in the framework.

This approach has the advantage of being able to integrate external tools implemented using any programming language and/or computing model. Currently we are working on formalizing these concepts into a concrete Smalltalk based transformation engine, which will be able to provide the users with an environment for algorithm design and integration into domain specific tool-flows.

4. Smalltalk MDE: Applications to FPGA Physical Design

The purpose of this section is to show how the Smalltalk-based MDE methodology (presented in the previous section) is used in the context of the FPGA physical design.

The physical design step is responsible for allocating all design components for creating the configuration bitstream for FPGAs. This means that each gate of the application netlist will be assigned a spatial location (placement). Then, the interconnection signals will be reified using appropriate routing structures. Physical design has a direct impact on the circuit characteristics (performance, area, power, etc). The main steps of physical design are: partitioning, floorplanning, placement, and routing[17].

Figure 5 shows a typical FPGA physical synthesis flow. It starts from the description of a particular FPGA architecture as an architectural model (ArchM) instance, and a target application as an application model (AppM) instance. These models are domain specific, fully independent from any algorithms that are used for physical design. The output is a refined ArchM instance, configured to implement the AppM instance.

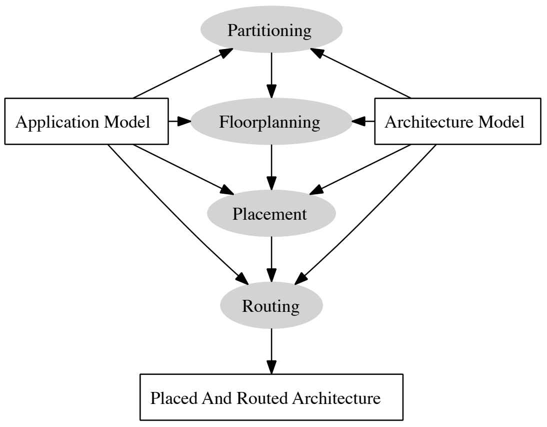


Figure 5. A standard physical design flow. The rectangles represent models, while the ellipses represent algorithms.

In the following sections we introduce two different specializations of the hierarchical port-graph model, introduced in the Section 3.1, the ArchM and the AppM. Then in Section 4.3 we present the tool-flow modeling methodology based on the "transformation metaphor", used for implementing the four steps of the FPGA physical design flow.

4.1 Island style FPGA model

To capture the particularities of FPGA architectures (see Figure 1), the meta-model, presented in Section 3.1, was refined. Figure 6 shows the domain-specific concepts added. Using this meta-model the FPGAs are modeled based on 3 primitives (specializations of MHLeaf): Logic Block, Switch, and Connection Block. *LogicBlock* instances represent the programmable logic blocks which provide the basic computation and storage elements. The routing architecture is modeled using *Switch* and *ConnectionBlock* instances. These primitives contain a number of *internalConnections* that specify the way their pins are connected internally (see Figure 7). *Tile* instances aggregate these primitives into logical bricks which are replicated regularly to form a *Cluster*. The primitive's pins are aliased at the tile level and cluster level, using *InterfacePin* instances, to expose the possible connection points.

Figure 7 presents a view of the principal types of internal connections. *HardConnect* can be seen as a direct connection between components without any physical property (it is used for example to represent long wires which logically are composed of different wire segments, the connection between these wire segments is a HardConnect instance). *Wire* represents a physical wire. *ProgrammableConnect* is an abstract representation of electrically configurable components.

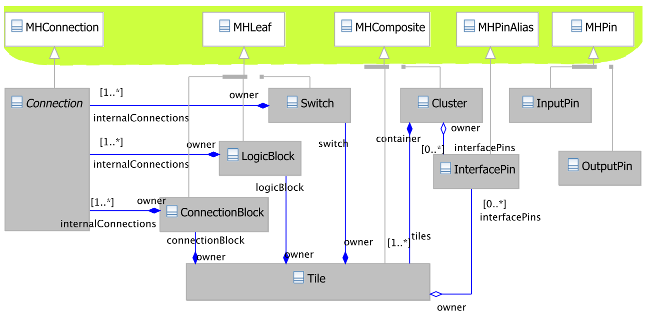


Figure 6. A simplified view of the Island-style FPGA extension of the core meta-model

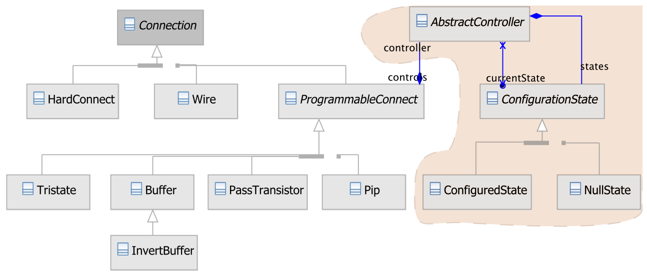


Figure 7. Connection hierarchy

In our framework the FPGA (re-)configuration process is seen from two different perspectives:

- *Fine-grained configuration*, that represents the detailed configuration of all available resources. The target FPGA configuration bitstream is generated based on these information. At this level all details of the target FPGA are captured, and set in the final configuration state.
- *Functional configuration*, that represents a an abstract, functional view of the fine-grain configurations. This view is used to speed up some of the physical design algorithms by providing them with a condensed representation of the target architecture. i.e. at this level the wires in a routing channel are seen as a virtual connection having a certain capacity and an occupancy.

For representing the configuration state of any FPGA element we rely on using a configuration controller described based on the state-machine design pattern (the shaded area in Figure 7). These fine-grain controllers represent the real configurability of each physical component and are managed as a whole based on a global configuration controller. The global configuration controller corresponds to the physical configuration controller present on the FPGA circuit.

The FPGA meta-model (ArchM) presented in this section is not intended to exhaustively cover the programmable architecture design concepts, nor to be used as a real-life FPGA meta-model but rather is a simplified version that capture the basic properties of interest for the purpose of this study.

4.2 Combinatorial Logic Meta-Model

In this section we present another refinement of the meta-model, presented in Section 3.1, this time focusing on hierarchically description of combinatorial logic.

In digital electronics, combinatorial logic is a type of digital logic implemented by Boolean circuits where the logic network computes a pure function based solely on the inputs. As opposed to sequential logic that relies on memory elements to implement state that is used, besides the inputs, for computing the outputs.

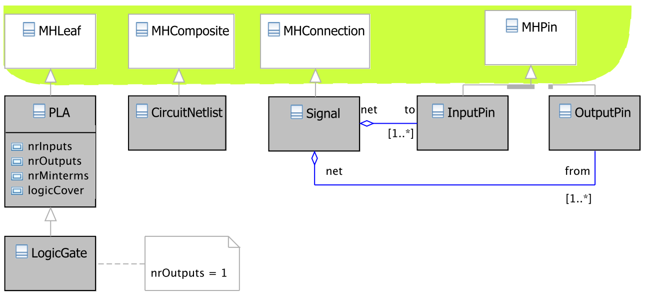


Figure 8. Combinatorial Logic extension of the core meta-model

The combinatorial logic meta-model (AppM), presented in Figure 8, uses two-level single-output logic functions (*LogicGate*) to model the application netlist.

As for the FPGA model, presented in the previous section, only a simplified view of Combinatorial Logic meta-model is presented.

4.3 Tool-flow Modeling

The flow is an endogenous transformation realized via four composite transformations (see Figure 5): partitioning, floorplanning, placement, and routing. Each of these four transformations is a composition of more-elementary transformations. Thus, the physical synthesis tool flow is a hierarchical directed acyclic graph (DAG) of transformations, where the nodes represent the transformations to be done, and the edges represent the dependencies between the tools. The execution of the transformations happens in topological order from the DAG inputs (ArchM and AppM instances) to the outputs (AppM instance mapped on the ArchM instance).

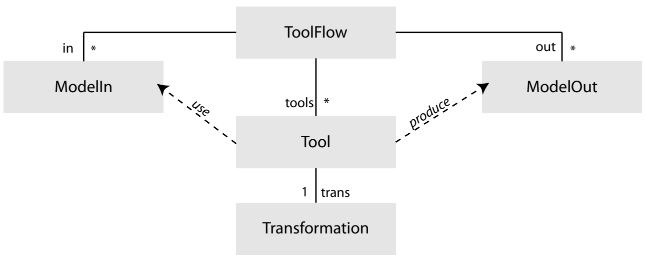


Figure 9. The abstract toolflow meta-model

Figure 9 shows the abstract toolflow model. A *ToolFlow* instance takes a number of *ModelIn* as input and produces a number of *ModelOut* as output using any number of tools to do it. Each tool is uniquely associated with a *Transformation* (either composite, or atomic). In our case the a *ToolFlow* instance is created with AppM and ArchM models as inputs (*in*) and four *Tool* instances corresponding to the four automation steps. The result produced would be a refined ArchM instance.

The principal advantage of this flow is the capacity to easily replace any of the physical synthesis algorithms with different implementations, with no constraint on the implementation language or formalism. But this also has a drawback, the high number of simple transformations (concern extraction) needed to implement the flow. However, such a view of the physical design automation, that isolates and encapsulates the different steps of the flow, poses the bases for future standardization, which that can cut down the number of intermediate transformations.

5. From Legacy to MDE toolkit – Successful Experiences

This section provides an account for some of the steps we followed towards the MDE-based framework. Moreover it shows the flexibility of the approach as the environment remained usable during the whole evolution process.

5.1 Improving on Legacy – First steps

The first modifications that were integrated in the toolkit focused on improving some of the optimization routines already present in our legacy code-base, namely the floorplanning and routing routines.

In the case of the floorplanning routine, we have chosen to replace the TCG-based[29] heuristic present by an improved version relying on a different floorplan representation, namely TCG-S[30]. From the implementation perspective we tried to decouple as much as possible the heuristic from our domain models, so that it can be reused in other context with no modifications. The integration into the toolkit was done by redirecting the automation flow towards the newly created module. Concern extraction was used to instantiate the TCG-S specific floorplan model from the AppM using ArchM geometrical information. Once the floorplan model instantiated, the optimization goals (metrics) where added as closures (Smalltalk blocks) independent of the heuristic implementation.

For the routing routine we refactored the existing routing algorithm (Pathfinder[32]) decoupling it from the architectural model with which it had numerous dependencies, and we created a transformation-based version. As for the TCG-S algorithm, the architecture specific optimization goals are set using closures. The results using this new implementation were impressive in terms of execution speed (over 40% faster), principally due to the possibility to prune the routing resource graph, thus reducing considerably ($\geq$50%) the number of nodes explored during the execution. One negative aspect of using this approach is the increase in the memory footprint due to the duplication of some ArchM aspect in the routing specific model.

5.2 Extensions for Nanoscale Physical Design

In [10, 27] the extensibility of the MADEO framework was put to a test for the first time with the advent of emerging technologies. The core concepts of the NASIC fabric[34], see Figure 10, were introduced into the framework, and a reconfigurable nanoscale architecture, called NFPGA, was designed. This required to extend both the reconfigurable architecture model and its associated tools in such a way that NASIC can be modeled and programmed. Process that goes through several steps:

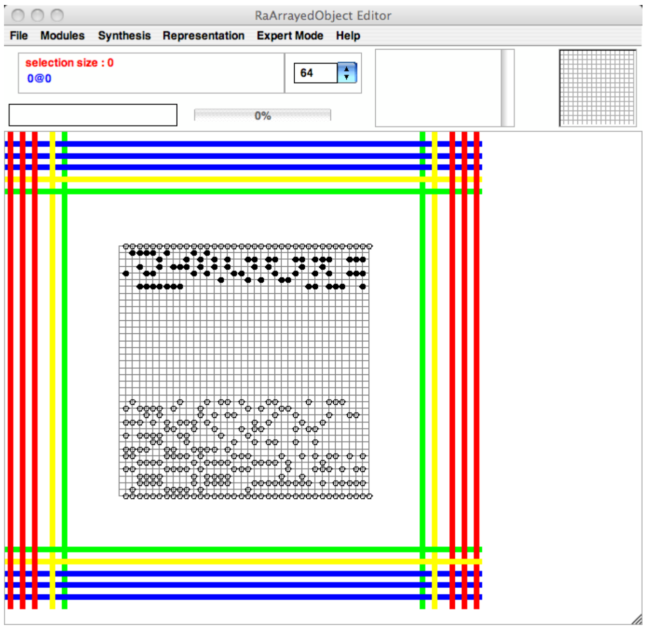


Figure 10. Madeo viewer on an nanoscale tile instance

1. The generic model must evolve to describe new potential components (nanogrid, nanowire, etc. . .) with specific characteristics.
2. This generic model must be instantiated using a proprietary HDL. As the HDL expresses the model, any in-depth change within the model leads to an evolution of the HDL(i.e. new keywords).
3. Some algorithms must be adapted or substituted for technology-specific counterparts while preserving the API. For example, the logical functions are implemented using a 2 level logic rather than FPGAs LUTs or processor μ-instruction.

More recently[48, 49] the methodology presented in this study was used to propose a complete physical synthesis tool-flow for a new nanoscale architecture template. As it can be seen in the Figure 11 the Madeo toolkit legacy was

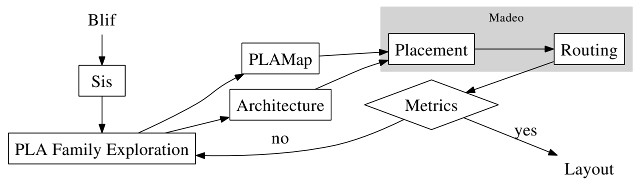


Figure 11. The R2DNasic CAD flow.

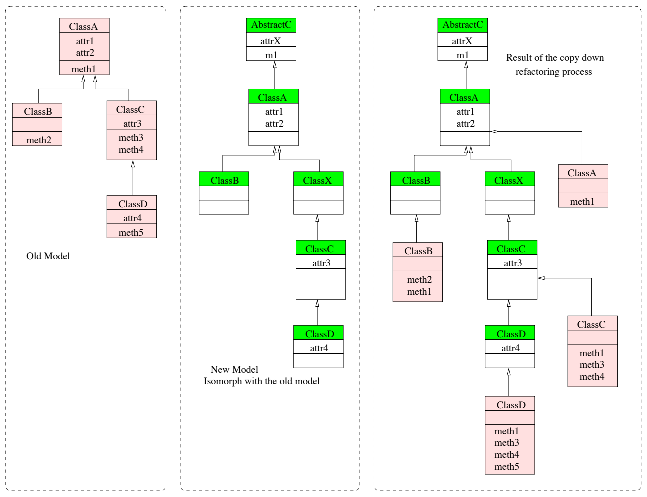


Figure 12. Example model transformation using Copy-Down method

used for placement and routing (as well as for archM description, instantiation and visualization), external tools like Sis[45], PLAMap[7] where seamlessly integrated with new internal tools for pla family exploration, and metric computing. Different tool-flows were created using these tools, each one having different optimization goals, and working on different architecture variants. Moreover by opening the toolbox the design-space exploration (DSE) was bootstrapped relying on standard reconfigurable place & route routines, thus enabling a baseline evaluation which showed the need for more optimized routing. Once the new routing algorithm was developed it was integrated into a new tool-flow, specializing the baseline tool-flow via inheritance.

The main conclusion of this experiment is that using this MDE approach effective incremental design-space exploration is enabled, and a new tool-flow exploration axis is added to the typical application/architecture trade-off, while the tool-flow specialization reduces the development effort.

5.3 Refactoring Domain-Models

The most extensive evolution of our legacy code-based was the replacement of old domain-specific OO models with a newly engineered set of FAME-based domain models, relying on the hierarchical port-graph abstraction, described in Section 3.1. The preservation of legacy functionality is the principal constraint in this case.

To this purpose we engineered the new models to replicate the old-model entities and then we merged the two models in such a way to factorize the available functionality of the two.

Two automated methods of merging the two models were devised: copydown method, and doesnotunderstand method. They are both explained in detail in the following paragraphs along with their advantages and constraints.

CopyDown Method Starts by inlining the calls to super methods in order to obtain inheritance hierarchy independent methods that can be safely copied to all subclasses of a specific node. After the inlining step the CopyDown step proceeds where all superclass methods will be copied recursively to all subclasses. The next step is to remove the duplicated instance variables from the old model. This step is required because the new model already contains some instance variables and they will be accessible from the future superclasses. This step being done the old class hierarchy is destroyed, and the new designed superclasses are assigned to the old model classes. Because the two models have some classes with identical names the name clash is prevented by isolating the classes in different namespaces. Figure 12 shows the result of this refactoring method applied on an example. The different colors in class representation represent the different namespaces that isolate the classes with identical name. The rightmost diagram shows the method duplication through the classes of the old hierarchy in order to preserve their inherited functionality.

This refactoring method is a good solution to the model-refactoring problem encountered in the development of the framework since the old functionality is maintained, the old applications developed around the old model continue working without any modification, and the new model can be used freely without any execution delay. Another advantage of this method is that it does not change the execution mechanism of the underlying platform, and so it can be used almost unmodified with all OO languages. Still this method has its drawbacks principally because the classes of the old model contain duplicated behavior. This duplication decreases the maintainability of the system, and renders the old model entities less comprehensible.

A solution can be the introduction of another refactoring step that will push up in the new hierarchy the equivalent methods.

But, despite its drawbacks, this method can still be used provided that some necessary precautions are taken:

- If the tools using the old model are mature enough so they can be used in their actual state without modifications;
- If the developers intention is to replace the old model with the new one in all the tools using it. In this case the CopyDown method can be used as an intermediate evolution step, where some tools are ported to the new model and others are still using the old one.

DoesNotUnderstand Method Another method for merging the two models into one directly perform the last two steps of the previous technique (remove the instance variables; Cut the superclass link. Add the new classes as the superclass of all old classes) without copying-down the superclass methods. The old hierarchy must be stored (in a Dictionary for example) in order to be able to replicate the old hierarchy method inheritance. To be able to use the behavior declared in the old model classes, the #doesNotUnderstand method will be redefined. Once the error message is intercepted we start searching in the old inheritance hierarchy to find the implementation class of that message. If we find it we send the message to that class and return the result to the sending class. If the message was not found we simply throw the "does not understand" error.

This method solves the previous problems related to the message duplication throughout the old hierarchy. But it comes with new drawbacks like:

- The execution mechanism of the object oriented framework must be modified. While this is possible in an open context like Smalltalk in most object-oriented environments will be difficult to implement this method.
- Since all missing methods need to be searched in the old inheritance hierarchy this will add some overhead to the overall execution of code using this model.
- If the new model classes implement one of the methods implemented in a class of the old hierarchy, say class X, all subclasses of the class X will execute the new implementation of the method instead of executing the implementation in class X. Thus rendering the tools using the old model unusable. That happens because the "does not understand" error will not be triggered once a method with an identical signature is found in the new inheritance hierarchy.

At the end we want to emphasize that the evolution process presented in this study may not be possible in non-Smalltalk environments mainly due to the lack of reflectivity, openness and flexibility. But at the same time imposing a certain development methodology might be hard mainly due to the same reasons. At times during the experiences described in this section we found ourselves wanting to hack our way into simpler (or faster) solutions by instantiating new objects as needed, creating unnecessary dependencies or using shared state just because its possible.

In conclusion, we believe that, for a Smalltalk developer, at times it is best to resist the temptation of instant gratification and take a step back to reflect on the overall system design.

6. Conclusion and Future Works

The Madeo FPGA toolkit served as trustworthy physical design back-end for enabling our research group to innovate in the challenging field of EDA. But after 15 years of various project specific modifications, and numerous integration cycles our environment started to degrade, and at time functionality was lost. We identified that an outdated, adhoc prototype-driven development process was at the core of these degradations.

In this study we present a solution to this problem based on a mix of MDE methodology coupled with the last innovation of the Smalltalk community. Practically our approach relies on the FAME executable meta-modeling framework the development of domain-specific models, and the decoupling of the optimization algorithms from these models using the "transformation" metaphor.

Applying this methodology to our legacy code-base not only improves the software architecture of our solution, but it also opens the toolkit enabling plug-and-play algorithm reuse. Relying on these developments we were able to easily prototype and test different automated solutions targeting new nanoscale architectures.

In conclusion, we believe that the key to taming the complexity of today IC design is our capacity to reuse the results of the last 50 years of high-end EDA research.

In the future, we plan to improve our methodology by formalizing our transformation model and by developing a transformation engine able to harness the advantages of MDE while minimizing its apparent shortcomings most notably in terms of memory consumption.

References

[1] C. Atkinson and T. Kühne. Concepts for comparing modeling tool architectures. In L. C. Briand and C. Williams, editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 398–413. Springer, 2005.

[2] C. Atkinson, M. Gutheil, and B. Kennel. A flexible infrastructure for multilevel language engineering. *IEEE Trans. Software Eng.*, 35(6):742–755, 2009.

[3] V. Betz, J. Rose, and A. Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA, USA, 1999. ISBN 0792384601.

[4] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, 2009. ISBN 978-3-9523341-4-0. URL `http://pharobyexample.org`.

[5] G. Bracha, P. Ahe, V. Bykov, Y. Kashai, and E. Miranda. The newspeak programming platform. Technical report, Cadence Design Systems, 2008.

[6] J. Brant and D. Roberts. SmaCC, a Smalltalk compiler-compiler, 2011. URL `http://www.refactoryworkers.com/SmaCC`.

[7] D. Chen, J. Cong, M. Ercegovac, and Z. Huang. Performance-driven mapping for cpld architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 22(10):1424 – 1431, oct. 2003. ISSN 0278-0070.

[8] S. Cranefield and M. Purvis. UML as an Ontology Modelling Language. In *In Proceedings of the Workshop on Intelligent*

Information Integration, 16th International Joint Conference on Artificial Intelligence (IJCAI-99, pages 46–53, 1999.

[9] C. Dezan, L. Lagadec, and B. Pottier. Object oriented approach for modeling digital circuits. In *Microelectronic Systems Education, 1999. MSE '99. IEEE International Conference on*, pages 51 –52, 1999.

[10] C. Dezan, C. Teodorov, L. Lagadec, M. Leuchtenburg, T. Wang, P. Narayanan, and A. Moritz. Towards a framework for designing applications onto hybrid nano/cmos fabrics. *Microelectron. J.*, 40(4-5):656–664, 2009. ISSN 0026-2692.

[11] S. Ducasse and T. Gîrba. Using Smalltalk as a reflective executable meta-language. In *International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006)*, volume 4199 of *LNCS*, pages 604–618, Berlin, Germany, 2006. Springer-Verlag.

[12] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, Mar. 2006. ISSN 0164-0925.

[13] S. Ducasse, T. Girba, A. Kuhn, and L. Renggli. Meta-environment and executable meta-language using Smalltalk: an experience report. *Software and Systems Modeling*, 8:5–19, 2009. ISSN 1619-1366.

[14] A. Gamatié, É. Rutten, H. Yu, P. Boulet, and J.-L. Dekeyser. Model-Driven Engineering and Formal Validation of High-Performance Embedded Systems. *Scalable Computing: Practice and Experience (SCPE)*, 10, 2009.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[16] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting. A cad suite for high-performance fpga design. In *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '99, pages 12–, Washington, DC, USA, 1999. IEEE Computer Society.

[17] A. Kahng, J. Lienig, I. Markov, and J. Hu. *VLSI Physical Design: From Graph Partitioning to Timing Closure*. Springer, 2011.

[18] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[19] A. Kuhn and T. Verwaest. FAME, a polyglot library for metamodeling at runtime. In *Workshop on Models at Runtime*, pages 57–66, 2008.

[20] I. Kuon, R. Tessier, and J. Rose. Fpga architecture: Survey and challenges. *Found. Trends Electron. Des. Autom.*, 2:135–253, February 2008. ISSN 1551-3076.

[21] L. Lagadec. *Abstraction and modélisation et outils de cao pour les architectures reconfigurables*. PhD thesis, Université de Rennes 1, 2000.

[22] L. Lagadec and D. Picard. Software-like debugging methodology for reconfigurable platforms. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1 –4, may 2009.

[23] L. Lagadec and D. Picard. Smalltalk debug lives in the matrix. In *International Workshop on Smalltalk Technologies*, IWST '10, pages 11–16, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0497-9.

[24] L. Lagadec and B. Pottier. Object-oriented meta tools for reconfigurable architectures. In *Reconfigurable Technology: FPGAs for Computing and Applications II, SPIE Proceedings 4212*, 2000.

[25] L. Lagadec, D. Lavenier, E. Fabiani, and B. Pottier. Placing, routing, and editing virtual fpgas. In G. Brebner and R. Woods, editors, *Field-Programmable Logic and Applications*, volume 2147 of *Lecture Notes in Computer Science*, pages 357–366. Springer Berlin / Heidelberg, 2001.

[26] L. Lagadec, D. Picard, and B. Pottier. *Dynamic System Reconfiguration in Heterogeneous Platforms*, chapter 13. Spatial Design : High Level Synthesis. Springer, 2009.

[27] L. Lagadec, B. Pottier, and D. Picard. Toolset for nano-reconfigurable computing. *Microelectronics Journal*, 40(4-5):665 – 672, 2009. European Nano Systems (ENS 2007); International Conference on Superlattices, Nanostructures and Nanodevices (ICSNN 2008).

[28] L. Lavagno, G. Martin, and L. Scheffer. *Electronic Design Automation for Integrated Circuits Handbook - 2 Volume Set*. CRC Press, Inc., Boca Raton, FL, USA, 2006. ISBN 0849330963.

[29] J.-M. Lin and Y.-W. Chang. Tcg: a transitive closure graph-based representation for non-slicing floorplans. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 764–769, New York, NY, USA, 2001. ACM. ISBN 1-58113-297-2.

[30] J.-M. Lin and Y.-W. Chang. TCG-S: Orthogonal Coupling of P*-Admissible Representations for General Floorplans. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 842–847, New York, NY, USA, 2002. ACM. ISBN 1-58113-461-4.

[31] G. Martin, L. Lavagno, and J. Louis-Guerin. Embedded uml: a merger of real-time uml and co-design. In *Proceedings of the ninth international symposium on Hardware/software codesign*, CODES '01, pages 23–28, New York, NY, USA, 2001. ACM.

[32] L. McMurchie and C. Ebeling. Pathfinder: A negotiation-based performance-driven router for fpgas. In *Field-Programmable Gate Arrays, 1995. FPGA '95. Proceedings of the Third International ACM Symposium on*, pages 111 – 117, 1995.

[33] T. Mens and P. V. Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152: 125 – 142, 2006. ISSN 1571-0661. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005).

[34] C. A. Moritz, T. Wang, P. Narayanan, M. Leuchtenburg, Y. Guo, C. Dezan, and M. Bennaser. Fault-Tolerant Nanoscale Processors on Semiconductor Nanowire Grids. *IEEE Transactions on Circuits and Systems I, special issue on Nanoelectronic Circuits and Nanoarchitectures*, november 2007.

[35] P. A. Muller, F. Fleurey, and J. M. Jézéquel. Weaving Executability into Object-Oriented Meta-Languages. In *LNCS*, Montego Bay, Jamaica, Oct. 2005. MODELS/UML'2005, Springer.

[36] O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York NY, 2005. ACM Press. ISBN 1-59593-014-0. Invited paper.

[37] N. Oliveira, M. J. V. Pereira, P. R. Henriques, and D. da Cruz. Domain specific languages: A theoretical survey. *In Proceedings of the 3rd Compilers, Programming Languages, Related Technologies and Applications (CoRTA'2009)*, 2009.

[38] omg. *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006.

[39] P. R. Panda. Systemc: a modeling platform supporting multiple design abstractions. In *Proceedings of the 14th international symposium on Systems synthesis*, ISSS '01, pages 75–80, New York, NY, USA, 2001. ACM. ISBN 1-58113-418-5. doi: http://doi.acm.org/10.1145/500001.500018.

[40] D. Picard. *Méthodes et outils logiciels pour l'exploration architecturale d'unité reconfigurable embarqueés*. PhD thesis, Université de Bretagne Occidentale, Brest, 2010.

[41] A. Plantec and V. Ribaud. PLATYPUS : A STEP-based Integration Framework. In *14th Interdisciplinary Information Management Talks (IDIMT-2006)*, pages 261–274, Tchèque, République, Sept. 2006.

[42] B. Pottier and J.-L. Llopis. Revisiting smalltalk-80 blocks: a logic generator for fpgas. In *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, pages 48 –57, apr 1996.

[43] I. R. Quadri, H. Yu, A. Gamatie, E. Rutten, S. Meftali, and J.-L. Dekeyser. Targeting reconfigurable fpga based socs using the uml marte profile: from high abstraction levels to code generation. *International Journal of Embedded Systems*, 4 (3/4):204–224, 2010.

[44] L. Renggli. *Dynamic Language Embedding With Homogeneous Tool Support*. Phd thesis, University of Bern, Oct. 2010.

[45] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Technical Report UCB/ERL M92/41, EECS Department, University of California, Berkeley, 1992.

[46] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley Professional, 2 edition, Jan. 2008.

[47] B. Stroustrup. What is object-oriented programming? *Software, IEEE*, 5(3):10 –20, may 1988. ISSN 0740-7459. doi: 10.1109/52.2020.

[48] C. Teodorov and L. Lagadec. Fpga sdk for nanoscale architectures. In *6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC'11)*, 2011.

[49] C. Teodorov, P. Narayanan, L. Lagadec, and C. Dezan. Regular 2d nasic architecture and design space exploration. In *Nanoscale Architectures, IEEE / ACM International Symposium on (NanoArch'11)*, 2011.

[50] C. Teodorov, D. Picard, and L. Lagadec. Fpga physical-design automation using model-driven engineering. *6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC'11) 6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC'11)*, 2011.

[51] Y. Vanderperren and W. Dehaene. UML 2 and SysML: An Approach to Deal with Complexity in SoC/NoC Design. In E. European design and Automation Association, editors, *Design, Automation and Test in Europe DATE'05*, volume 2, pages 716–717, Munich Allemagne, 03 2005. Submitted on behalf of EDAA (http://www.edaa.com/).

[52] J. Vidal, F. de Lamotte, G. Gogniat, P. Soulard, and J.-P. Diguet. A co-design approach for embedded system modeling and code generation with uml and marte. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 226–231, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

Efficient Proxies in Smalltalk

Mariano Martinez Peck[1 2] Noury Bouraqadi[2] Marcus Denker[1]
Stéphane Ducasse[1] Luc Fabresse[2]

[1]RMoD Project-Team, Inria Lille–Nord Europe / Université de Lille 1
[2]Université Lille Nord de France, Ecole des Mines de Douai

marianopeck@gmail.com, {stephane.ducasse,marcus.denker}@inria.fr,
{noury.bouraqadi,luc.fabresse}@mines-douai.fr

Abstract

A proxy object is a surrogate or placeholder that controls access to another target object. Proxy objects are a widely used solution for different scenarios such as remote method invocation, future objects, behavioral reflection, object databases, inter-languages communications and bindings, access control, lazy or parallel evaluation, security, among others.

Most proxy implementations support proxies for regular objects but they are unable to create proxies for classes or methods. Proxies can be complex to install, have a significant overhead, be limited to certain type of classes, etc. Moreover, most proxy implementations are not stratified at all and there is no separation between proxies and handlers.

In this paper, we present Ghost, a uniform, light-weight and stratified general purpose proxy model and its Smalltalk implementation. Ghost supports proxies for classes or methods. When a proxy takes the place of a class it intercepts both, messages received by the class and lookup of methods for messages received by instances. Similarly, if a proxy takes the place of a method, then the method execution is intercepted too.

Keywords Object-Oriented Programming and Design » Message passing control » Proxy » Interception » Object Swapping » Smalltalk

1. Introduction

A proxy object is a surrogate or placeholder that controls access to another target object. A large number of scenarios and applications [11] have embraced and used the Proxy Design Pattern [12].

Proxy objects are a widely used solution for different scenarios such as remote method invocation [24, 25], distributed systems [3, 20], future objects [23], behavioral reflection [10, 15, 29], aspect-oriented programming [16], wrappers [6], object databases [7, 19], inter-languages communications and bindings, access control and read-only execution [1], lazy or parallel evaluation, middlewares like CORBA [13, 17, 28], encapsulators [22], security [27], among others.

Most proxy implementations support proxies for regular objects (instances of common classes) only. Some of them, *e.g.,* Java Dynamic Proxies [11, 14] even requires that at creation time the user provides a list of *Java interfaces* for capturing the appropriate messages.

Creating uniform proxies for not only regular objects, but also for classes and methods has not been considered. In existing work, it is not possible for a proxy to take the place of a class and a method and still intercept messages, in order to perform operations such as logging, swapping or remote class interaction. This weakness strongly limits the applications of proxies.

In addition, traditional implementations (based on error handling [22]) result in non stratified proxies: not all the proxified API messages can be trapped leading to severe limits, and there is no clear division between trapping a message and handling it, *i.e.,* there is no separation between proxies and handlers. Trapping a message is intercepting it, and handle a message means to do something in particular with such interception. The handling actions depends on the user needs, hence they are defined by the user of the framework. Bracha et al. [5] defined stratification in the field of reflection as the following statement: "meta-level facilities must be separated from base-level functionality". The same applies for proxies, where instead of meta-level facilities there are trapping or intercepting facilities [27].

Another interesting property of proxy implementations is memory footprint. As any other object, proxies occupy memory and there are cases in which the number of proxies and their memory footprint becomes a problem.

In this paper, we present Ghost, a uniform, light-weight and stratified general purpose proxy model and its implementation in Pharo Smalltalk [4]. In addition, Ghost supports proxies for classes or methods. This means that it is not only possible to create a proxy for a class or a method but also that such proxy takes the place of the target original class or method, intercepts messages without crashing the system. If a proxy takes the place of a class it intercepts both, messages received by the class and lookup of methods for messages received by instances. Similarly, if a proxy takes the place of a method, then the method execution is intercepted too. Ghost provides low memory consuming proxies for regular objects as well as for classes and methods.

The contributions of this paper are:

- Describe and explain the common proxy implementation in dynamic languages and specially in Smalltalk.
- Define a set of criteria to evaluate and compare proxies implementations.
- Present Ghost, a new proxy model and implementation which solves most of the proxy's problems in a uniform, light-weight and stratified way.
- Evaluate our solution with the defined criteria.

The remainder of the paper is structured as follows: Section 2 defines and unifies the vocabulary and roles that are used throughout the paper, and then it presents the list of criteria used to compare different proxy implementations. Section 3 describes the typical proxy implementation and by evaluating it against the previously defined criteria, it presents the problem. Section 4 introduces and discusses the Ghost model, and then evaluates the needed language and VM support. An introduction to Smalltalk reflective model and its provided hooks is explained by Section 5. Ghost implementation is presented in Section 6, which also provides an evaluation of Ghost implementation based on the defined criteria. Finally, in Section 7 related work is presented, before concluding in Section 8.

2. Vocabulary and Proxy Evaluation Criteria

2.1 Vocabulary and Roles

For sake of clarity, we define here the vocabulary used throughout this paper. We hence make explicit entities in play and their respective roles.

Target. It is the original object that we want to *proxify*, i.e. the object that will be replaced by a proxy.

Client. This is an object which uses or holds a reference on the target object.

Interceptor. It is an object whose responsibility is to *intercept* messages that are sent to it. It may intercept some messages or all of them.

Handler. The handler is responsible of *handling* messages caught by the interceptor. By *handling* we refer to whatever the user of the framework wants to do with the interceptions, *e.g.,* logging, forwarding the messages to the target, control access, etc.

One implementation can use the same object for taking the roles of interceptor and handler. Hence, the proxy plays as interceptor and also as handler. In another solution such roles can be achieve by different object where the proxy usually takes the role of interceptor.

2.2 Proxies Implementation Criteria

From the implementation point of view, there are criteria that can be taken into account to compare and characterize a particular implementation [10]:

Stratification. Stratification means that there is a clear separation between the proxy support and application functionalities. With a stratified approach, all messages sent by the application's business objects to the proxy are intercepted.

The proxy API should not pollute the application's namespace. In a truly stratified proxy, all messages received by a proxy should be intercepted. This means that the handler itself cannot send messages to the proxy. Not only the handler cannot do that, but none other object in the system. Having this stratification is important to achieve security and to fully support transparency of proxified object for the end-programmers [5].

Stratification also covers the design of the proxy. There are two responsibilities in a proxy toolbox: 1) trapping or intercepting messages (interceptor role) and 2) managing the interception (handler role), *i.e.,* performing actions once the message is intercepted. In a stratified proxy framework the first responsibility can be covered by a proxy itself, and the second one by a handler. This means that proxies are just traps to intercept messages. When they intercept a message they just delegate to a handler, which does something in particular with it, *e.g.,* logging, access control, etc. Consequently, different proxies instances can use the same or different handler instance.

Interception granularity. There are the following possibilities:

- Intercept *all* messages sent to an object, even messages not defined in the object API.
- Intercept only user defined messages.
- Intercept only messages imposed by the system.

With the last two options, there are messages that are not intercepted and hence answered by the proxy itself. This can be a problem because it is not possible to distinguish messages sent to the proxy to ones that should be trapped. For example, when a proxy is asked its class it must answer not its own class but the class of the target object. Otherwise, this can cause errors difficult to manage.

Object replacement. Replacement is making client objects to reference the proxy instead of referencing the target. Two cases can be distinguished. On the one hand, there are scenarios where some objects become new clients. So, they will get a reference to a proxy instead of the reference to the target object. For example, for remote method invocation, targets are located in a memory space different from the clients one. Therefore, clients can only hold references on proxies to interact with targets. Messages sent by clients to proxies will be handled and forwarded to remote targets.

On the other hand, sometimes the target is an already existing object which is pointed to by other objects in the system and it needs to be *replaced* by a proxy, *i.e.,* all objects in the system which have a reference on the target should be updated so that they point to the proxy instead. For instance, for a virtual memory management we need to swap out objects and to replace them by proxies. In this case, we need to retrieve all objects which were pointing to the existing unused object to now point to the proxy. We refer to this functionally as *object replacement*.

Uniformity. We refer to the ability of creating a proxy for any object (regular object, method, class, block, process. . .) and replacing the object by the proxy.

Most proxy implementations support proxies only for regular objects and *without* object replacement, *i.e.,* proxies cannot replace a class, a method, a process, etc, without crashing the system. There can be not only more *classes* that require special management but also more special objects that require so. For example objects like nil, true, false, etc.

This is an important criteria since there are scenarios where being able to create proxies for living runtime entities is mandatory.

Transparency. A proxy is fully transparent if client objects have no mean to find out whether they reference the target or the proxy. .

One of the typical problems related to transparency is the identity issue in cases where the proxy and the target are located in the same memory space. Given that different objects have different identities, a proxy's identity is different from the target's identity. The expression proxy == target will answer false, revealing hence the existence of the proxy. This can be temporary hidden if there is object replacement between the target object and the proxy. When we replace all references to the target by references to the proxy, clients will only see the proxy. However, this "illusion" will be broken as soon as the target provides its reference as an answer to a message or a parameter.

Another common problem is asking the class or a type of an object since most of the times the proxy answers its own type or class instead of the target's one. The same happens if there is special syntax or operators in the language such Javascript's "+", "/", "=", ">", etc. In order to have the most transparent possible proxy, these situations should be handled in such a way that the proxy behaves like the target.

Now the question is whether the identity of an object should be controlled similarly to central messages such as class. We believe that most of the time it is important that the identity is treated similarly to messages, since code working based on object identity should work the same whether the object has been proxified or not. Now depending on the language or optimization in place, identity is not treated as a message but provided as a built in primitive, which means that it can be difficult to offer proper identity swapping.

Efficiency. Proxy handling must be efficient from both points of view: performance and memory usage. In addition, we can distinguish between installation performance and runtime performance. For example, for installation, it is commonly evaluated if a proxy installation requires extra overhead like recompiling.

Moreover, depending on the usage, the memory footprint of the proxies can be fundamental. It is not only important the size in memory of the proxies, but also the space analysis *i.e.,* how many objects are needed per target. Only a proxy instance? A proxy instance and a handler instance?

Implementation complexity. Since at constant functionality, a simpler implementation is better, this criteria evaluates the complexity of the implementation. For example, if the proposed solution is easy to implement or if it needs complex mechanisms.

Ease of debugging. It is difficult to test and debug proxies because the debugger or the test framework usually send messages to the objects that are present in the current stack. Those messages include, for example, printing an object, accessing its instance variables, etc. When the proxy receives any of those messages it may intercept it (depending whether the proxy understands that message or not). Hence, debugging is usually complicated in the presence of proxies.

Constraints. The toolbox may require, *e.g.,* that the target implements certain interface or inherits from a specific class. In addition, it is important that the user of the proxy toolbox can easily extent or change the purpose of the proxy adapting it to his own needs.

Portability. A proxy implementation can depend on the VM or the language where it is developed which can be different in other Virtual Machines or languages.

3. Common Proxy Implementations

Even if there are different proxy implementations and solutions, there is one that is the most common among dynamic programming languages: it is based on error raising and resulting error handling. We briefly describe it and show that it fails to fulfill important requirements.

3.1 Typical Proxy Implementation

In *dynamic languages*, the type of the message's receiver is resolved at runtime. When an unknown message is sent to an object, an error exception is thrown. The basic idea is then to create objects that raise errors for all the possible messages (or a subset) and customize the error handling process.

In Smalltalk, for instance, the Virtual Machine sends the message doesNotUnderstand: to the receiver object. To avoid infinite recursion, all objects must understand the message doesNotUnderstand:. That is the reason why such method is implemented in the class Object, the root of the hierarchy chain. In Smalltalk, the default implementation throws a MessageNotUnderstood exception. Similar mechanisms exist in dynamic languages like Ruby, Python, Objective-C, Perl, etc.

Since doesNotUnderstand: is a normal method, it can be overwritten in subclasses. Hence, if we can have a minimal object and we override the doesNotUnderstand: method to do something special (like forwarding messages to a target object), then we have a possible proxy implementation. This technique has been used for a long time [20, 22] and it is the most common proxy implementation. Readers knowing this topic can directly jump to Section 3.2. Most dynamic languages provide a mechanism for handling messages that are not understood as shown in Section 7.

Obtaining a minimal object. A minimal object is that one which understands none or only a few methods. In some programming languages, the root class of the hierarchy chain (usually called Object) already contains several methods [1]. In Pharo Smalltalk, Object inherits from a superclass called ProtoObject which inherits from nil. ProtoObject understands a few messages[2]: the minimal amount of messages that are needed by the system. Here is a simple Proxy implementation in Pharo.

```
ProtoObject subclass: #Proxy
    instanceVariableNames: 'targetObject'
    classVariableNames: ''

Proxy >> doesNotUnderstand: aMessage
   |result|
   ..."Some application specific code"
   result := aMessage sendTo: targetObject.
   ..."Other application specific code"
^result
```

Handling not understood methods. This is the part of the code that is user-defined and not part of the Proxy framework itself. Common behavior include logging before and after the method, forwarding the message to a target object, validating some access control, etc. In case it is needed, it is perfectly valid to issue a super send to access the default doesNotUnderstand: behavior.

To forward a message to a target object, we need the message name and the list of parameters sent to it. The Smalltalk Virtual Machine invokes the doesNotUnderstand: aMessage with a message reification as argument. Such class specifies the method selector, the list of arguments and the lookup class (in normal messages it is the receiver's class and, for super sends, it is the superclass of the class where the method is implemented. To forward a message to another object, the class Message provides the method sendTo: anotherObject which sends such message to another object.

Notice that this solution is not limited to Smalltalk. For example, the Smalltalk's doesNotUnderstand: is in Ruby method_missing, in Python __getattr__, in Perl autoload, in Objective-C forwardInvocation:, etc. As we explain in Section 7, Objective-C provides a minimal object class called NSInvocation which understands the message invokeWithTarget:aTarget and forwards a message to another object. Example:

```
- (void)forwardInvocation:(NSInvocation *)invocation
    {
      [invocation invokeWithTarget:delegate];
    }
```

[1] Object has 338 methods in PharoCore 1.3

[2] ProtoObject has 40 methods in PharoCore 1.3

In Ruby we can do:

```
def method_missing(name, *args, &block)
    target.send(name, *args, &block)
end
```

In Python:

```
  def __getattr__(self, name):
  return getattr(self.realObject, name)
```

3.2 Evaluation

In this section we evaluate the common proxy implementation based on the criteria we provided above (see section 2.2).

Stratification. This solution is not stratified at all:

- The method doesNotUnderstand: cannot be trapped like a regular message. Moreover, when such message is sent to a proxy there is no efficient way to know whether it was because of the regular error handling procedure or because of a proxy trap that needs to be handled. In other words, the doesNotUnderstand: occupies the same namespace as application-level methods [27], hence this solution is not stratified.
- There is no separation between proxies and handlers.

Interception granularity. It cannot intercept all messages but instead *only* those that are not understood. As explained, this generates method name collisions.

Object replacement. In the common proxy implementation object replacement is usually not supported. Nevertheless, Smalltalk implementations do support it but suffer the problem of "reference leaks": the target might provide its own reference as a result of a message or a parameter. This way the client gets a reference to the target, and hence it can by-pass the proxy.

Transparency. This solution is not transparent. Proxies do understand some methods (those from its superclass) generating method name collisions. For instance, if we evaluate "Proxy new pointersTo" (pointersTo is a method implemented in ProtoObject) it answers the references to the proxy instead of intercepting the message and forward it to a target. The same happens with the identity comparison or asking the class.

Efficiency. From the CPU point of view, this solution is fast and it has low overhead. In contrast to other technologies, there is no need to recompile the application and the system libraries or to modify their bytecode, or to do other changes such as in Java modifying the environment variable CLASSPATH, the class loader. Regarding the memory usage, there is no optimization. Efficiency is not normally addressed in typical proxy implementations.

Implementation complexity. This solution is easy to implement: it just needs the doesNotUnderstand:, a minimal object, and be able to forward a message to another object.

Ease of debugging. It is not provided by this solution. The debugger sends messages to the proxy which may not be understood, and hence, delegated to a target object. This makes it hard to debug, inspect and print Proxy instances.

Constraints. This solution is flexible since target objects do not need to implement any interface or method, nor to inherit from specific classes. The user can easily extent or change the purpose of the proxy adapting it to his own needs by just reimplementing the doesNotUnderstand:.

Uniformity. This implementation is not uniform since proxies cannot be used as classes, methods, etc.

Portability. This approach impose few requirements for the language and the VM that are provided by almost all available dynamic languages. With the examples of the previous section we demonstrate that it is really easy to implement this approach in different dynamic languages.

4. The Ghost Model

This section describes and explains the Ghost proxy model. This model fits better for dynamic programming languages and it is intended to be a reference model, *i.e.,* developers from different dynamic languages can implement it. In addition, the model must clarify which are the expected requirements and hooks from the host language.

4.1 Proxies

Ghost model supports proxies for regular objects as well as for classes, methods, and any other class that requires special management. In addition, Ghost supports proxies for classes or methods. Furthermore, Ghost model distinguishes between *interceptors* and *handlers*. Proxies play solely the role of interceptors. Since we are describing the model, the design is abstract and general. The design of an implementation may look different from this model. Figure 1 shows the proxies hierarchy and the following is a quick overview of the responsibilities of each class:

ObjectProxy. This is the base class for all proxies of Ghost model and provides proxies for regular objects, *i.e.,* objects that do not need any special management. Its responsibility, as well as its subclasses, is to take care about the message interception, which is represented in Figure 1 as the method intercept(). In Ghost model, Proxies only play the role of interceptors. Proxies are instances of ObjectProxy or any of its subclasses and all they do is to forward intercepted messages to handlers. Each proxy must have an associated handler. Different proxies can use different handlers and vice versa.

Finally, note that since proxies just intercept messages and forward them to handlers, it is unlikely that the user of the framework needs to customize or subclass any of the proxy classes. What the user needs to define is what to do in the handler.

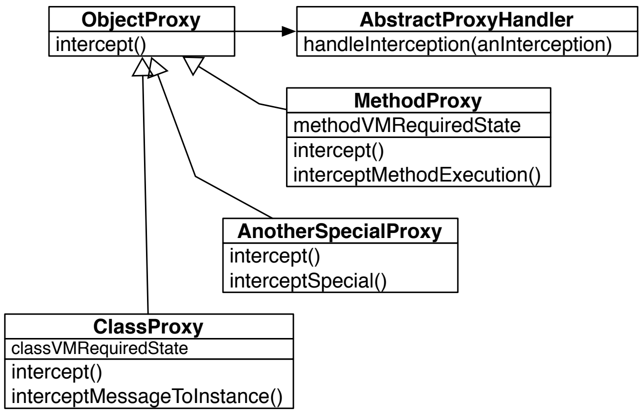


Figure 1. Proxies hierarchy in Ghost model.

ClassProxy. There are object-oriented programming languages that represent classes as first-class objects, *i.e.,* classes are not more than just instances from another class known as the Metaclass. ClassProxy provides proxies for class objects.

ClassProxy is needed as a special class in the model because the VM might impose specific constraints on the memory layout of object representing classes. For example, the Smalltalk VM expects the object to have three instance variables: format, methodDict, superclass. Since we are presenting Ghost *model*, that shape is generic. Different implementations may require different attributes or none. This is the reason why in Figure 1 the possible imposed memory layout for ClassProxy is represented by the attribute classVMRequiredState.

Frequently, the developer needs to be able to replace an existing class by a proxy. In that case, we need that the object replacement not only updates the references from other objects, but also the class pointer in the instances of the original class. For example, suppose there is an instance of User called bestUser. There is also a SecurityManager class that has a class variable called userClass which in this case points to User.

ClassProxy has to intercept the following type of messages:

- Messages that are sent directly to the class as a regular object. To continue with our example, imagine the method controlLogin in SecurityManager that sends the message maxLoggedUsers to its userClass instance variable. In Figure 1 this kind of interception is represented with the method intercept().
- Messages that are sent to an instance of the original class, *i.e.,* objects whose class references are pointing to the proxy (this happens as a consequence of replacing the class with the proxy). In our example, we can send the message username to the bestUser instance. In Figure 1 this kind of interception is represented with the method interceptMessageToInstance(). Notice that this kind of messages are only necessary when there is an object replace-

ment, *i.e.,* the instances' class pointers of the original class were updated to reference the proxy.

MethodProxy. In some dynamic languages, not only classes are first-class objects but also methods as well. In addition, similarly to the case of ClassProxy, there are two kinds of messages that MethodProxy needs to intercept:

- When sending messages to the method as a regular object. For example, in Smalltalk when you search for senders of a certain method, the system has to check in the literals of the compiled method if it is sending such message. To do this, the system searches all the literals of the compiled methods of all classes. This means it will send messages (sendsSelector: in this case) to the objects that are in the method dictionary. When creating a proxy for a method we need to intercept such messages. In Figure 1 this kind of interception is represented with the method intercept().
- When the compiled method is executed. Suppose we want to create a proxy for the method register of User class. We need to intercept the method execution, for example, when doing User new register. This kind of interception is represented in Figure 1 with the method interceptMethodExecution(). Note that this type of message exist *only* if there is object replacement, *i.e.,* when the original method is replaced by a proxy.

The same way that the VM imposes an object shape on classes, it may also do it on methods. This requirement is represented in Figure 1 with the instance variable methodVMRequiredState which may vary from one implementation to the other.

AnotherSpecialProxy. This class is just to document that the model must support different classes that need special management. In this paper, and in our implementation, we concentrate on classes and methods, but there can be more.

4.2 Handlers

Figure 2 shows the handler hierarchy of the Ghost model. Once again, note that this is an abstract model and a concrete implementation can vary significantly. Handler's responsibility is to handle the method interceptions that the proxies trap. It is not necessary to explain in details each handler, since we think it is self explanatory.

The information passed from a proxy to a handler can vary depending on the implementation. The typical passed information is:

- The name of the message received and its arguments.
- The proxy.
- The proxy's state. It can contain anything such as the target object, a filename or a number. This is necessary only if such state is in the proxy and not in the handler. Indeed, the proxy is supposed to intercept messages even if they are sent by the handler. So, the handler cannot send a message to the proxy to get its state. This is why it is the responsibility of the proxy to provide this state if any.

All that information is reified in the model as an instance of class Interception.

4.3 Discussions

Users can adapt and extend the Ghost framework according to their own needs via inheritance. In Figure 2 LoggerClassProxyHandler a user-defined class logs every intercepted messages and forwards them to the target object.

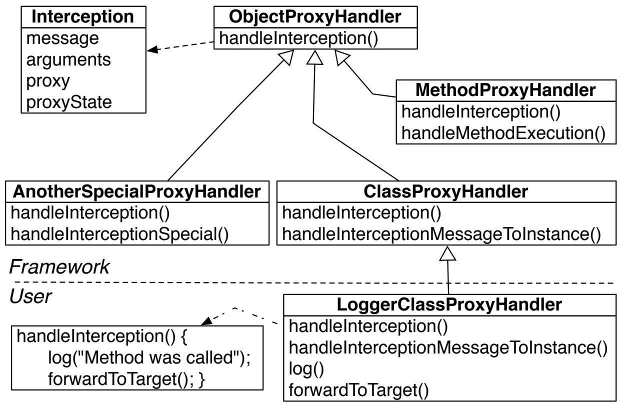


Figure 2. Handlers hierarchy in Ghost model.

Normally, some information is needed to accomplish the proxy process, for example, a target object, an address in secondary memory, a filename, an identifier, etc. This information can be stored in the proxies, in the handlers or elsewhere. However, as explained, if the state is kept in the proxy the handler cannot ask for it because such message sent will be intercepted. Hence, if the desire is to store the state in the proxy, such state must be included in the Interception object that is passed to the handler. This is represented as the instance variable proxyState in Figure 2. That instance variable can represent a target object, an address in secondary memory, a filename, an identifier, etc. Where to put this state is user's application dependent and a matter of design regarding the relationship between proxies and handlers.

Proxies delegates the interception to a handler. How the proxy gets the reference to the handler depends on the implementation. For example, in one case the handler can be an instance variable of the proxy that is provided when the proxy is created. In another case, all proxies can use the same handler, which in this case the previous instance variable may not be necessary and instead they reference directly to the handler class.

Notice that in the model we are modeling the interception of messages. However, some languages do not threat everything like a message sent, but instead they have special operators or syntax as part of the language. To implement Ghost, there must be a way to intercept such special syntax or otherwise pay the cost of not being able to intercept them.

5. Smalltalk Support for Proxies

Before presenting the Ghost implementation, we first explain the basis of the Pharo Smalltalk reflective model and some provided hooks. We show that Smalltalk provides all the necessary support for proxies *i.e.,* object replacement, interception of method execution and the reification of classes and methods as first-class objects.

5.1 Pharo Reflective Model and VM Overview

Readers familiar with the Pharo reflective model please feel free to skip this section. The reflective model of Smalltalk is easy and elegant. There are two important rules [4]: 1) *Everything is an object*; 2) *Every object is instance of a class*. Since classes are objects and every object is an instance of a class, it follows that classes must also be instances of classes. A class whose instances are classes is called a metaclass. Whenever you create a class, the system automatically creates a metaclass. The metaclass defines the structure and behavior of the class that is its instance. Figure 3 shows a *simplified* reflective model of Smalltalk.

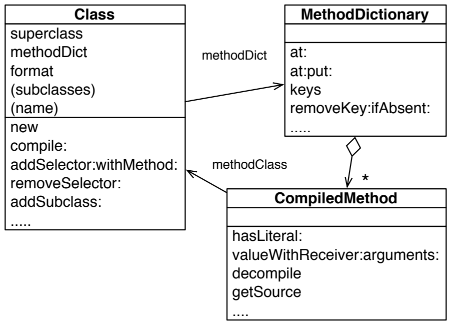


Figure 3. The basic Smalltalk reflective model.

Figure 3 shows that a class contains a name, a format, a method dictionary, its superclass, a list of instance variables, etc. The method dictionary is a map where keys are the methods names (called selectors in Smalltalk) and the values are the compiled methods which are instances of CompiledMethod.

5.2 Hooks and Features Provided by Pharo Smalltalk

Before explaining Ghost implementation on Pharo, we present some of the Smalltalk reflective facilities and hooks that can be used for implementing proxies.

Class with no method dictionary. The method dictionary is just an instance variable of a class, hence it can be changed. When an object receives a message and the VM does the method lookup, if the method dictionary of the receiver class (or of any other class in the hierarchy chain) is nil, then the VM directly sends the message cannotInterpret: aMessage to the receiver. But, the lookup for method cannotInterpret: starts in the *superclass* of the class whose method dictionary was nil.

Imagine the class MyClass which has its method dictionary in nil, and its superclass MyClassSuperclass. There is also an instance of MyClass called myInstance. Figure 4 shows how the hook works when sending the message printString to the object myInstance.

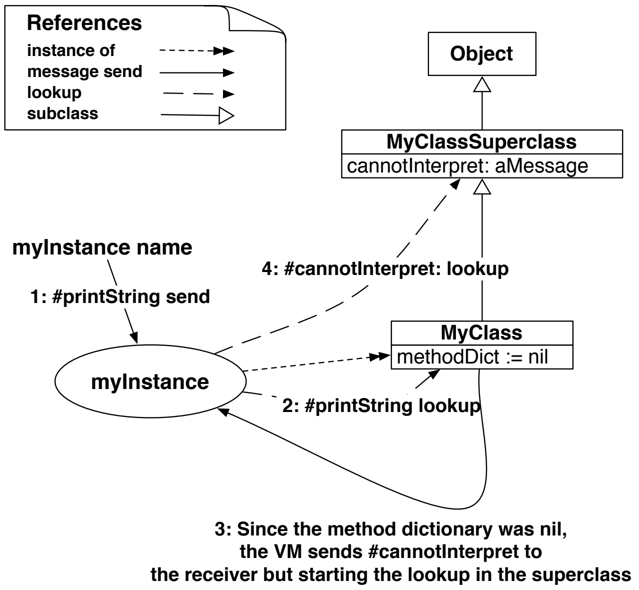

References
instance of
message send
lookup
subclass
Object
MyClassSuperclass
cannotInterpret: aMessage
myInstance name
1: #printString send
4: #cannotInterpret: lookup
MyClass
methodDict := nil
myInstance
2: #printString lookup
3: Since the method dictionary was nil,
the VM sends #cannotInterpret to
the receiver but starting the lookup in the superclass

Figure 4. Message handling when a method dictionary is nil.

The cannotInterpret: is sent to the receiver but starting the method lookup from the *superclass*. Otherwise there will be an infinite loop. This hook is very powerful for proxies since it let us intercept all messages that are sent to an object.

Objects as methods. This facility allows intercepting method executions. It relies on replacing in a method dictionary a method by an object that is not an instance of CompiledMethod. Interception occurs if the object does understand the message run:with:in: as we explain below. Otherwise, we get a MessageNotUnderstood exception.

To illustrate interception consider the following code:

```
MyClass methodDict at: #printString put: MethodProxy new.
MyClass new printString.
```

When the printString message is sent the VM does the method lookup and finds an entry for #printString in the method dictionary. If the retrieved object is actually an instance of CompiledMethod (which is the case in the normal scenario), then the VM executes it. Otherwise, the VM sends a special message run: aSelector with: arguments in: aReceiver to that object, *i.e.,* the one that replaces a method in the method dictionary.

This technique is used when implementing MethodWrappers [6]. Using run:with:in is not the only possible technique to implement MethodWrappers in Smalltalk. In fact, the original implementation rely on subclassing CompiledMethod.

It is important to notice that the previous explanation means that the Pharo VM does not impose any shape to objects acting as methods such us having certain amount of instance variables or certain format. This is because the

VM checks whether the object in the MethodDictionary is a CompiledMethod or not and if it is not it sends the message run:with:in:. The only requirement is to implement that method. Therefore, MethodProxy does not need to fulfill any class shape in a Ghost implementation on Pharo Smalltalk.

Object replacement. The primitive become: anotherObject is provided by the Pharo VM and it swaps the object references of the receiver and the argument. All variables in the entire system that used to point to the receiver now point to the argument, and vice versa. In addition, there is also becomeForward: anotherObject which updates all variables in the entire system that used to point to the receiver now point to the argument, *i.e.,* it is only one way.

Change the class of an object. Smalltalk provides a primitive to change the class of an object. Although it has some limitations, *e.g.,* the object format and the class layout of both classes need to be the same. These primitives are Object»primitiveChangeClassTo: or Behavior»adoptInstance: .

6. Ghost Implementation

In this section, we present the Ghost implementation. Its most important features are: to be stratified (*i.e.,* clear separation between proxies and handlers), to be able to intercept all messages, and to be uniform. For this implementation we use the previously mentioned Pharo Smalltalk reflective facilities: classes with no method dictionary, objects as methods, object replacement and the ability to change the class of an object.

Regarding the discussions of Section 4.3, in this implementation we store the needed information, for example, the target object, an identifier, a filename, etc, in the proxies. Another possible implementation is to store the information in the handler for example. In addition, in the following implementation each proxy instance uses a particular handler instance, hence the handler is represented as an instance variable of the proxy.

To explain the implementation we use a SimpleForwarderHandler which just forwards the interceptions to a target object. Therefore, the state stored in the proxy is a target object.

6.1 Kernel

Figure 5 shows the basic design of Ghost.

To explain the implementation we start with the following simple test:

```
testSimpleForwarder
    | proxy |
    proxy := Proxy proxyFor: (Point x: 3 y: 4) handler: SimpleForwarderHandler new.
    self assert: proxy x equals: 3.
    self assert: proxy y equals: 4.
```

The class side method proxyFor:handler: creates a new instance of Proxy, sets the handler, and finally changes the class of the just created Proxy instance to ProxyTrap. The user of the toolbox can specify which handler to use just by sending it as a parameter of the proxy creational message proxyFor:handler:.

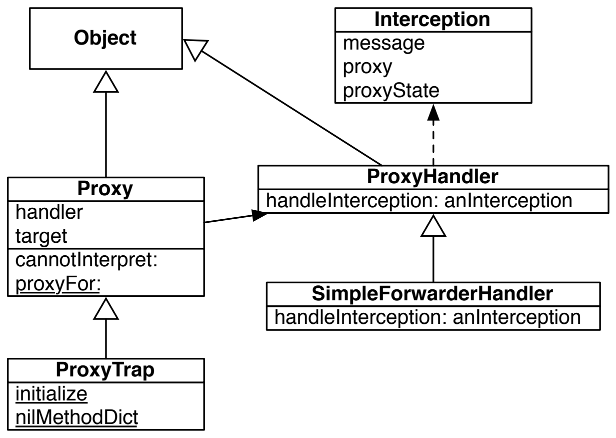


Figure 5. Ghost implementation's basic design.

```
Proxy class >> proxyFor: anObject handler: aHandler
    | aProxy |
    aProxy := self new
            initializeWith: anObject
            handler: aHandler.
    ProxyTrap adoptInstance: aProxy.
    ^ aProxy.
```

The class side method initialize is called right after loading ProxyTrap into the system and it sets the method dictionary of the class to nil. Notice that the system does not deal correctly with classes whose method dictionary is nil. Hence, we need to overwrite the method Behavior » methodDict to:

```
Behavior >> methodDict
  methodDict == nil ifTrue: [^ MethodDictionary new ].
  ^ methodDict
```

Since the system access the method dictionary with methodDict it looks like if the class has an empty method dictionary, but instead it has a nil. Since the VM access directly to the slow where the method dictionary is, *i.e.,* the VM does not use methodDict, it works for both things: the interception and the system.

With the line ProxyTrap adoptInstance: aProxy we change the class of aProxy to ProxyTrap, whose method dictionary is nil. This means that for *any* message sent to aProxy, the VM will finally send the message cannotInterpret: aMessage. Remember that such message is sent to the receiver (in this case aProxy) but starting the method lookup in the super class, which in this case is Proxy. Hence, Proxy implements the method cannotInterpret:

```
Proxy >> cannotInterpret: aMessage
    | interception |
      interception := Interception for: aMessage proxyState: target proxy: self.
    ^ handler handleInterception: interception.
```

An Interception instance is created and passed to the handler. In this example, the instance variable proxyState is the target object.

Handler classes are user-defined and in this example we use a simple forwarder handler, *i.e.,* it logs and forwards the received message to a target object. Users of the toolbox can create their own handlers that achieve their requirements.

```
SimpleForwarderHandler >> handleInterception: anInterception
    | answer |
     self log: 'Message ', anInterception message selector, ' intercepted'.
        answer := anInterception message sendTo: anInterception proxyState.
    self log: 'The message was forwarded to the target object'.
    ^ answer
```

For the moment, we can say that the class Proxy can only be used for regular objects (in the example we create a proxy for Point instance). We see in the following sections how Ghost handles objects that do require special management like classes or methods.

6.2 Proxies for Methods

As we have already explained in Section 4, for methods there are two kind of messages that we need to intercept:

- When the compiled method is executed.
- When sending messages to the compiled method object.

To clarify, imagine the following test:

```
testSimpleProxyForMethods
    | aProxy kurt method |
    kurt := User named: 'Kurt'.
    method := User compiledMethodAt: #username.
    aProxy := Proxy
        createProxyAndReplace: method
        handler: SimpleForwarderHandler new.
    self assert: aProxy getSource equals: 'username ^ name'.
    self assert: kurt username equals: 'Kurt'.
```

What the test does is to create an instance of a User and a proxy for method username. Then, we replace the original method username with the created proxy. Finally, we test both type of messages: when sending a message to the proxy (in this case aProxy getSource) and when sending message username that leads to the execution of the proxified method.

With Ghost implementation, both kind of messages are solved out of the box: the first case, *i.e.,*aProxy getSource has nothing special and it behaves exactly the same way we have explained so far. The second one, *i.e.,*kurt username, also works without any special management by using the explained hook of the method run:with:in:. However, this second type of message is only captured if the original method was replaced by the proxy. This is why in this test we use the method createProxyAndReplace:handler: instead of proxyFor:handler:, because we want to not only to create a proxy for the method but instead replace it with the proxy. The following is the implementation of such method:

```
Proxy class >> createProxyAndReplace: aClass handler: aHandler
    | aProxy newProxyRef newObjectRef|
    aProxy := self new
      initializeWith: anObject
      handler: aHandler.
    aProxy become: anObject.
     "After the become is done, variable aProxy points to anObject
     and variable anObject points to aProxy. We create two new
     variables just to clarify the code"
    newProxyRef := anObject.
    newObjectRef := aProxy.
    newProxyRef target: newObjectRef.
    ProxyTrap adoptInstance: newProxyRef.
    ^ newProxyRef.
```

Notice that createProxyAndReplace:handler: is useful for method proxies, as well as for regular objects. In the previous section where we used the method proxyOn: we could perfectly have used createProxyAndReplace:handler: instead.

Coming back to the test of kurt username, when the VM does the method lookup for the message username it notices that in the method dictionary is not a CompiledMethod instance but instead an instance from another class. Hence, it sends the message run:with:in to such object. Since such object is a proxy in this case, the message run:with:in: will be intercepted and forwarded just like any other message. In the base Pharo image, CompiledMethod does not implement such method, so Ghost implements it as a method extension in the following way:

```
CompiledMethod >> run: aSelector with: anArray in: aReceiver
     ^ self valueWithReceiver: aReceiver  arguments: anArray
```

That method just executes the method (the receiver). However, such change does not need to be necessary implemented in CompiledMethod. As we will see later, Ghost supports a way to define specific messages so that they are treated and answered by the handler instead of being managed as a normal interception. So we can tell the handler to perform something in particular if the message run:with:in: is intercepted (this information is available in the Message instance referenced by the Interception object). In this case we can directly use the method valueWithReceiver:arguments: to execute the CompiledMethod.

The previous explanation demonstrates how Ghost can create not only proxies for methods, but also how to replace them by proxies. In contrast to what we defined in the model, the Pharo Smalltalk VM does not impose any shape to methods. Therefore, we can use the same Proxy class that we use for regular objects, *i.e.,* the class MethodProxy defined in the Ghost model does not exist in this concrete implementation since we can directly use Proxy.

6.3 Proxies for Classes

Implementing proxies for classes and also to be able to replace and use a proxy as a class, has some important constraints:

- Class proxies must fulfill the expected object shape that the VM imposes in classes. In the case of Pharo Smalltalk, the minimum amount of instance variables that a class must have are: superclass, methodDict and format.
- Instances hold a reference to their class and the VM uses this reference for the method lookup.
- A class is involved with two kinds of messages that need to be intercepted as introduced in Section 4:
  - Messages that are sent directly to the class.
  - Messages that are sent to an instance of the class. Such messages are intercepted only if the original class was replaced by the proxy.

To explain class proxies, consider the following test:

```
testSimpleProxyForClasses
    | aProxy kurt |
    kurt := User named: 'Kurt'.
    aProxy := ClassProxy
        createProxyAndReplace: User
        handler: SimpleForwarderHandler new.
    self assert: User name equals: #User.
    self assert: kurt username equals: 'Kurt'.
```

The test creates an instance of a user, and then with the message createProxyAndReplace:handler: we create a proxy for the User class and we replace it by the created proxy. Finally, we test that we can intercept both messages: those which are sent to the proxy (in this case User name) and those which are sent to instances of the original class (kurt username in this case).

The first message, User name, has nothing special and it is handled the same way as any other message. The second one is more complicated and it requires certain explanation.

Figure 6 shows the design of ClassProxy. First, notice that we do not use the class Proxy but instead ClassProxy. This is because proxies for classes need to fulfill the expected object shape that the VM imposes in classes, *i.e.,* the instance variables superclass, methodDict and format. Second, in the model we showed that ClassProxy was a subclass of ObjectProxy but in this case it is not. The reason is that the VM does not only imposes the mentioned instance variables but also the *order*: superclass at position 1, methodDict at 2 and format at 3. If ClassProxy is a subclass of ObjectProxy it inherits the two instance variables target and handler and since they are defined in the superclass they are "first" in the array of instance variables of the object. So, superclass will be at position 3, methodDict at 4 and format at 5. Therefore, we are not respecting the expected shape.

The previous issue is not really a problem because ObjectProxy implements only two methods and they are even different in ClassProxy. Hence, even if the limitation is real, we are not duplicating code because of that.

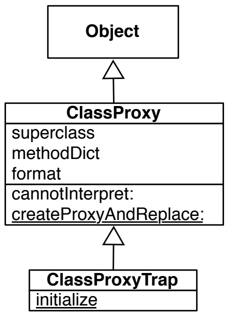


Figure 6. Class proxies in Ghost stratified implementation.

The method createProxyAndReplace:handler: is similar to the one used in Proxy:

```
ClassProxy class >> createProxyAndReplace: aClass handler: aHandler
  | aProxy newProxyRef newClassRef|
  aProxy := self new
       initializeWith: aHandler
         methodDict: nil
         superclass: ClassProxy
         format: aClass format.
  aProxy become: aClass.
   "After the become is done, aProxy now points to aClass
    and aClass points to aProxy. We create two new variables
    just to clarify the code"
  newProxyRef := aClass.
  newClassRef := aProxy.
  newProxyRef target: newClassRef.
  ClassProxyTrap adoptInstance: newProxyRef.
  ^ newProxyRef.
```

The difference is that in addition to setting the handler and the target, we also set the method dictionary, the superclass and the format. This is because an *instance* of ClassProxy must work as a class. Thus, we set its method dictionary in nil, ClassProxy as the superclass and finally the format (this is important so that the adoptInstance: does not fail).

Coming back to the example, when we evaluate kurt username this is what happens: the class reference of kurt is pointing to the created ClassProxy instance (as a result of the become:), and this proxy object that acts as a class, has the method dictionary instance variable in nil. Hence, the VM sends the message cannotInterpret: to the receiver (kurt in this case) but starting the method lookup in the superclass which is ClassProxy (as set in method ClassProxy class » createProxyAndReplace:handler: defined above). The definition of the cannotInterpret: of class ClassProxy is the following.

```
ClassProxy >> cannotInterpret: aMessage
   | interception |
   "The order of this expression is important
   because a proxy intercepts all messages including =="
   (ClassProxyTrap == aMessage lookupClass )
```

```
    ifTrue: [ interception := Interception for: aMessage
                          proxyState: target proxy: self.
        ^ handler handleInterception: interception]
    ifFalse: [ interception := Interception for: aMessage
                      proxyState: target proxy: aMessage lookupClass.
        ^ handler handleInterception: interception toInstance: self]
```

It is important to notice the difference in this method regarding the kind of message it is intercepting. On the one hand, when we evaluate User name and the cannotInterpret: is called, the receiver, *i.e.,* what self is pointing to, is the proxy itself. On the other hand, when we evaluate kurt userame and cannotInterpret: is called, self points to kurt and not to the proxy.

The method Message lookupClass answers the class where lookup will start. If it is ClassProxyTrap it means the receiver was proxy, and not an instance of the original class.

A problem is that the CompiledMethod of cannotInterpret: cannot be correctly executed with a receiver like kurt. In fact, it can only be correctly executed with proxy instances. The reason is that the method ClassProxy » cannotInterpret: access the instance variable handler. Hence the first problem is that the class User does not define such instance variable. The second problem is that CompiledMethod do not store instance variable names but instead its offsets. So when the CompiledMethod of cannotInterpret: is executed the instructions (bytecodes) to access the instance variable handler is just something like "access instance variable at position 5", which is correct in the class where it was defined (ClassProxy). When evaluating the method with receivers of other classes *e.g.,*User then the VM can crash because it is accessing outside the object or just answer whatever is at that place. For example, if a class defines only two instance variables, the bytecode "accessing instance variable at position 5" means that the VM will access a memory address outside the object. Whether the VM crashes or not depends on the concrete VM implementation. In the case of Pharo Smalltalk, the VM crash in such scenario so we cannot use this solution.

Instead of directly accessing the instance variable handler one may think why not to send a message handler. This is not possible because since the proxy intercepts all messages, such message sent will finally call cannotInterpret: generating an infinite loop.

To that limitation, Ghost provides the following alternative. Instead of doing handler handleInterception: interception toInstance: self we send a special message to the proxy, which is accessible through aMessage lookupClass. Hence, we can do aMessage lookupClass handleInterception: interception toInstance: self. In the item *Ease of debugging* of the next section we explain that we can define a list of specific messages in the handler so that it does not manage such messages interceptions as it is done with the regular ones, but instead those messages are processed and answered by the handler itself. The message handleInterception:toInstance: is one of those messages and it is managed by the handler. At that point the handler has everything he needs *e.g.,*Interception object and receiver, so it can perform its task.

Coming back to the implementation, the last missing explanation is why we need ClassProxyTrap instead of reusing ProxyTrap. The reason is that the message adoptInstance: requires certain conditions, like having the same object format. Since ClassProxy and Proxy have different amount of instance variables and hence format, then we cannot reuse the same ProxyTrap.

```
ProxyTrap class >> initialize
  superclass := Proxy.
  methodDict := nil.
  format := Proxy format.

ClassProxyTrap class >> initialize
  superclass := ClassProxy.
  methodDict := nil.
  format := ClassProxy format.
```

The Ghost implementation uses ProxyClass and ClassProxyTrap not only because it is cleaner from the design point of view but also because of the memory footprint. Technically, we can use ProxyClass and ClassProxyTrap *also* for regular objects and methods. But that implies that for every target to proxify the size of the proxy can be unnecessary bigger in memory footprint, because of the additional instance variables needed by ClassProxy.

To conclude, with this implementation we can successfully create proxies for classes, *i.e.,* to be able to intercept the two mentioned kind of messages and replace classes by proxies.

6.4 Criteria Evaluation

Stratification. This solution is completely stratified. On the one hand, there is a clear separation between proxies and handlers. On the other hand, interception facilities are separated from application functionality. Indeed, the application can even send the cannotInterpret: message to the proxy. Since, proxies do not understand any message, cannotInterpert: would be intercepted like any other message. Thus, the proxy API does not pollute the application's namespace.

Object replacement. This is provided by Ghost thanks to the Smalltalk become: primitive.

Interception granularity. It intercepts all messages.

Transparency. The Pharo compiler associates special bytecodes for the messages class and == (identity), *i.e.,* even if there is an implementation of those methods, they are actually never executed and, therefore, they cannot be intercepted. Our solution is to modify the compiler so that it does not associate a special bytecode for both methods. Such modification is the following:

```
(ParseNode classVarNamed: 'StdSelectors') removeKey: #class.
(ParseNode classVarNamed: 'StdSelectors') removeKey: #==.
Compiler recompileAll.
```

We did a benchmark to estimate the overhead impact of such change. We run all the tests (8003 unit tests) present in a PharoCore 1.3 - 13204 image, twice: once with the class and == optimizations and once without them. The overhead of removing those optimizations was only about 4%, which means that it is only slightly perceptible in general system interactions.

In the discussion of Section 4 we talk about the possibility of some languages to have special syntax or operators in addition to messages sent. These special selectors class and == can be considered like that. However, Smalltalk allows us to convert them into messages so we have an easy way to deal with them. This way Ghost solution is fully transparent and both messages are intercepted and handled as any other message.

Efficiency. From the CPU point of view, this solution is fast and it has low overhead.

This solution provides an efficient memory usage with the following optimizations:

- Proxy and ClassProxy are "Compact Classes". This means that in a 32 bits system, their instances' object header are only 4 bytes long instead of 8 bytes for instances of regular classes. For instances whose "body" part is more than 255 bytes and whose class is compact, their header will be 8 bytes instead of 12. The first word in the header of regular objects contains flags for the Gargbage Collector, the header type, format, hash, etc. The second word is used to store a reference to the class. In compact classes, the reference to the class is encoded in 5 free bits in the first word of the header. These 5 bits represent the index of a class in the compact classes array set by the image[3] and accessible to the VM. With these 5 bits, there are 32 possible compact classes. This means that, from the language side, the developer can determinate up to 32 classes as compact. Their instances' object header are only 4 bytes long as we said. Hence, declaring the proxy classes as compact makes proxies to have smaller header and then smaller memory footprint.
- Proxies only keep the minimal state they need. For example, as we have already explained, we can use ClassProxy for every type of object. However, the size of the proxies would be unnecessary larger to store the additional needed instance variables of ClassProxy.
- In proxy creation methods presented so far (proxyFor:handler: and createProxyAndReplace:handler:) the last parameter is an instance of the handler. This is because in our examples, each proxy holds a reference to handler. However, this is only necessary when the user needs one handler instance per target object, which is not often the case. The handler is often stateless and can be shared and referenced through a class variable or a global one. Hence, we can avoid the memory cost of a handler instance variable in the proxy. Instead, one possible solution is to reference in the Proxy»cannotInterpret: method a handler class which has a class side method handleInterception:. For example:

```
Proxy >> cannotInterpret: aMessage
    | interception |
     interception := Interception for: aMessage proxyState: target proxy: self.
    ^ SimpleForwarderHandler handleInterception: interception.
```

An alternative is to use a handler class with a singleton or a default instance. For example:

```
Proxy >> cannotInterpret: aMessage
    | interception |
     interception := Interception for: aMessage proxyState: target proxy: self.
    ^ SimpleForwarderHandler uniqueInstance handleInterception: interception.
```

In both cases we save the memory corresponding to the instance variable to reference the handler plus the handler instance itself. If we consider that the handler has no instance variable, then it is 4 bytes for the instance variable in the proxy and 8 bytes for the handler instance. That gives a total of 12 bytes saved per proxy in a 32 bits system.

Implementation complexity. This solution is easy to implement: an approximation of 5 classes, with an average of 3.4 methods per class, and each method is of an average of 5 lines of code.

Ease of debugging. Ghost implementation supports special messages that the handler must answer itself instead of managing it as a regular interception. The handler can keep a dictionary that maps selector of messages intercepted by the proxy to selectors of messages to be performed by the handler itself. This user-defined list of selectors can be used for debugging purposes, *i.e.,* those messages that are sent by the debugger to the proxy are answered by the handler and they are not managed as a regular interception. This significantly ease the debugging of proxies. For example, the handler's dictionary of special messages for debugging can be defined as following:

```
SimpleForwarderHandler >> debuggingMessagesToHandle
| dict |
dict := Dictionary new.
dict at: #basicInspect put:#handleBasicInspect:.
dict at: #inspect put:#handleInspect:.
dict at: #inspectorClass put:#handleInspectorClass:.
dict at: #printStringLimitedTo: put: #handlePrintStringLimitedTo:.
dict at: #printString put: #handlePrintString:.
^ dict
```

The keys of the dictionary are selectors of messages received by the proxy and the values are selectors of messages that the handler must send to itself. All the selectors of messages to be sent to the handler (*i.e.,* the dictionary values)

[3] see methods SmalltalkImage»compactClassesArray and SmalltalkImage»recreateSpecialObjectsArray

have a parameter which is an instance of Interception, which contains the receiver, the message, the proxy and the target. Therefore, all those methods have access to all the information they need.

Moreover, these special messages are "pluggable" *i.e.,* they can be easily enabled *e.g.,* for debugging, and disabled for production.

Constraints. The solution is flexible since target objects can inherit from any class and they are free to implement or not implement all the methods they want. There is not any kind of restriction imposed by Ghost. In addition, the user can easily extent or change the purpose of the proxy adapting it to his own needs: he just needs to subclass a handler and implement the necessary methods like handleInterception:.

Uniformity. This implementation is uniform since proxies can be used for regular objects, as classes and as methods. Moreover they all provide the same API and can be used polymorphically. Nevertheless, there is still non-uniformity regarding some other special classes and objects. Most of them are those that are present in what is called the *special objects array* (check method recreateSpecialObjectsArray) in Pharo Smalltalk. Such array contain the list of special objects that are known by the VM. Examples are the objects nil, true, false, etc. It is not possible to do a correct object replacement of those objects by proxies. The same happens with immediate objects, *i.e.,* objects that do not have object header and are directly encoded in the memory address, like SmallInteger.

The special object array contains not only regular objects but also classes. Those classes are known and used by the VM so it may impose certain shape, format or responsibilities in their instances. For example, one of those classes in Process. Once again, it is not possible to correctly replace a Process instance by a proxy. The same limitation exists if we want to create a proxy not for instances of those special classes but for those classes.

The mentioned limitations occur only when object replacement is desired. Otherwise, there is no problem and proxies can be created for those objects. In addition, we believe that creating proxies for methods and classes is useful in several scenarios as we see in next section. The rest of the mentioned limitations is not a common need. Hence, those restrictions are not a real problem for Ghost users.

Portability. This is the bigger disadvantage of this approach. It requires the hook of setting nil to a method dictionary and the VM sending the message cannotInterpret:. In addition, it also requires object replacement (become: primitive) and to be able to change the class of an object (adoptInstance: primitive). However, without these reflective facilities we cannot easily implement all the required features of a good proxy library. In the best case, we can get everything but with substantial development effort such as modifying the VM or compiler, or even creating them from scratch. Smalltalk provides all those features by default.

7. Related Work

7.1 Proxies in dynamic languages

Objective-C provides an out-of-the-box Proxy implementation called NSProxy [21]. This solution consists of an abstract class NSProxy that implements the minimum number of methods to be a root class. Indeed, this class is not a subclass of NSObject (the Objective-C root class in the hierarchy chain) but a separate root class (like subclassing from nil in Smalltalk). The intention is to reduce method conflicts between the proxified object and the proxy. Subclasses of NSProxy can be used to implement distributed messaging, future objects or other proxies usage. Typically, a message to a proxy is forwarded to a profixied object which can be an instance variable in a NSProxy subclass.

Since Objective-C is a dynamic language, it needs to provide a mechanism like the Smalltalk doesNotUnderstand: for the cases where an object receives a message that cannot understand. When a message is not understood, the Objective-C runtime will send methodSignatureForSelector: to see what kind of argument and return types are present. If a method signature is returned, the runtime creates a NSInvocation object describing the message being sent and then sends forwardInvocation: to the object. If no method signature is found, the runtime sends doesNotRecognizeSelector:.

NSProxy subclasses must override the forwardInvocation: and methodSignatureForSelector: methods to handle messages that they do not implement themselves. A subclass's implementation of forwardInvocation: should do whatever is needed to process the invocation such as forwarding the invocation over the network or loading the real object and passing the invocation. methodSignatureForSelector: is required to provide argument type information for a given message. A subclass' implementation should be able to determine the argument types (note that ObjectiveC is not so dynamic from this regard) for the messages it needs to forward and should construct a NSMethodSignature object accordingly.

To sum up, the developer needs to subclass NSProxy and implement the forwardInvocation: to handle messages that are not understood by itself.

One of the drawbacks of this solution is that the developer does not have control over the methods that are implemented in NSProxy. For example, such class implements the methods isEqual:, hash, class, etc. This is a problem because those messages will be understood by the proxy instead of being forwarded to the wrapped object producing different paths in the code execution. This solution is similar to the common solution in Smalltalk with doesNotUnderstand:. A possible, yet tedious, solution may be to overwrite such methods in the NSProxy subclass so that they delegate to the wrapped object.

In Ruby, there is a proxy implementation which is called Delegator. This is just a class included with Ruby standard library but it can be easily modified or implemented from scratch. Similar to Objective-C and Smalltalk (indeed, similar to most dynamic languages), Ruby provides a mechanism to handle the situation when an object receives a message that cannot understand. This method is called method_missing(aSelector, *args). Moreover, since Ruby 1.9 Object is not the root of the hierarchy chain and Object is a subclass of a new minimal class called BasicObject which understands a few methods and is similar to ProtoObject in Smalltalk.

The idea of Ruby proxies are similar to the Smalltalk solution using doesNotUnderstand: and to NSProxy: have a minimal object (subclass from BasicObject) and implement method_missing(aSelector, *args) to intercept messages. In Python, an analogous implementation can be done by overwriting the __getattr__ method in a proxy. Such method is called when an attribute lookup has not found the attribute in the usual places.

Arnaud et al. [1] took a much deeper approach: internally, an object X does not refer directly to another object Y, but instead X has a reference to a special Handler object that refers to Y. The handler object is fully invisible for the developer. The idea is that different references to an object can use different handlers. This can be used for several things, like defining read-only references to an object. But the solution is generic so for example a handler could be used as a proxy. For example, a simple handler could be implemented so that it does something in particular with the message interception *e.g.,* logging, and then forward it to the target object.

7.2 Proxies in static languages

Java, being a statically typed language, supports quite limited proxies called *Dynamic Proxy Classes* [14]. It relies on the Proxy class from the java.lang.reflect package. "Proxy provides static methods for creating dynamic proxy classes and instances, and it is also the superclass of all dynamic proxy classes created by those methods."[14]. The creation of a dynamic proxy class can only be done by providing a list of java interfaces that should be implemented by the generated class. All messages corresponding to declarations in the provided interfaces will be intercepted by a proxy instance of the generated class and forwarded to a handler object. "Each proxy instance has an associated invocation handler object, which implements the interface InvocationHandler. A method invocation on a proxy instance through one of its proxy interfaces will be dispatched to the invoke method of the instance's invocation handler, passing the proxy instance, a java.lang.reflect.Method object identifying the method that was invoked, and an array of type Object containing the arguments. The invocation handler processes the encoded method invocation as appropriate and the result that it returns will be returned as the result of the method invocation on the proxy instance. " [14].

Java proxies have the following limitations:

- You *cannot* create a proxy for instances of a class which methods aren't all declared in interfaces. This means that, if you want to create a proxy for a domain class, you are forced to create an interface for it. Eugster [11] proposed a solution which provides proxies for classes. There is also a third-party framework based on bytecode manipulation called CGLib [9] which provides proxies for classes.
- *Only* the methods defined in the interface will be intercepted which is a big limitation.
- Java interfaces do not support private methods. Hence since Java proxies require interfaces, private methods cannot be intercepted either. Depending of the proxy usage this can be a problem.
- Proxies are subclass from Object, forcing them to understand several messages. When the messages hashCode, equals or toString (declared in Object) are sent to a proxy instance they are encoded and dispatched to the invocation handler's invoke method, *i.e.,* they are intercepted. However, the same does not happen with the rest of the public methods, *e.g.,*getClass. So a proxy answers its own class instead of the target's one. Therefore, the proxy is not transparent and it is not fully stratified.

Microsoft's .NET platform [26] proposes a closely related concept of Java dynamic proxies with nearly the same limitations as in Java. There are others third-party libraries like *Castle DynamicProxy* [8] or *LinFu* [18]. DynamicProxy differs from the proxy implementation built into .NET which requires the proxified class to extend MarshalByRefObject. Extending MashalByRefObject to proxy an object can be too intrusive because it does not allow the class to extend another class and it does not allow transparent proxying of classes. In LinFu, every generated proxy, dynamically overrides all of its parent's virtual methods. Each one of its respective overridden method implementations delegates each method call to the attached interceptor object. However, non of them can intercept non-virtual methods.

7.3 Comparison

Statically typed languages, such as Java or .NET, support quite limited proxies [2]. In Java the problem is that types are bound to classes and in addition the lookup is done statically *i.e.,* at compile-time. There is also the replacement issue and transparency. Another problem in Java is that one cannot build a proxy with fields storing any specific data. Therefore, one has to put everything in the handler, hence no handler sharing is possible ending in a bigger memory footprint.

Proxies are far more powerful, flexible, transparent and easy to implement in dynamic languages than static ones.

In dynamic languages, just two features are enough to implement a naive Proxy solution: 1) a mechanism to handle messages that are not understood by the receiver object and 2) a minimal object that understands a few or no messages so that the rest are managed by the mentioned mechanism.

Objective-C NSProxy, Ruby Decorator, etc, all work that way. Nevertheless, non of them solves all the problems mentioned in this paper:

Memory footprint. None of the solutions take special care of the memory usage of proxies. This is a real limitation when proxies are being used, *e.g.,* to save memory.

Object replacement. Most proxy solutions can create a proxy for a particular object X. The user can then use that proxy as the original object. The problem is that there may be other objects in the system referencing to X. Without object replacement, those references will still be pointing to X instead of pointing to the proxy. Depending on the proxies usage, this can be a drawback.

Proxies for classes and methods All the investigated solutions create proxies for specific objects but none of them are able to create proxies for class objects or compiled methods.

8. Conclusion

In this paper, we described the Proxy pattern, its different usages and common problems while trying to implement them. We introduced Ghost, a generic, light-weight and stratified Proxy model and its implementation on top of Pharo Smalltalk.

Our solution provides uniform proxies not only for regular instances, but also for classes and methods. In addition, Ghost proxies can have a really small memory footprint. Proxies are powerful, easy to use and extend and its overhead is low.

Ghost was easy to implement on Pharo Smalltalk because the language and the VM provide unique reflective facilities and hooks. Nevertheless, we believe that such specific features, provided by Smalltalk and its VM, can also be ported to other dynamic programming language.

Acknowledgements

This work is supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the CPER 2007-2013.

References

[1] J.-B. Arnaud, M. Denker, S. Ducasse, D. Pollet, A. Bergel, and M. Suen. Read-only execution for dynamic languages. In *TOOLS-Europe'10*, June 2010.

[2] T. Barrett. Dynamic proxies in Java and .NET. *Dr. Dobb's Journal of Software Tools*, 28(7):18, 20, 22, 24, 26, July 2003.

[3] J. K. Bennett. The design and implementation of distributed Smalltalk. In *Proceedings OOPSLA '87*, volume 22, pp 318–330, Dec. 1987.

[4] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[5] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *OOPSLA'04*, pp 331–344, 2004. ACM Press.

[6] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to the rescue. In *ECOOP'98*, LNCS 1445, pp 396–417. Springer-Verlag, 1998.

[7] P. Butterworth, A. Otis, and J. Stein. The GemStone object database management system. *Commun. ACM*, 34(10):64–77, 1991.

[8] Castle dynamicproxy library. http://www.castleproject.org/dynamicproxy/index.html.

[9] cglib code generation library. http://cglib.sourceforge.net.

[10] S. Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

[11] P. Eugster. Uniform proxies for java. In *OOPSLA'06*, pp 139–152, 2006.

[12] E. Gamma, R. Helm, J. Vlissides, and R. E. Johnson. Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings ECOOP '93*, LNCS 707, pp 406–431, 1993.

[13] Y. Hassoun, R. Johnson, and S. Counsell. Applications of dynamic proxies in distributed environments. *Software Practice and Experience*, 35(1):75–99, Jan. 2005.

[14] Oracle. java dynamic proxies. the java platform 1.5 api specification. hhttp://download.oracle.com/javase/1.5.0/docs/api/java/lang/reflect/Proxy.html.

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

[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings ECOOP '97*, LNCS 1241, pp 220–242, 1997.

[17] R. Koster and T. Kramp. Loadable smart proxies and native-code shipping for CORBA. In *USM*, LNCS 1890 , pp 202–213, 2000.

[18] Linfu proxies framework. http://www.codeproject.com/KB/cs/LinFuPart1.aspx.

[19] P. Lipton. Java proxies for database objects. http://www.drdobbs.com/windows/184410934, 1999.

[20] P. L. McCullough. Transparent forwarding: First steps. In *Proceedings OOPSLA '87*, volume 22, pp 331–341, Dec. 1987.

[21] Apple. developer library documentation. http://developer.apple.com/library/ios/#documentation/cocoa/reference/foundation/Classes/NSProxy_Class/Reference/Reference.html.

[22] G. A. Pascoe. Encapsulators: A new software paradigm in Smalltalk-80. In *Proceedings OOPSLA '86* , volume 21, pp 341–346, Nov. 1986.

[23] P. Pratikakis, J. Spacco, and M. Hicks. Transparent proxies for java futures. In *OOPSLA '04*, pp 206–223, 2004.

[24] N. Santos, P. Marques, and L. Silva. A framework for smart proxies and interceptors in RMI, 2002.

[25] M. Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *ICDCS'86*, pp 198–205, 1986. IEEE Computer Society.

[26] T. Thai and H. Q. Lam. .NET framework essentials / T. thai, H.Q. lam., 2001.

[27] T. Van Cutsem and M. S. Miller. Proxies: design principles for robust object-oriented intercession apis. *Dynamic Language Symposium*, 45:59–72, 2010.

[28] N. Wang, K. Parameswaran, D. Schmidt, and O. Othman. The design and performance of Meta-Programming mechanisms for object request broker middleware. In *COOTS'01 (USENIX)*, pp 103–118, 2001.

[29] I. Welch and R. Stroud. Dalang - A reflective extension for java, OOPSLA99 Workshop on Reflection, 1999.

Challenges to support automated random testing for dynamically typed languages

Stéphane Ducasse
RMoD – INRIA Lille Nord Europe, France
http://stephane.ducasse.free.fr/

Manuel Oriol
University of York, UK
manuel@cs.york.ac.uk

Alexandre Bergel
Pleiad Lab, Department of Computer Science (DCC), University of Chile, Chile
http://bergel.eu

Abstract

Automated random testing is a proven way to identify bugs and precondition violations, and this even in well tested libraries. In the context of statically typed languages, current automated random testing tools heavily take advantage of static method declaration (argument types, thrown exceptions) to constrain input domains while testing and to identify errors. For such reason, automated random testing has not been investigated in the context of dynamically typed languages. In this paper we present the key challenges that have to be addressed to support automated testing in dynamic languages.

1. Introduction

Random testing is a form of automatic testing that randomly generates data input and test cases. Random testing shines in its effectiveness to identify software faults against manual testing [CPO+11]. Random tests are appealing because they are relatively easy and cheap to obtain.

Autotest [CPL+08, COMP08] and Yeti [OT10][1] have proven that automated random testing is an effective way to identify bugs and generate test cases which reproduce them. Autotest typically finds more bugs than any kind of manual testing in very small amounts of time [COMP08]. While all types of testing find different kinds of bugs, there is an overlap between bugs found by random testing and bugs found by other techniques.

Dynamically typed languages, including Pharo, [BDN+09] should be able to take advantages of the benefits of automated random testing. Unfortunately, current automated random testing tools heavily rely on of static type annotation: method signatures, including argument types, return types and thrown exceptions, constrain input domains used by a random test. In addition, static types qualify the software faults found by random tests (*e.g.,* whether a fault is effectively a bug or a false positive). In this paper we present the key challenges and paths that further researchers may follow to support automated testing in dynamic languages.

First we present the principles of random automated testing in Java (Section 2) by explaining the strategies of YETI, one of the best automated random testing tools. We subsequently identify the challenges that dynamic languages pose and we sketch some possible tracks of solutions (Section 3). The focus of this article is not to propose a solution but to stress the challenges that have to be addressed to support automated random testing for dynamically typed languages.

[1] http://www.yetitest.org

[Preprint of International Workshop on Smalltalk Technologies 2011.]

2. Random Automated Testing in Java: The Facts

2.1 Random Automated Testing Principles

To explain how random testing tools for statically typed languages work, we present how the York Extensible Testing Infrastructure (YETI) – a language agnostic automated random testing tool – works.

YETI is an application coded in Java. It tests programs at random in a fully automated manner. It is designed to support various programming languages – for example, functional, procedural and object-oriented languages can easily be bound to YETI. YETI is a lightweight platform with around 10,000 lines of code for the core, the strategies and the Java binding and is available under BSD license.

Figure 1 shows the general process of an automated random testing tool. An instance database is created with some seeds (for example, 0, 1, -1 for numbers). Such database is used during the instance generation period. The instance generation step requires instances for both the receiver and arguments of a message. It uses class type profiles (types method declaration) which are stored in another database.

Then tests are created by calling methods, the tests may check the returns values and the thrown exceptions. During tests execution, when a new instance is created it may be added to the instance database. To determine whether a fault is actually found, the declared exceptions and precondition (argument types are used) are key to determine whether a fault is a bug in the analyzed software or whether it is a false positive.

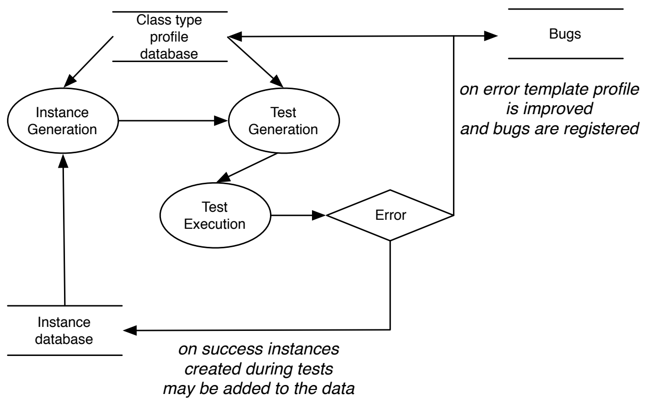

Class type profile database
Bugs
Instance Generation
Test Generation
on error template profile is improved and bugs are registered
Test Execution
Error
Instance database
on success instances created during tests may be added to the data

Figure 1. Automated random testing process

YETI contains three parts: the core infrastructure, the strategies, and the language-specific bindings. The core infrastructure contains the code to represent routines (methods or functions), a graph of types (class profile database), and a pool of typed objects (instance database). Routines use arguments of certain types and return an object of a certain type (if any) for which they are considered constructors.

```
// Input: Program/Strategy
// Output: found bugs
foundBugs = new Vector<Bug>();
M0 = strategy.getModuleToTest();
while (not endReached){
  R0=strategy.getRandomRoutineFromModule(M0);
  Vector<Variables> arguments =
    new Vector<Variable>();
  for(T in R0.getArgumentTypes()){
    arguments.addLast(strategy.getInstanceOfType(T));
  }
  try {
    new Variable(languageBinding.call(R0,arguments));
  } catch (Exception e) {
    if (languageBinding.representsFailure(e)){
     foundBugs.add(e);
    }
  }
}
```

Figure 2. Algorithm of automated random testing. Bolded source lines show where a dynamic language cannot provide the necessary typing information.

Similarly to other automated random testing tools, YETI follows the algorithm in Figure 2. In this algorithm, getModuleToTest, getRandomRoutineFromModule, and getInstanceOfType are defined within the strategies. How to make a call (call) and how to interpret the results (representsFailure) are both defined in the language binding.

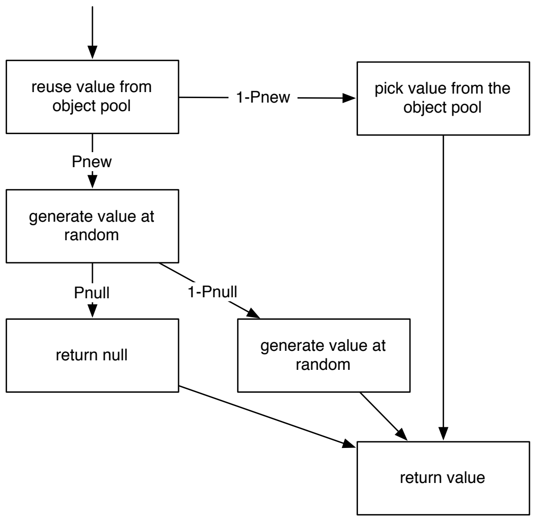


Figure 3. Generation of values.

By default YETI uses a strategy that generates calls and selects values at random. Two main probabilities can be adjusted: the percentage of null values p_{null}, and the percentage of newly created objects to use when testing p_{new}. Figure 3 shows the overall process followed when YETI needs an instance for a test and calls getInstanceOfType. By default, YETI uses 10% as a default value for both p_{new} and p_{null}.

In the Java binding, YETI uses class loaders to find definitions of classes to tests, reflection to make calls, and a separate thread to run them. Any undeclared RuntimeException or Error is interpreted as a failure and failures are grouped into unique failures by comparing their call stack trace beyond the first line. Receiver and arguments are discarded: to avoid the fact that if we would reuse sets of arguments/receiver we could trigger the same bug.

To understand how strong typing impacts the testing process, we identify 4 main areas where the typing information is used extensively. For each of these areas, we indicate whether a dynamically typed language has an easy way of supporting it:

Type description and pool of objects. Types are necessary to construct message arguments. In YETI, a type is mainly made of a list of supertypes, a list of subtypes, a list of "constructors" (by constructors we mean method returning an instance of the class), and a list of instances. Constructors are all routines that return a value of the type, and the list of instances is a pool of objects of that type.

Many dynamically typed languages offer support for listing existing instances for a given class, which can be used as example for feeding the testing tool. Supertypes and subtypes are easily and efficiently accessible via reflection. It is however more difficult to know the return type of a routine because 1) it is not declared and 2) it may vary over multiple executions.

Instance generation. In YETI, generating an instance is done through a "constructor" for such an instance. If the routine to call needs arguments, these are typed and it is easy to reuse instances of such types already present in the system through their type or create new ones through their own constructors.

Test generation and execution. To generate and execute a test, YETI either uses or generates instances of the needed types and makes a call.

Without restriction to given types for the arguments, calling a method with random instances is very unlikely to produce a meaningful test.

Feedback and bug identification. In YETI, because the type is supposedly valid due to the typing information, collecting runtime exception is meaningful as only a superficial check on the APIs is needed to make sure that unique failures are in fact bugs.

As mentioned previously, testing an untyped routine leads to calls that have a high risk of failing. It is also not useful to know that a routine fails when using arguments of types which were not foreseen to be usable there.

As we can see in the previous description, typing information is used at every single step of the process.

2.2 YETI Facts

By using such typing information YETI is able to run a million tests per minute – this represents a barely noticeable overhead over reflexive invocation. So far, YETI found thousands of bugs and was able to test many programs such as the Java core libraries, GWT [2], and all programs in the Qualitas Corpus [TAD+10]. YETI also has a graphical user interface which allows the test engineers to monitor how a testing session performed so far.

3. Challenges for Dynamic Languages

We now explore the challenges and possible solutions posed by the absence of static types to support automated random testing.

3.1 Instance generation and execution

Obtaining instances to execute the tests is key in automated random testing. Instances are necessary to feed random tests. Constructing messages (which received objects as argument) intended to be sent to object receivers are used as input of a test.

In Smalltalk, the fact that no constructor is natively supported by the language makes the generation of well formed instance challenging. One way to circumvent this lack of constructor is to use existing instances (and thus well formed) as an input for random testing. An approach is to use heuristics based on method categories and some patterns for methods. The fact that in Pharo and Squeak the method new automatically invokes the method initialize is a good step to obtain more systematically well initialized instances.

Getting instances is easily done through exploring the instances currently in the system. In Smalltalk, it is possible to get access to all the instances of a class using messages allInstances. Reflection is simple and efficient to use. Accessing existing instances can provide specific information that generated instances could not present.

However, using existing instances as input in random tests presents two issues:

1. *Partially initialization* – Since Smalltalk does not offer any guarantee on whether an object is properly initialized or not, the standard Pharo distribution contains uninitialized or partially initialized instances. Those objects have their invariant broken. As an example we have found an instance of the class Point which contains the nil value as x and y and totally unused by the system. A properly initialized point contains numerical values instead.
2. *Fragile objects* – Randomly modifying existing objects may lead to situation where the system is put in a dangerous state. Such objects may either be discarded for not being used as a random test input or be copied. Copying objects may be an option, however it may lead to an overhead in case of a deep copy, if ever possible at all.

In addition, since we do not know the supposed type of the arguments, it is not clear what instances to pass.

Using type inference. To know the argument type, one possibility is to use type inference. Different approaches are possible: lightweight with low precision [PMW09] or heavy computation (and with the problem of the availability of the approach) [SS04]. We believe that a lightweight approach is a first step to be used. As an example, RoelTyper only considers single method body to identify argument type of the method, therefore the precision of the type information is low. Another heavy approach could be to annotate methods with type information but this clearly does not scale.

Using libraries like MethodWrappers or other techniques to monitor method execution [BFJR98, Duc99] could be a possible way to collect type information. However, this is not the panacea because (1) we face a bootstrap problem - we need to use some pre-existing information (existing tests could be used to run but again the results would be driven by test quality and in absence of large coverage we may get incomplete type information). In statically type languages, the process does not require existing tests to run. (2) not all the methods should be executed randomly.

[2] http://code.google.com/webtoolkit/

Executing destructive methods. Once we get access to instances to which messages can be sent, we should consider which methods to invoke. There again care is important because it is not trivial to identify potentially destructive methods. Smalltalk offers powerful features such as pointer swapping, class changing, quitting the VM, unloading classes.... Not invoking these methods is important. The Finder tool manually specifies a list of problematic methods. Another approach can be to ask programmers to tag the methods with meta information. As a general point, we believe that knowing methods which may endanger the system when run randomly is a valuable information that can make the system more robust in presence of tools such as the Finder in Pharo or Squeak [BDN+09].

3.2 How to identify errors?

While the other problems can be fixed with the (tedious) addition of meta-data, identifying that a bug is found is a difficult question. Indeed raised exceptions are not part of method declaration in Smalltalk as this is the case in Java. Therefore we cannot simply identify a bug by looking if the exception raised was part of the declared raised exception. In addition, since there are no contracts, they cannot be used to filter wrongly passed arguments. In addition, since a wide range of objects can be passed as arguments (and could be invalid) we cannot use the fact that the arguments are valid when analyzing an exception. The combination of the absence of argument type, contracts and declared exception is clearly what makes the real error identification a challenge.

As a first way to distinguish between various situations we can consider the object receiver contained in the top frame of the method stack and to identify different errors.

Receiver on top of stack and message not understood. The idea here is how do we make sure that one sends messages that are understood by the receiver. Such situation should not happen because the tools should enumerate the methods from the receiver class. However, even such simple assumption is not always true: a message can be cancelled in superclasses using "should not implement" exception and superclass methods are banned from subclasses.

Receiver not on the top of the stack and error. This is the regular case when we get an error is because you send a message and this method sends another one and along the road there is an error. The question then is what to do? Such a situation can arise due to different causes:

Badly initialized objects. For example, executing methods on an object that is not well initialized or whose invariant is broken can happen. The automatic detection of such case is difficult. Such object can either be created by a bogus initialization or the result of inadequate method execution. It is thus important to know that instances in the instance database are in a valid state.

Breaking precondition. It may happen that messages not respecting non explicit invariant lead to errors. For example sending the message reciprocal to 0@0 leads to a problem because x reciprocal does not work on number zero.

```
(0@0) reciprocal
   -> x reciprocal @ y reciprocal.
      -> x reciprocal
```

Probably the receiver being different of 0 should be a precondition on Number»reciprocal and similarly, x != 0 and y != 0 should be the precondition of Point»reciprocal.

Another example is #() first (accessing the first element of an empty array). Here clearly a precondition would help to capture that this behavior is not an error but just a normal behavior. In addition considering specific error raised by the class can be a good starting help.

```
#() first
   -> errorSubscriptBounds:
```

The definition of the at: method (which leads to the previous error) shows that some errors could be used to build up a list of raised errors. Such errors could then be used to define preconditions.

```
Object>>at: index
   "Primitive. Assumes receiver is indexable.
Answer the value of an indexable element in the receiver.
Fail if the argument index is not an Integer or is out of bounds.
Read the class comment for a discussion about that the fact
that the index can be a float."

   <primitive: 60>
   index isInteger ifTrue:
      [self class isVariable
         ifTrue: [self errorSubscriptBounds: index]
         ifFalse: [self errorNotIndexable]].
   index isNumber
      ifTrue: [^self at: index asInteger]
      ifFalse: [self errorNonIntegerIndex]
```

Since there is no contract declared, software bugs cannot be distinguished from contract violation. This clearly shows that preconditions could be useful for automated random testing in addition to the other properties they bring to software quality.

3.3 Understanding results

While performing some preliminary experiments with automated random testing in Smalltalk, we identified also the following challenge: How can we identify the impact of a bogus instances on the resulting generated errors?

For example we got an instance of Point, the point nil@nil as an available instance and it generated a lot of false positives by generating errors when executing methods that were totally correct when we picked any other instances available in the system. We identified this instance in a ad-hoc way and we believe that tools to support the understanding of the results are needed. Notice that this problem does not happen in YETI because YETI uses types and contracts as specifications for valid input. In Smalltalk, this is obviously not possible so the nil@nil example is a good one.

If somebody had specified formally Point, then nil@nil would have been illegal. In the case of Pharo, this instance lives in the system, so it was taken as possible input, and polluted your output. We analyzed why such instance was there and it was unused. Since Pharo 1.3 this instance was removed, still this is an interesting case for understanding the impact of an instance of the resulting method that are raising errors.

To illustrate this challenge, here are the data we got when performing our experiments. The existing instance nil@nil broke a lot of messages: in our experiments we obtained 3695 problematic messages. While checking the problematic messages we found that we obtained 371 buggy receivers (points with at least one zero) and a number of buggy methods r, reciprocal, guarded, negated, degrees, max, theta, asFloatPoint, abs, isZero, truncated, rightRotated, eightNeighbors, ceiling, leftRotated, asIntegerPoint, min, normal, transposed, normalized, sign, floor, fourNeighbors, deepCopy, fourDirections, angle, rounded.

When filtering the buggy selectors by removing the bogus instance (nil@nil) from the instance database, we reduced the number of problematic methods to two: normalized and reciprocal. Such methods raised errors because of points with one zero. We identified the problems with nil@nil because we looked at all the instances that generated errors and noticed it. In this case this was simple: we got points with nil, or at least one zero. We tried to see how a tool could have help us and report a problem but we did not found a simple approach based on a correlation or statistics. In our dataset, sorting the number of problematic methods according to the receiver was not a great help.

Being able to represent results and the influence of receiver/arguments on the generated problems is thus of importance to reduce noise.

4. Paths towards random testing

To enable automated random testing for dynamic languages, we propose to follow these milestones:

- *Dynamic Type Inference using Random Testing.* One of the first actions to be done is to use type inference to get some type information. The minimum is to use a simple static analysis as proposed by RoelTyper [PMW09]. Combining an approach like the one of RoelTyper with the type collected by MethodWrapper is an interesting track to follow. However, using MethodWrapper or any execution based appraoch requires both tests availability and
- *Random testing for dynamically typed languages using existing instances.* Once type inference is available, it becomes possible to use instances in an image and perform either random testing or exhaustive testing of programs in an automated way. The performance associated to the testing is then dependent on the quality of the existing instances. It seems thus likely that such a technique should be applied while the program to test is stalled after being run for a while.

  In parallel, identifying classes defining an initialize method (or inheriting one) should be considered to identify classes where creating instances using the method new may provide well initialized instances.
- *Automated random testing for dynamically typed languages.* The next step is to be create meaningful instances at random. As of now, creating random instances is easy. Making sure these are meaningful is quite difficult without additional support. It might be useful in this case to add support for contracts (pre-, postconditions, and class invariants) and to let programmers specify such contracts to decide whether instances are valid for testing. This might, for example, allow the filtering for testing of instances such as nil@nil.
- *Sandboxing for testing.* As a more long-term goal it might also be interesting to consider how to restrain testing so that it does not corrupt the tested images and external resources.

Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the 'Contrat de Projets Etat Region (CPER) 2007-2013' and the Royal Academy of Engineering.

References

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[BFJR98] John Brant, Brian Foote, Ralph Johnson, and Don Roberts. Wrappers to the rescue. In *Proceedings European Conference on Object Oriented Programming (ECOOP'98)*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998.

[COMP08] Ilinca Ciupa, Manuel Oriol, Bertrand Meyer, and Alexander Pretschner. Finding faults: Manual testing vs. random+ testing vs. user reports. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, Nov 2008.

[CPL+08] Ilinca Ciupa, Alexander Pretschner, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. On the predictability of random tests for object-oriented software. In *International Conference On Software Testing, Verification And Validation (ICST 2008)*, July 2008.

[CPO+11] I. Ciupa, A. Pretschner, M. Oriol, A. Leitner, and B. Meyer. On the number and nature of faults found by random testing. *Software Testing, Verification and Reliability*, 21(1):3–28, 2011.

[Duc99] Stéphane Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

[OT10] Manuel Oriol and Sotirios Tassis. Testing .net code with yeti. In *15th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2010, Oxford, United Kingdom, 22-26 March, 2010*. IEEE Computer Society, 2010.

[PMW09] Frédéric Pluquet, Antoine Marot, and Roel Wuyts. Fast type reconstruction for dynamically typed programming languages. In *DLS '09: Proceedings of the 5th symposium on Dynamic languages*, pages 69–78, New York, NY, USA, 2009. ACM.

[SS04] S. Alexander Spoon and Olin Shivers. Demand-driven type inference with subgoal pruning: Trading precision for scalability. In *Proceedings of ECOOP'04*, pages 51–74, 2004.

[TAD+10] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, December 2010.

PHANtom: a Modern Aspect Language for Pharo Smalltalk

Johan Fabry *   Daniel Galdames
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Santiago, Chile
http://pleiad.cl

Abstract

In the context of our research on Aspect-Oriented Programming, we have a need for a modern and powerful aspect language for Smalltalk. Current aspect languages for Smalltalk however fall short on various points. To address this deficit, we elected to design and build PHANtom: a modern aspect language for Pharo Smalltalk. PHANtom is designed to be an aspect language in the spirit of Smalltalk: dynamic, simple and powerful. PHANtom is a modern aspect language because it incorporates the best features of languages that precede it, includes recent research results in aspect interactions and reentrancy control, and is designed from the onset to be optimized and compiled where possible. In this paper we present the language and outline salient points of its current implementation.

Keywords PHANtom, Aspect-Oriented Programming, Smalltalk

1. Introduction

To address the issue of code for one concern being scattered among different classes in an application, Aspect-Oriented Programming (AOP) proposes to modularize such cross-cutting concerns into a new kind of module: an *aspect*. Aspects are a different kind of module because they not only implement the behavior of a concern, but also specify when this behavior should be invoked. To allow this, conceptually each step of the application is reified in what is called a *join point*. The execution of the application consequently produces a stream of join points that is presented to the aspects. The aspects identify relevant join points, *i.e.,* steps in the execution of the application, by means of *pointcuts* that match on the stream of join points. The behavior of the aspect, called *advice*, is linked to the pointcut such that when a pointcut matches, the corresponding advice is executed. To make this all possible, a special tool called the *aspect weaver* implements all the required infrastructure. One possible implementation strategy is that the aspect weaver is a compiler. This compiler modifies each source code location that produces join points of interest to the aspects, locations known as *join point shadows*. At each join point shadow, the modified code then executes the required advice, as specified by the aspects.

A significant part of the research we perform is in the domain of AOP, where we are interested in interactions between aspects [3], as well as Domain-Specific Aspect Languages (DSALs) [4]. For our latest experiments with AOP we chose to use Pharo Smalltalk, which meant we also required an aspect language that incorporates recent AOP research results, is solid, is powerful and is extensible. When evaluating existing aspect languages for Smalltalk we however found these did not meet our criteria and hence set out to build our own language, called PHANtom.

PHANtom is designed to be an aspect language in the spirit of Smalltalk: dynamic, simple and powerful. PHANtom is a modern aspect language because it incorporates the best features of languages that precede it, includes recent research results in aspect interactions and reentrancy control, and is designed from the onset to be optimized and compiled where possible. PHANtom adds to the state of the art in aspect languages as its joining of recent research results is new. Moreover, it improves on existing aspect languages in Smalltalk on various points, *e.g.* by allowing for advice execution ordering, as is discussed later. We therefore consider that PHANtom is also of interest to the Smalltalk programmer wanting to use AOP.

* Partially funded by FONDECYT project 1090083

In this paper we present the PHANtom language, along with some relevant parts of its current implementation.

The structure of the paper is as follows: in Section 2 the language is introduced, starting with the core interplay between join points, pointcuts, advice, inter-type declarations and aspects in Section 2.1. This is followed by a presentation of the advice ordering features, in Section 2.2, and control of reentrancy in Section 2.3. Section 3 shows points of interest in the current implementation of PHANtom. This is followed by a discussion of related work in Section 4, before the paper concludes and presents future work in Section 5.

2. The PHANtom Language

PHANtom is designed to be an aspect language fully in the spirit of dynamic languages like Smalltalk. This is achieved by combining dynamism, simplicity and power. Firstly, PHANtom is dynamic with regard to changes in the code base: it allows for classes and aspects to be added, removed and changed at runtime. PHANtom is also dynamic in the sense of "dynamic AOP" which means that aspects can be added to and removed from the system at runtime, through the process of deployment and undeployment. Secondly, we aim for simplicity by having a minimal set of language constructs who are as free from usage restrictions as possible. Moreover each of these has a straightforward and uniform behavior. Thirdly all language constructs are first class objects, thus providing all the power of first-class constructs. Additionally, PHANtom provides aspect language features, absent in other aspect languages, that give programmers more power over the order of aspect execution and more powerful reentrancy control.

We call PHANtom a modern aspect language for three reasons. First, it was designed taking into account existing aspect languages. It incorporates what we believe to be the best features of these languages. Second, it incorporates recent important research results that have not yet been used together. Last but not least, it was designed with compilation and optimization in mind. While currently PHANtom is an interpreted language, key future work is to compile aspects. For this we plan to use the new compiler that is being developed for Pharo, Opal, which we selected due to its extensibility features.

A number of existing aspect languages have been influential in the design of PHANtom. We mention these influences here but refer to related work for a description of these languages. From AspectJ [8] we have taken the use of patterns in pointcuts, advice ordering through precedence declarations. AspectS [7] inspired our dynamic AOP features, the use of method wrappers [2], and having all constructs available as first class objects. Eos-U [9] introduced what is known as a more symmetric view of AOP, which is also present in PHANtom: aspects are as classes in that they can be instantiated, their behavior is present in methods, and aspects can match join points of other aspects.

The contributions of PHANtom to the aspect language research domain is the specific combination of the advanced language features of advice ordering (see Section 2.2) and reentrancy control (see Section 2.3). Also, to the best of our knowledge, this is the first time a granularity refinement has been proposed to the static AspectJ aspect ordering scheme (see Section 2.2). Beyond these contributions and considering aspect languages in Smalltalk, to the best of our knowledge, PHANtom is the only aspect language for Smalltalk that uses patterns for pointcut definitions, advice ordering and reentrancy control, and was designed with optimization in mind.

Currently, no special syntax has been developed for PHANtom. Instead, aspects are built by constructing their component parts using standard Smalltalk, *i.e.,* by instantiating the corresponding Smalltalk objects and adding them to the aspect as needed. In this we follow the Smalltalk philosophy that everything is realized by sending messages to objects, even the creation of classes (which is the sending of the `subclass:` message to a class). For the sake of conciseness, a special syntax for PHANtom could be elaborated, but we consider this as future work.

2.1 Overview of the Core of PHANtom

We now proceed with giving an overview of core of PHANtom, by means of some illustrative examples. We show here the interplay between pointcuts, advice, inter-type declarations and aspects by means of a small example aspect described below. In the next sections we detail more advanced capabilities of PHANtom: advice ordering (Section 2.2) and reentrancy control (Section 2.3).

As an example initial aspect, in this section we build an extension to the SUnit test suite that counts the number of assertions that are performed in the execution of a given collection of test classes. Note that, due to the simplicity of the example, alternative implementations may be envisioned that provide the same behavior. As it is not the goal of this paper to argue for AOP, but to introduce PHANtom, we have chosen to keep the example plain to ease readability of this text.

2.1.1 Join Point Model

In AOP, the join point model of an aspect language defines the events that are produced by the application when it runs, and their properties. Each such event is a join point, and an aspect defines at which join points its behavior is executed.

Following the goal of simplicity, and in accordance with the Smalltalk philosophy that a computation consists of objects and messages, the join point model of PHANtom consists solely of method executions. Also, each join point contains a reference to the sender, the receiver and the arguments of the message that lead to the method being executed.

2.1.2 Pointcuts

Pointcuts are what is also known as the quantification part of the aspect: a pointcut is responsible for determining when the behavior of an aspect is triggered. More precisely, pointcuts are predicates over the stream of join points that is emitted by the base application as it runs. When pointcut matches a join point, the advice that is linked to that pointcut executes.

Basic Pointcut Definitions

In PHANtom, pointcuts use static information of the join point to decide whether they match, *i.e.,* pointcuts filter the join point stream based on the type of the receiver and the selector name. Pointcuts are instances of the PhPointcut class and an example instantiation of a pointcut is below:

```
PhPointcut receivers: 'TestCase' selectors: 'assert:'.
```

The above code matches all executions of the method with selector assert: where the receiver of the method is an instance of TestCase (excluding its subclasses).

Type	Pattern	Meaning
Receiver	+	Class and Subclasses
Receiver	*	String Match
Selector	#()	All Selectors
Selector	_	One Keyword of Selector

Table 1. Patterns and their meaning.

PHANtom allows AspectJ-like patterns to be used when specifying a receiver class as well as selectors, outlined in Table 1. Allowed patterns for receiver class are as in AspectJ. They allows for the use of two special symbols: a + which is used as a suffix to a class name, and a * which is used anywhere within the class name. The + indicates that the receiver of the method should be an instance of the indicated class, or an instance of a subclass. The * is the string match like in regular expressions. For selector patterns, an array of size zero: #(), matches all selectors and the wildcard _ may also be used[1]. This wildcard matches unary and binary messages, as well as one keyword in a keyword message. In a keyword message with multiple keywords, _: matches one keyword and may be used more than once. For example, with:_:with:_: matches with:for:with:and:, with:with:with:with:, and so on.

Using the patterns we now show the pointcut that will be used in the example assertion count aspect. Let us assume we wish to count all assertions,which can be either assert:, assert:equals:, assert:description: or assert:description:resumable:. However, we only wish to do this for a select group of test case classes, namely all subclasses of ACTestCase. The pointcut that matches those join points is given below:

[1] as * is a valid selector.

```
PhPointcut receivers: 'ACTestCase+'
           selectors: #(assert: assert:_: assert:_:_:)
```

This code shows the use of the + symbol and the ability to use an array of selectors. The use of patterns has been described above. Using arrays of selectors allows us to define a pointcut that matches on different selectors. Recall that passing an array of size zero matches on all selectors. The receivers may also be specified as an array, where each element in the array is a class name pattern.

Context Exposure

Context exposure is the means by which information that is present in the join point is made accessible to the advice. Declaring this in the pointcut allows the aspect weaver to perform optimizations: avoiding creation and/or reification of elements that are not used in the advice [6]. If context exposure is specified in a pointcut, the corresponding advice will be passed an instance of PhContext that contains the specified context values. These can be as follows:

- receiver includes the object that received the message
- sender includes the object that sent the message
- selector includes the selector of the message
- arguments includes the arguments of the message
- proceed allows for the original behavior to be invoked
- advice allows for advice execution order to be modified

Most of the above context values are straightforward. An example of the use of proceed will be given in Section 2.1.3 and an example of using advice is present in Section 2.2.2. Below we show how to add context exposure to the above pointcut, specifically including the receiver object in the context. This yields a pointcut that will be useful for our example aspect, and hence we store it in a temporary variable for later use.

```
pc := PhPointcut receivers: 'ACTestCase+'
                 selectors: #(assert: assert:_: assert:_:_:)
                 context: #(receiver)
```

Package Restrictions

A different pointcut that might be used for the example is given below. It assumes all tests to be counted have a name that starts with Test and are located in a category named Tests-Mine.

```
PhPointcut receivers: 'Test*'
           selectors: #(assert: assert:_: assert:_:_:)
           context: #(receiver)
           restrict: #(Tests-Mine).
```

This pointcut uses a * as receivers specification to indicate all classes that start with the name Test. It also provides a restrict: specification, which restricts the matching of classes to those which are present in the array of specified categories.

Note that the above pointcut does **not** match on instances of metaclass, even though their name, *e.g.* TestFoo class, matches the regular expression, Test* in the example. To mach instances of metaclass, the receivers pattern must include the word class. In the example, a match would instead be made when using the pattern Test* class.

Pointcut Combinations, Custom Pointcut Matching

Pointcuts can also be combined using logical and, or and not operators. PhPointcut instances understand the corresponding messages pand:, por: and not. The former two take a second pointcut as argument and perform, resp. the logical and, or[2]. The context exposure of the resulting pointcut is the union of the context exposures of both pointcuts. Sending a not returns the negation of the receiver pointcut.

To provide more power for pointcut specification, PHANtom allows for the programmer to define custom pointcut patterns if required. This is made possible by using the PetitParser parser generator framework [10] in the implementation of the matching of pointcut patterns. Instead of using a string or array as argument of the receivers or selectors specification, a PetitParser parser can be passed as an argument for either, or both. This parser is given the complete class description of candidate classes, or candidate selector strings, respectively. If the parser matches, the class is a valid receiver or selector. A complete discussion of PetitParser parsers is out of the scope of this paper. We only provide the following example that uses two custom parsers. It matches the count and count: messages of instances of classes that have one instance variable named count.

```
PhPointcut
  receivers: (#any asParser plusGreedy:
    'instanceVariableNames: ''count''' asParser)
  selectors: ('count' asParser , ':' asParser optional).
```

2.1.3 Advice

Next to the quantification part of the aspect, the pointcut, the advice determines the behavior to be executed when a pointcut matches. In PHANtom, we choose to have the behavior implemented as regular methods, inspired from Eos-U [9]. Advice, which are instances of PhAdvice, merely associate a pointcut to the execution of this behavior. This is different from Eos-U, which does not have the notion of advice. It instead places responsibility of determining what method to run on the pointcut itself. This however restricts reuse of the pointcut, as it is now tightly coupled to a specific method name for the advice. Furthermore, the interplay of specifying a method in the pointcuts with combining two pointcuts (with pand, por) is not immediately obvious. An argument can be made for various different choices of when and how the behavior of one of the constituent pointcuts should be executed when the composed pointcut matches. This adds complexity to the language, which we aim to avoid.

Creation of a PhAdvice takes as arguments the pointcut to be used, the name of a message to send and its receiver, and when this message must be sent. The sent message will be given one argument: the join point context as exposed by the pointcut. Note that, for the sake of brevity, in the remainder of this text we will use the term *executing the advice* to mean the execution of the method that results from the message sent by the PhAdvice instance when the pointcut matches. Also, in this section consider a situation where only one advice needs to execute at a given join point. The case where multiple advice execute at one join point is discussed in detail in Section. 2.2.

Below is the advice for our assertion count example:

```
adv := PhAdvice pointcut: pc
                send: #incCount: to: self
                type: #after
```

This code specifies that we wish that matches of the pointcut pc, which we declared in Section 2.1.2, trigger the sending of the incCount: message to self (whose nature will be revealed in Section 2.1.5). This message will be sent after the assertion method has executed.

There are three types of advice: *before*, *after* and *around* advice. Before advice executes before the join point, *i.e.,* before the selected method execution, after advice executes after the join point and returns the value of the join point execution. Around advice executes instead of the selected join point, effectively replacing the original behavior of the method with the behavior of the advice. When the advice executes it is however possible to invoke the original behavior of the method by sending the proceed message to the context. The original behavior can be executed with different arguments, by using the proceed: message and passing an array of new arguments. Note that the proceed and proceed: messages may be sent any number of times.

Strictly speaking, before and after advice are redundant. This is as the behavior of the former can be obtained by using an around advice that ends with a proceed, and the latter by an around advice that starts with a proceed (returning the result of the proceed send at the end of the advice). The use of before and after advice however allows the aspect weaver to perform optimizations, as they do not require the original behavior, nor receiver and arguments to be reified into the context.

To allow for more flexibility and conciseness, and inspired from AspectS, instead of a message send specification a block may also be used by a PhAdvice. The block takes as argument the context, and instead of sending a message, advice execution consists in evaluating the block. For example,

[2] They are called thus to avoid an overly aggressive smalltalk compiler optimization of the and: and or: messages

the code below will print the receiver on the transcript before the assertion method executes.

```
PhAdvice pointcut: pc
        advice: [:ctx | Transcript show:
            (ctx receiver asString); cr.]
        type: #before.
```

Note that this feature is mainly intended as a facility for fast prototyping. Its use at a larger scale is discouraged as it does not combine well with advanced features such as dynamic advice ordering (see Section 2.2).

2.1.4 Inter-Type Declarations

Inter-type declarations, also known as introductions or structural aspects, are a means to perform modifications of classes such that they contain the desired state and behavior required for the functionality of the aspect. In aspect languages for statically typed languages, *e.g.* AspectJ, these modifications may also include modifications to the type hierarchy to obtain the required behavior.

In PHANtom, inter-type declarations are purely a means to permit modular class extensions. They are instances of PhClassModifier and they specify additions of variables and methods. This may be performed both at instance and at class side of the target class. The motivation of this class extension mechanism in addition to the standard class extensions in Pharo is twofold. First, inter-type declarations allow the addition of variables, whereas standard class extensions do not allow variables to be added. Second, the inter-type declarations can be textually contained within the definition of the aspect. Arguably this is the module where they should be placed. Standard class extensions place their definition in the target class, which then also appears in the same category of the aspect.

For our running example, we need the test cases to contain a variable that maintains the number of times that assertions have been called. For it to be available to the aspect we also require accessor methods that expose this variable. This is realized by the below two inter-type declarations:

```
cm1 := PhClassModifier on: ACTestCase
    addIV: 'phacount'
cm2 := PhClassModifier on: ACTestCase
    addIM: 'phacount
        ↑phacount ifNil: [phacount := 0]'.
cm3 := PhClassModifier on: ACTestCase
    addIM: 'phacount: anObject
        phacount := anObject'.
```

The first PhClassModifier instance declares the addition of a phacount instance variable to the ACTestCase class. The second instance adds the corresponding accessor to the ACTestCase class. The third declares the corresponding mutator. As in AspectJ, no patterns for a type are used here, so the target class is specified directly.

When the aspect is deployed (see Section 2.1.5), the inter-type method variable additions will be added to the target class that is then recompiled. If the class already contains this variable, an exception will be thrown. Similarly, method declarations will be compiled within the scope of the target class and added there. If methods with the same selector are already present in the class, an exception will be thrown.

To add class variables, resp. class methods, the PhClassModifier needs to be instantiated with the on:addCV:, resp. on:addCM: messages.

2.1.5 Aspects and their Deployment

An aspect is a class that modularizes cross-cutting behavior, by means of pointcuts, advice and inter-type declarations. In PHANtom, aspects are subclasses of the PhAspect class, and contain a collection of PHAdvice and PHClassModifiers (and each PHAdvice keeps a reference to a PhPointcut instance). Instances of aspects are not automatically active. To add the inter-type declarations and have advice execute when their pointcuts match, aspects must be deployed. Conversely, a deployed aspect can be undeployed, which removes its effects on the system.

To complete our running example, we first need to subclass PhAspect and add the initialize method as below,

```
initialize
|pc adv cm1 cm2 cm3|

"... take pc, adv, cm1, cm2, cm3 from above ..."

self add: adv.
self addClassModifier: cm1.
self addClassModifier: cm2.
self addClassModifier: cm3.

self install.
```

Adding the advice to self achieves that the incCount: message is sent to the aspect after all executions of the assert methods identified by the pointcut of the advice. Adding the three class modifiers ensures that ACTestCase understands the phacount and phacount: messages and behaves accordingly.

All that remains to be done is to define the behavior of incCount:, which is done by adding the below method to our aspect:

```
incCount: aContext
   aContext receiver phacount:
      aContext receiver phacount + 1
```

The implementation of incCount: uses the context to obtain the receiver (an instance of ACTestCase or a subclass thereof) and simply increments the count by one.

Using the aspect is straightforward, as instantiating sends it the install message that deploys it in the system. This immediately makes all ACTestCase and subclass instances count their assertions. The assertion count of each can be obtained through the accessor method when needed. Sending the uninstall message to the aspect undeploys it[3].

Note that the aspect weaver maintains a list of all installed aspects, which is obtained as follows: PhAspectWeaver installedAspects. This can be useful to uninstall aspects whose references are out of scope.

This concludes our overview of the core of PHANtom, where we discussed how instances of PhPointcut, PhAdvice, PhClassModifier and PhAspect can be combined to modularize a cross-cutting concern.

2.2 Advice Ordering

In the above discussion we assumed that at a given join point only one advice will execute. In this section we detail the behavior of PHANtom when various advice need to execute at one given join point. We now first specify the default behavior, before considering the need for program-specific behavior and detailing the two advice ordering features offered by PHANtom.

By default, if multiple advice specify that they should be run at a given join point, they will be run in sequence. The entire process is illustrated in Figure 1. First all before advice are run in an unspecified order. Then an around advice is run (if any need to be run). If this advice sends a proceed message, instead of executing the original behavior, another around advice is run (if any more need to be run), and so on. Last, all after advice are run in an unspecified order.

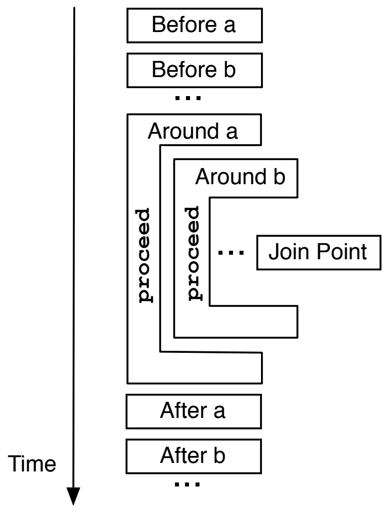


Figure 1. Execution order of multiple advice for one join point

An important issue when multiple aspects, or multiple advice, are present is however their order of execution as it can significantly impact the behavior of the application.

[3] Although we speak of deployment and undeployment of aspects, these messages are called install: and uninstall, which was how they were originally named in AspectS [7]

Aspect execution order is one case of interaction between aspects [3]. A typical example is the interaction between a timing and a logging aspect, which is as follows: Timing adds around advice to a number of methods, sending a proceed message and registering the time it needs to execute. Logging adds around advice to a number of methods, sending a proceed message and logging it. When both aspects affect the same method, the situation is similar to the around advice execution depicted in Figure 1. Timing will or will not include the time taken for the log operations, depending on which advice executes first, *i.e.,* whether Around a is the timing advice and Around b is the logging advice or vice-versa. The required behavior is clearly a decision that should be taken when developing the application, and hence the aspect language should provide for a means to define an ordering for the execution of advice.

PHANtom has two mechanisms to define the ordering of advice. It first has a precedence relationship inspired from AspectJ, which we discuss next. Second it has a fully dynamic advice order list that can be manipulated by the advice themselves, which is discussed afterward.

2.2.1 Deployment Ordering

The first aspect ordering scheme provided by PHANtom is a refinement of the static aspect precedence declaration scheme of AspectJ, adapted to a setting with aspect deployment. A precedence declaration determines that an aspect, or a set of aspects, is more important than another (set of) aspect(s). PHANtom implements the global precedence scheme of AspectJ and adds to it a local precedence scheme. In Table 2, we give an overview of both precedence schemes, and we talk about them next.

To declare a global precedence order, the precedence: message is sent to a PhAspect instance, with as argument an ordered collection of receiver class patterns (see Section 2.1.2). The first element of the collection determines the most important aspect(s), the second element determines (a) lesser important aspect(s), and so on.

The group of all declared aspect precedence relationships should form a partially ordered set of aspects, which allows a directed acyclic graph of aspect dependencies to be formed. When a declared precedence does not respect the partial order, a cycle would be formed in the dependency graph. At aspect deployment time the graph is therefore verified for cycles. If any are present, an error is produced and the aspect is not deployed.

The aspect dependency graph is used at advice execution time to order the advice execution as in AspectJ, which is as follows: The order of before advice and around advice is from more important to less important, *i.e.,* executing the advice of the most important aspect first. The order of execution of after advice is however the reverse: from less important to more important, *i.e.,* executing the advice of the most important aspect last.

Receiver	Message	Ordering
aPhAspect	precedence:	Add precedence declarations to the global order.
aPhPointcut	precedence:	Add local order to the global order, ignore conflicting global precedences.
aPhPointcut	overridePrecedence:	Ignore all precedences of the global order, only use the local order.

Table 2. Deployment ordering specification API.

Fine-Grained Deployment Ordering

In addition to declaring a global precedence order, a precedence order with a more fine-grained scope can be declared, if necessary. To the best of our knowledge, this is the first time such a refinement has been proposed to the AspectJ aspect ordering scheme. The finer-grained ordering is performed by sending the precedence: message to a pointcut. As a result, the precedence specification that is passed as argument is present in the join points that match the pointcut. In other words, in those join points the precedence order is the global order that is extended with the order declared on the pointcut. The order declared on the pointcut is considered to be more important than the global order: If joining both orders leads to cycles in the dependency graph, the precedence declarations of the aspect that cause the cycle are ignored.

Lastly, PHANtom also allows for the precedence of a pointcut to override the global precedence, by sending the overridePrecedence: message to a pointcut. This sets the precedence order for the matched join points to be exclusively the order that is declared in the pointcut. If this order contains conflicts, an error is produced at aspect deployment time

2.2.2 Dynamic Ordering

Combining dynamism and power, PHANtom also allows for aspect ordering to be decided when advice is executing. This is inspired by the work on Dynamic AspectJ by Assaf and Noyé [1]. In this work, the authors argue for a more dynamic approach to the topic of multiple advice execution. They propose an extension to AspectJ that allows advice execution to be scheduled dynamically, including the possibility for some advice execution to be skipped. At runtime, each advice can obtain a representation of the list of advice that is scheduled to be executed at this join point, which is called an *AspectGroup*. Operations on the AspectGroup allow execution of all advice to be skipped, execution of advice with a certain name to be skipped, and positioning advice with a given name at the head of the sequence. By repeatedly using the latter operation the sequence can then be reordered.

In accordance to the aim of simplicity, the API of PHANtom is conceptually more straightforward and less restrictive than Dynamic AspectJ, and is shown in Table 3. In PHANtom, a pointcut may expose the advice ordering in the context through the advice keyword. The context can then be queried for the list of scheduled before, after and around advice, as well as for the currently executing sequence of advice. This returns an ordered collection of such advice. This collection can be manipulated as required by the advice, and the context can be given a new sequence of scheduled advice, setting the execution sequence. When changing the group of scheduled advice to which this advice belongs, it is however ambiguous where in the sequence the advice execution process should continue. To avoid this, such a change must be performed by sending the currentAdvice continueAt: message. This message also indicates at what index in the sequence is the advice that should be run next, removing the ambiguity. The other advice setter messages (beforeAdvice: when used in a before advice, . . .) are silently ignored.

2.3 Reentrancy Control

A common pitfall in the use of aspects is the appearance of infinite loops when aspects are added to the application. Typically what occurs is that the pointcut of an advice captures the behavior of the advice itself. An example of this is shown in Figure 2. When the advice executes, its pointcut matches its own execution. As a result, the advice executes and its pointcut matches its own execution. Consequently, the advice executes, and so on.

```
|asp|
asp := PhAspect new
  add:(PhAdvice
    pointcut: (PhPointcut
            receivers: 'Transcript class' selectors: 'show:')
    advice: [ Transcript show: 'Showing ' ]
    type: #before).
asp install.
Transcript show: 'reentrancy control'; cr.
```

Figure 2. A potential infinite loop as part of the advice execution is captured by its own pointcut.

In this section we introduce the reentrancy controls provided by PHANtom. Based on the concepts of computational membranes, PHANtom provides a safe default, that can however be modified by the programmer if required.

2.3.1 Computational Membranes for Reentrancy Control

A number of proposals have previously been made that aim to avoid such reentrancy issues. The recent work by Tanter on execution levels for aspect-oriented programming [11] provides a thorough analysis of the problem, and a solution

Receiver	Message	Result
aPhContext	beforeAdvice, beforeAdvice:	Obtain, set sequence of before advice
aPhContext	aroundAdvice, aroundAdvice:	Obtain, set sequence of around advice
aPhContext	afterAdvice, afterAdvice:	Obtain, set sequence of after advice
aPhContext	currentAdvice	Obtain sequence of advice that is currently executing
aPhContext	currentAdvice:continueAt:	Set currently executing sequence and index of where to continue

Table 3. Dynamic ordering specification API.

that supersedes previous proposals. The latest work of Tanter *et al.*, *computational membranes* [12], is a generalization of execution levels that provides more flexibility. PHANtom uses these computational membranes to provide for reentrancy control.

At its core, the work on membranes proposes to deploy membranes around a given computation, to serve as a scoping mechanism for the join points emitted by that computation. Membranes are then used to structure execution of aspects. Membranes can be nested, resulting in a hierarchy of membranes, and crosscut, when multiple membranes are deployed around (a part of) the same computation. Figure 3 (taken from [12], with permission) illustrates how membranes can be deployed on a control flow of three computations X, Y and Z. Membranes m1 and m2 crosscut, as they both wrap the Y computation.

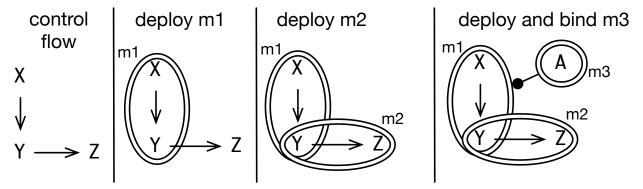


Figure 3. Deployment of membranes (taken from [12], with permission). Membranes m1 resp. m2 wrap computation X and Y resp. Y and Z. Membrane m3 is bound to m1 causing join points from the computation in m1 to be visible to the aspect A.

Membranes scope the observation of join points by controlling the propagation of join points of the computation on which they are deployed. For an aspect to observe a join point, it needs to register itself in a membrane. In Figure 3, the aspect A is registered in the membrane m3. All the join points that are observed by that membrane are visible to the aspect, *i.e.,* can be matched by its pointcuts. To observe a join point, a membrane however must first be bound to a membrane that generates join points (which may be itself). In the example, the membrane m3 is bound to membrane m1. As a result, the join points generated in membrane m1 'flow' to the membrane m3 and hence are visible to the aspect A. Note that join points that occur in the computation Z will not be seen by the aspect A, as the membrane m1 does not capture this computation.

Note that membranes are more powerful than what is presented here, and PHANtom only implements the subset of the capabilities of membranes that allows for topological scoping of join points. For example, the membranes proposal posits that membranes may filter the incoming and outgoing join points. For more information we refer to the membranes publication [12].

2.3.2 Addressing Reentrancy Issues

Following the goal of simplicity, PHANtom provides straightforward behavior: it by default avoids the typical reentrancy issues we identified above. Recall that in Figure 2 the advice sends the show: message to the Transcript, which is captured by its own pointcut[4]. Without reentrancy control, it hence leads to an infinite loop. In PHANtom, the code however does not loop and solely prints Showing reentrancy control on the transcript. This is achieved by letting the computation that is intercepted by the aspect (*i.e.,* Transcript show: 'reentrancy control'; cr.) be contained by its own membrane, and the computation of the aspect be a separate membrane in which the aspect is registered. The aspects' membrane is then bound to the computations' membrane. Join points of the computations' membrane flow to the aspects' membrane, but join points of the aspects' membrane do not flow to itself. Hence the aspect does not capture its own computation and thus does not loop.

2.3.3 User-Specified Membrane Topologies

PHANtom provides the programmer the power to define a custom topology of membranes and their binding relations, which aspects are registered in what membranes, and on what computation the membranes are deployed by means of pointcuts. The API of membranes is shown in Table 4. It allows the programmer to straightforwardly specify situations where join points emitted by aspects need to be visible to (other) aspects, in a structured manner.

The membranes paper [12] provides a number of examples of how such advanced topologies can be achieved through the use of membranes. We repeat here the example used to illustrate how a directed acyclic graph of membranes and their binding can be constructed. Suppose that we have a browser computation that is complemented by a cache as-

[4] In Pharo, Transcript is actually an instance of a different class (that depends on the version of Pharo). Hence for the code to work the argument to receivers: needs to be changed to Transcript class asString

Receiver	Message	Result
aPhMembrane	advise:, unAdvise:	Binds, resp. unbinds the argument membrane to the receiver.
aPhMembrane	registerAspect:, unregisterAspect:	Register, resp. unregister an aspect instance in the receiver.
aPhMembrane	pointcut:	Pointcut argument defines computation around which the receiver will be deployed.
aPhMembrane	install, uninstall	Deploy, resp. undeploy the receiver.

Table 4. The Membranes API

pect and a quota aspect. The cache aspect intercepts URL requests by the browser to cache the retrieved pages locally on disk. The quota aspect should verify that disk operations by the browser (*e.g.*saving a page to disk) as well as the cache do not exceed the allowed disk quota. The implementation of this topology is shown below:

```
|browserMem cacheMem quotaMem|
" assuming browser class, cache and quota aspects "

browserMem := PhMembrane new pointcut:
  (PhPointcut receivers: 'Browser' selectors: #()).
browserMem install.

cacheMem := PhMembrane new.
cache registerOn: cacheMem.
cacheMem advise: browserMem.

quotaMem := PhMembrane new.
quotaMem advise: browserMem; advise: cacheMem.
quota registerOn: quotaMem.
```

In this code we see three groups of statements. The first group creates a membrane, stating the computation around which it will be deployed using a pointcut, and deploys it. In the second group, the cache aspect is registered in a second membrane and this cache membrane is bound to the browser membrane. By registering an aspect in a membrane, the membrane is automatically deployed on the aspect, *i.e.,*the join points of the aspect are captured by the membrane. This will enable the quota aspect to match on join points of the cache aspect. In the third group, the quota aspect is registered in its own membrane as well, and the quota membrane is bound to both the browser and cache membrane.

As an illustration of the power and ease of use of membranes, note that if the quota aspect should not need to intercept actions of the browser, the binding of the quota membrane to the base membrane should merely be omitted.

3. Implementing PHANtom

In this section we give a brief overview of the more salient points of the current implementation of the aspect weaver of PHANtom.

Note that while PHANtom is designed with optimization and compilation in mind, currently the implementation is still interpreted. This is because our first priority was to work on establishing the language semantics. Now that these have been stabilized, our next step will be to perform more optimizations and to compile aspects.

3.1 Use of Method Wrappers

As in AspectS, PHANtom makes use of method wrappers [2] to capture join points. This is done by placing method wrappers around methods that are join point shadows of pointcuts used in the aspects. In this section we detail our use of method wrappers.

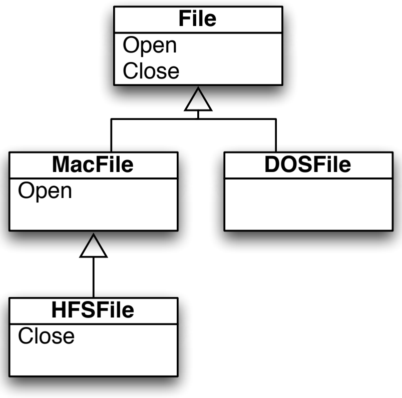


Figure 4. An example File hierarchy.

PHANtom places a method wrapper around join point shadows where advice needs to execute, *i.e.,* methods that are identified in a pointcut. In the presence of inherited and overridden methods, this relationship can not always be obvious as some behavior can be defined locally and some not. Moreover subclasses of an affected class may not be matched by the pointcut and therefore their behavior should be unchanged. To clarify, we present here the relationship between pointcuts and the installed method wrappers by means of an example. Suppose we have a class hierarchy as in Figure 4, we now show where two different pointcuts on this hierarchy cause method wrappers to be installed:

```
PhPointcut receivers: 'MacFile' selectors:#(open close)
```

As open is defined in MacFile, a method wrapper is installed on this method. However, instances of HFSfile inherit the behavior of open while the pointcut does not match them, and hence no advice should execute. To achieve this, the method wrapper first establishes the class of the receiver. If it is different from MacFile, advice execution is skipped.

The close method is inherited from File but may not be modified there, as the pointcut does not identify that class. Hence the method in File, more concretely the instance of CompiledMethod that is in the method dictionary of File, is copied. This copy is wrapped in a method wrapper and added to the method dictionary of MacFile.

```
PhPointcut receivers: 'MacFile+' selectors:#(open close)
```

As open is defined in MacFile, a method wrapper is installed on this method. As above, the close method is copied and the wrapped copy is installed in MacFile. In HFSFile, a method wrapper is installed on the close method, and the open method is copied from MacFile and wrapped. The latter is necessary to differentiate between an execution of open in HFSFile versus one in MacFile. This is needed as a different sequence of advice may need to execute at both points, so we need to be able to differentiate between both.

3.2 Installing and uninstalling aspects

Deploying or installing an aspect is a four-step process, which is as follows:

Installing Class Modifiers. First, the weaver verifies that the class modifiers of the aspect do not redefine existing instance variables or fields. If not, the class is recompiled with new instance variables, if needed, and the new methods are compiled and added to the class.

Membrane Creation. If the aspect is not registered in a membrane, a new membrane is created on the computation identified by the pointcuts used by the aspect. The aspect is registered in the membrane, and the membrane is deployed.

Join Point Shadow Resolution. The membrane of the aspect iterates over the advice of the aspect, collecting the pointcuts that are used there. It then determines the join point shadows for those pointcuts.

Method Wrapper Installation. In each join point shadow a method wrapper instance is installed that is parameterized with the advice collection that executes at the join points of that shadow.

To uninstall an aspect, first the method wrappers are uninstalled, then the membrane is undeployed and lastly methods and instance variables added by class modifiers are removed. To ensure that there are no unforeseen interactions with aspects that have not been removed, all aspects are uninstalled, and the aspects that should not have been uninstalled are installed again. We are aware that this process is clearly far from optimal, and the uninstall process is therefore a prime candidate for optimization in future work.

3.3 Support for Dynamism

In addition to providing dynamic AOP, *i.e.,* the ability to install and uninstall aspects at runtime, PHANtom must also provide support for the dynamic nature of Smalltalk itself. When classes or methods in the system change, this may affect the aspects if their pointcuts match (or no longer match) on these changed classes or methods. In the current implementation changes to the system are detected and treated correctly. As we show below, the implementation of this support for dynamism is however primitive. It is therefore also a prime candidate for optimization in future work.

Method Implementation Change. When a method that is a join point shadow is recompiled, in the method wrapper the old implementation is replaced by the new implementation.

Adding or Removing of Methods. If a method that is a join point shadow is removed from the system, its wrapper is merely discarded. When adding methods all aspects are uninstalled and reinstalled to ensure that the new methods are captured by membranes and aspects if they match the respective pointcuts.

Changes in the Class Structure or Hierarchy. To ensure consistency of join point shadows and of the computation on which membranes are deployed, all aspects are uninstalled and then reinstalled.

4. Related Work

Beyond being the first work on AOP, AspectJ [8] is the reference work for aspect languages. AspectJ is an AOP extension to Java, allowing aspects to be woven at compile time or at class loading time, hence it has no dynamic features. In AspectJ, an aspect can be seen as a class that cannot be instantiated and which may also contain pointcuts and advice. Pointcut are defined in a specific DSL that allows for patterns to specify class and method name, as well as the type of pointcut: method call, object instantiation, field access, and so on. Furthermore, pointcuts have access to the context at runtime, which may be used as part of the predicate, and can be exposed to advice. Advice specifications are similar to methods, named before, after or around, indicating before, after or around advice. Context exposed by the pointcut is made available as arguments to the advice and the pointcut identification is placed before the method body. Lastly, inter-type declarations are field or method declarations whose signature is prepended by the target type name and a dot. AspectJ is a language with an arguably large feature set, is still being developed and is in use by the industry. Due to its significant feature set, a more complete discussion of the features of AspectJ is outside of the scope of this paper

AspectS [7] is a seminal work on aspect-oriented programming, presenting an AOP extension to Squeak Smalltalk. It is the first aspect language to feature dynamic AOP, *i.e.,* the addition and removal of aspects at runtime. AspectS has no dedicated pointcut language, instead relying on the use of metaprogramming to specify the join point shadows, and what are termed *advice qualifiers* to perform runtime testing of the join points. The behavior of advice is given by using a block, and the context is passed to the block as multiple arguments. The implementation of AspectS combines method wrappers [2] with metaprogramming. The former are used to intercept computation at join point shadows, while the latter is used in various places, *e.g.* to determine join point shadows or to obtain the activation context of the message sender. AspectS also provides various extensions of the Squeak browsers to support, *e.g.* navigation between join point shadows and aspects, and vice-versa. AspectS however does not provide explicit support for advice ordering, nor explicit support for controlling reentrancy. Also, AspectS is not completely dynamic, as changes to classes or methods that affect the pointcuts of an aspect are not taken into account if the aspect is already deployed.

Eos-U [9] is an aspect language for C#, in the style of AspectJ, that unifies aspects and objects into a more symmetrical view of AOP. Initially, in AspectJ, aspects were seen as entities fundamentally different from normal classes. In AspectJ, Aspects could not be (and still cannot be) instantiated, and join points in the execution of advice could not be captured by other aspects. Eos-U introduced the concept of *classpects*, which remove these separations between classes and aspects. Classpects can be instantiated and their behavior *i.e.,* their advice, is implemented as regular methods, whose join points may be seen by other classpects. Classpects may contain AspectJ-like pointcut specifications that now also are responsible for identifying the associated advice and its type (before, after or around). The latter however restricts the reuse possibilities of pointcuts and complicates logical combinations of pointcuts. This is why in PHANtom the advice construct is still present and represents the binding between a pointcuts and the behavior to be executed.

PHANtom also builds on the work by Assaf and Noyé on Dynamic AspectJ, which has been discussed in Section 2.2.2, and work by Tanter *et al.*, which has been talked about in Section 2.3. We do not further go in detail on this work here but instead refer to the respective sections.

5. Conclusions and Future Work

We have introduced PHANtom, a modern aspect language for Pharo Smalltalk. PHANtom is an aspect language in the spirit of Smalltalk, combining dynamism, simplicity and power. PHANtom is a modern aspect language as it is inspired by features of existing aspect languages, integrates recent research results and is created with optimization in mind. PHANtom is built, on the one hand, as a base for experimentation of aspect-oriented programming in Smalltalk, and on the other hand, intends to provide a modern aspect language to the Smalltalk programmer.

In this paper, we first gave an overview of PHANtom, starting with the design considerations and detailing what features are taken from which previous work. We then described how pointcuts, advice, inter-type declarations and aspects are created, and how the latter are deployed. We followed this with a discussion of the more advanced features of PHANtom: aspect ordering at deployment time and at runtime, and management of reentrancy using computational membranes.

After the language introduction, we briefly touched on the more noteworthy points of the current implementation. While PHANtom is designed with optimization and compilation in mind the current implementation is an interpreter that makes used of method wrappers. We first described how we use method wrappers to intercept the computation where needed. We then gave an overview of the aspect deployment and undeployment mechanism, and ended with describing how PHANtom supports the dynamism of Smalltalk.

There are many avenues of future work. Firstly, the existing implementation can be optimized, as mentioned in Sections 3.2 and 3.3. Complementary to this, the work on compiling PHANtom is an important task we plan to undertake. Considering the feature set of the language, we want to add dynamic checks to the pointcuts, in the spirit of the if feature of AspectJ pointcuts. Also we would like to develop a specific syntax for PHANtom, to be able to program aspects in a more concise manner. Lastly, as tool support for PHANtom is currently lacking, we are considering adding tool support that allows to see the impact of aspects in the code browser, or even use specific aspect visualizations such as AspectMaps [5].

References

[1] A. Assaf and J. Noyé. Dynamic aspectj. In *Proceedings of the 2008 symposium on Dynamic languages*, DLS '08, pages 8:1–8:12, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-270-2. URL `http://doi.acm.org/10.1145/1408681.1408689`.

[2] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to the rescue. In E. Jul, editor, *ECOOP'98 - Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 396–417. Springer Berlin / Heidelberg, 1998. URL `http://dx.doi.org/10.1007/BFb0054101`.

[3] R. Chitchyan, J. Fabry, S. Katz, and A. Rensink. Editorial for special section on dependencies and interactions with aspects. 5490:133–134, 2009.

[4] T. Cleenewerck, J. Fabry, A.-F. Lemeur, J. Noyé, and É. Tanter, editors. *Proceedings of the 4th workshop on Domain-Specific Aspect Languages*, Charlottesville, VA, USA, Mar. 2009.

[5] J. Fabry, A. Kellens, and S. Ducasse. Aspectmaps: A scalable visualization of join point shadows. In *Proceedings of 19th IEEE International Conference on Program Comprehension (ICPC2011)*, Jul 2011. To appear.

[6] E. Hilsdale and J. Hugunin. Advice weaving in aspectj. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, AOSD '04, pages 26–35, New York, NY, USA, 2004. ACM. ISBN 1-58113-842-3. URL `http://doi.acm.org/10.1145/976270.976276`.

[7] R. Hirschfeld. Aspects - aspect-oriented programming with squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, volume 2591 of *Lecture Notes in Computer Science*, pages 216–232. Springer Berlin / Heidelberg, 2003. URL `http://dx.doi.org/10.1007/3-540-36557-5_17`.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. In J. Knudsen, editor, *ECOOP 2001 - Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–354. Springer Berlin / Heidelberg, 2001. URL `http://dx.doi.org/10.1007/3-540-45337-7_18`.

[9] H. Rajan and K. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 59 – 68, may 2005. URL `http://dx.doi.org/10.1109/ICSE.2005.1553548`.

[10] L. Renggli, S. Ducasse, T. Gîrba, and O. Nierstrasz. Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, Malaga, Spain, June 2010.

[11] E. Tanter. Execution levels for aspect-oriented programming. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, pages 37–48, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-958-9. URL `http://doi.acm.org/10.1145/1739230.1739236`.

[12] É. Tanter, N. Tabareau, and R. Douence. Exploring membranes for controlling aspects. Technical Report TR/DCC-2011-8, University of Chile, June 2011.

Talents: Dynamically Composable Units of Reuse

Jorge Ressia, Tudor Gîrba, Oscar Nierstrasz, Fabrizio Perin, Lukas Renggli

Software Composition Group, University of Bern, Switzerland

http://scg.unibe.ch/

Abstract

Reuse in object-oriented languages typically focuses on inheritance. Numerous techniques have been developed to provide finer-grained reuse of methods, such as flavors, mixins and traits. These techniques, however, only deal with reuse at the level of classes.

Class-based reuse is inherently static. Increasing use of reflection and meta-programming techniques in real world applications underline the need for more dynamic approaches. New approaches have shifted to object-specific reuse. However, these techniques fail to provide a complete solution to the composition issues arising during reuse.

We propose a new approach that deals with reuse at the object level and that supports behavioral composition. We introduce a new abstraction called a *talent* which models features that are shared between objects of different class hierarchies. Talents provide a composition mechanism that is as flexible as that of traits but which is dynamic.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.3.3 [*Language Constructs and Features*]: Classes and Objects

General Terms Design, Languages

Keywords Reflection, Traits, Mixins, Object-specific behavior, Object adaption, Smalltalk

1. Introduction

Classes in object-oriented languages define the behavior of their instances. Inheritance is the principle mechanism for sharing common features between classes. Single inheritance is not expressive enough to model common features shared by classes in a complex hierarchy. Due to this several forms of multiple inheritance have been proposed [3, 21, 31, 39, 43]. However, multiple inheritance introduces problems that are difficult to resolve [12, 44]. One can argue that these problems arise due to the conflict between the two separate roles of a class, namely that of serving as a factory for instances, as well as serving as a repository for shared behaviour for all instances. As a consequence, finer-grained reuse mechanisms, such as flavors [32] and mixins [5], were introduced to compose classes from various features.

Although mixins succeed in offering a separate mechanism for reuse they must be composed linearly, thus introducing new difficulties in resolving conflicts at composition time. Traits [14, 40] overcome some of these limitations by eliminating the need for linear ordering. Instead dedicated operators are used to resolve conflicts. Nevertheless, both mixins and traits are inherently static, since they can only be used to define new classes.

Ruby [28] relaxes this limitation by allowing mixins to be applied to individual objects. Object-specific mixins however still suffer from the same compositional limitations of class-based mixins, since they must still be applied linearly to resolve conflicts.

In this paper we introduce *talents*, object-specific units of reuse which model features that an object can acquire at run-time. Like a trait, a talent represents a set of methods that constitute part of the behavior of an object. Unlike traits, talents can be acquired (or lost) dynamically. When a talent is applied to an object, no other instance of the object's class are affected. Talents may be composed of other talents, however, as with traits, the composition order is irrelevant. Conflicts must be explicitly resolved.

Like traits, talents can be flattened, either by incorporating the talent into an existing class, or by introducing a new class with the new methods. However, flattening is purely static and results in the loss of the dynamic description of the talent on the object. Flattening is not mandatory, on the contrary, it is just a convenience feature which shows how traits are a subset of talents.

The contributions of this paper are:

- We identify static problems associated with multiple inheritance, mixins and traits.
- We introduce *talents*, an object-specific behavior composition model that removes the limitations of static approaches.

- We describe a Smalltalk prototype of our approach.
- We describe two flattening techniques which merge the behavior adaptations into the original class hierarchy or into a new class.

Outline. In Section 2 we motivate the problem. Section 3 explains the talent approach, its composition operations and a solution to the motivating problem. In Section 4 we present the internal implementation of our solution in the context of Smalltalk. In Section 5 we discuss related work. Section 6 discusses about features of talents such as state sharing, scoping and flattening. In Section 7 we present examples to illustrate the various uses of talents. Section 8 summarizes the paper and discusses future work.

2. Motivating Example

Moose is a platform for software and data analysis that provides facilities to model, query, visualize and interact with data [19, 33]. Moose represents the source code in a model described by FAMIX, a language-independent meta-model [45]. The model of a given software system consists of entities representing various software artifacts such as methods (through instances of `FAMIXMethod`) or classes (through instances of `FAMIXClass`).

Each type of entity offers a set of dedicated analysis actions. For example, a `FAMIXClass` offers the possibility of visualizing its internal structure, and a `FAMIXMethod` offers the ability to browse its source code.

Moose can model applications written in different programming languages, including Smalltalk, Java, and C++. These models are built with the language independent FAMIX meta-model. However, each language has its own particularities which are introduced as methods in the different entities of the meta-model. There are different extensions which model these particularities for each language. For example, the Java extension adds the method `namespace` to the `FAMIXClass`, while the Smalltalk extension adds the method `isExtended`. Smalltalk however does not support namespaces, and Java does not support extended classes. Additionally, to identify test classes Java and Smalltalk require different implementations of the method `isTestClass` in `FAMIXClass`.

Another problem with the extensions for particular languages is that the user has to deal with classes that have far more methods than the model instances actually support. Dealing with unused code reduces the developer productivity and it is error prone.

A possible solution is to create subclasses for each supported language. However, there are some situation in which the model requires a combination of extensions: Moose JEE [35, 36] — a Moose extension to analyze Java Enterprise Applications (JEAs) — requires a combination of Java and Enterprise Application specific extensions. This leads to an impractical explosion of the number of subclasses. Moreover, possible combinations are hard to predict in advance.

Multiple inheritance can be used to compose the different behaviors a particular Moose entity requires. However, the diamond problem makes it difficult to handle the situation where two languages want to add a method of the same name. Mixins address the composition problem by applying a composition order, this however might lead to fragile code and subtle bugs. Traits offer a solution that is neutral to composition order, but traits neither solve the problem of the explosion in the number of classes to be defined, nor do they address the problem of dynamically selecting the behavior. Traits are composed statically into classes before instances can benefit from them.

We need a mechanism capable of dynamically composing various behaviors for different Moose entities. We should be able to add, remove, and change methods. This new Moose entity definition should not interfere with the behavior of other entities in other models used concurrently. We would like to be able to have coexisting models of different languages, formed by Moose entities with specialized behavior.

3. Talents in a Nutshell

In this section we present our approach. We propose composable units of behavior for objects, called *talents*. These abstractions solve the issues present in other approaches.

The prototype of talents[1] and the examples presented in this paper are implemented in Pharo Smalltalk[2], an open-source Smalltalk [20] implementation.

3.1 Defining Talents

A talent specifies a set of methods which may be added to, or removed from, the behavior of an object. Although the methods of a talent may directly access the state of an object, it is recommended to use accessor methods instead.

We will illustrate the use of talents with the Moose extension example introduced in the previous section.

A talent is an object that specifies methods that can be added to an existing object. A talent can be assigned to any object in the system to add or remove behavior.

```
1 aTalent := Talent new.
2 aTalent
3    defineMethod: #isTestClass
4    do: '^ self inheritsFromClassNamed: #TestCase'.
5 aClass := FAMIXClass new.
6 aClass acquire: aTalent.
```

We can observe that first a generic talent is instantiated and then a method is defined. The method `isTestClass` is used to test if a class inherits from `TestCase`. In lines 5–6 we can see that a FAMIX class is instantiated acquiring the previous talent. When the method `acquire:` is called, the object — in this case the FAMIX class — is adapted. Only this `FAMIXClass` instance is affected, no other instance

[1] http://scg.unibe.ch/research/talents/

[2] http://www.pharo-project.org/

is modified by the talent. No adaptation will be triggered if an object tries to acquire the same talent several times.

Talents can also remove methods from the object that acquires them.

```
1 aTalent := Talent new.
2 aTalent excludeMethod: #clientClasses.
3 aClass := FAMIXClass new.
4 aClass acquire: aTalent.
```

In this case the existing method `clientClasses` is removed from this particular class instance. Sending this message will now trigger the standard `doesNotUnderstand:` error of Smalltalk.

3.2 Composing Objects from Talents

Talent composition order is irrelevant, so conflicting talent methods must be explicitly disambiguated. Contrary to traits, the talent definition of a method takes precedence if the object acquiring the talent already has the same method. Once an object is bound to a talent then it is clear that this object needs to specialize its behavior. This precedence can be overridden if it is explicitly stated during the composition by removing the definition of the methods from the talent.

In the next example we will compose a group with two talents. One expresses the fact that a Java class is in a namespace, the other that a JEE class is a test class.

```
1  javaClassTalent := Talent new.
2  javaClassTalent
3     defineMethod: #namespace
4     do: '^ self owningScope'.
5  jeeClassTalent := Talent new.
6  jeeClassTalent
7     defineMethod: #isTestClass
8     do: '^ self methods anySatisfy: [ :each | each
        isTestMethod ]'.
9  aClass := FAMIXClass new.
10 aClass acquire: javaClassTalent , jeeClassTalent.
```

In line 10 we can observe that the composition of talents is achieved by sending the comma message (`,`). The composed talents will allow the FAMIX class instance to dynamically reuse the behavior expressed in both talents.

3.3 Conflict Resolution

A conflict arises if and only if two talents being composed provide different implementations for the same method. Conflicting talents cannot be composed, so the conflict has to be resolved to enable the composition.

To gain access to the different implementations of conflicting methods, talents support an alias operation. An alias makes a conflicting talent method available by using another name.

Talent composition also supports exclusion, which allows one to avoid a conflict before it occurs. The composition clause allows the user to exclude methods from a talent when it is composed. This suppresses these methods and allows the composite entity to acquire the otherwise conflicting implementation provided by another talent.

We would like models originating from JEE applications to support both Java and JEE extensions. Composing these two talents however generates a conflict for the methods `isTestClass` for a FAMIX class entity. The next example produces a conflict on line 10 since both talents define a different implementation of the `isTestClass` method.

```
1  javaClassTalent := Talent new.
2  javaClassTalent
3     defineMethod: #isTestClass
4     do: '^ self methods anySatisfy: [ :m | m
        isAnnotatedWith: #Test ]'.
5  jeeClassTalent := Talent new.
6  jeeClassTalent
7     defineMethod: #isTestClass
8     do: '^ self inheritsFrom: #TestCase'.
9  aClass := FAMIXClass new.
10 aClass acquire: javaClassTalent , jeeClassTalent.
```

There are different ways to resolve this situation. The first is to define aliases, like in traits, to avoid the name collision:

```
10 aClass acquire: javaClassTalent , (jeeClassTalent @ {
       #isTestClass -> #isJEETestClass}).
```

When the talent is acquired the method `isJEETestClass` is installed instead of `isTestClass`, thus avoiding the conflict. Any other method or another talent can then make use of this aliasing.

Another option is to remove those methods that do not make sense for the specific object being adapted.

```
10 aClass acquire: javaClassTalent , (jeeClassTalent -
       #isTestClass).
```

By removing the definition of the JEE class talent the Java class talent method is correctly composed.

Each FAMIX extension can be defined as a set of talents, each for a single entity, *i.e.,* class, method, annotation, *etc.* For example, we have the Java class talent which models the methods required by the Java extension to FAMIX class entity. We also have a Smalltalk class talent as well as a JEE talent that model further extensions.

4. Implementation

In this section we describe how talents are implemented.

4.1 Bifröst

Talents are built on top of the Bifröst reflection framework [38]. Bifröst offers fine-grained unanticipated dynamic structural and behavioral reflection through meta-objects. Instead of providing reflective capabilities as an external mechanism we integrate them deeply into the environment. Explicit meta-objects allow us to provide a range of reflective features and thereby evolve both application models and the host language at run-time. Meta-objects provide a sound basis for building different coexisting *meta-level architectures* by bringing traditional object-oriented techniques to the meta-level.

In recent years researchers have worked on the idea of applying traditional object-oriented techniques to the meta-level while attempting to solve various practical problems motivated by applications [29]. These approaches, however, offer specialized solutions arising from the perspective of particular use cases.

The Bifröst model solves the main problems of previous approaches while providing the main reflection requirements.

Partial Reflection. Bifröst allows meta-objects to be bound to any object in the system thus reflecting selected parts of an application.

Selective Reification. When and where a particular reification should be reified is managed by the different meta-objects.

Unanticipated Changes. At any point in time a meta-object can be bound to any object thus supporting unanticipated changes.

Meta-level Composition. Composable meta-objects provide the mean for bringing together different adaptations.

Runtime Integration. Bifröst's reflective model lives entirely in the language model, so there is no VM modification or low level adaptation required.

4.2 Talents

Figure 1 shows the normal message send of `isTestClass` to an instance of `FAMIXClass`. The method lookup starts on the class finding the definition of the method and then executing it for the message receiver.

However, if we would like to factor the `FAMIXClass` JEE behavior out we can define a talent that models this. Each talent is modeled with a structural meta-object. A structural meta-object abstraction provides the means to define meta-objects like classes and prototypes. New structural abstractions can be defined to fulfill some specific requirement. These meta-object responsibilities are: adding and removing methods, and adding and removing state to an object. A composed meta-object is used to model composed talents. The specific behavior for defining and removing methods is delegated to the addition and removal of behavior in the structural meta-object.

In Figure 2 we can observe the object diagram for a FAMIX class which has acquired a talent that models JEE behavior. The method lookup starts in the class of the receiver. Originally, the `FAMIXClass` class did not define a method `isTestClass`, however, the application of the talent defined this method. This method is responsible for delegating the execution of the message to the receiver's talent. If the object does not have a talent, the normal method lookup is executed, thus talents do not affect other instances' behavior of the class. In this case, `aFAMIXClass` has a talent that defines the method `isTestClass`, which is executed for the message receiver.

5. Related Work

In this section we compare talents to other approaches to share behavior.

Mixins

Flavors [32] was the first attempt to address the problem of reuse across a class hierarchy. Flavors are small, incomplete implementations of classes, that can be "mixed in" at arbitrary places in the class hierarchy. More sophisticated notions of mixins were subsequently developed by Bracha and Cook [5], Mens and van Limberghen [30], Flatt, Krishnamurthi and Felleisen [16], and Ancona, Lagorio and Zucca [1].

Mixins present drawbacks when dealing with composition. Mixins use single inheritance for composing features and extending classes. Mixins have to be composed linearly thus limiting the ability to define the glue code necessary to avoid conflicts. However, although this inheritance operator is well-suited for deriving new classes from existing ones, it is not appropriate for composing reusable building blocks.

Bracha developed Jigsaw [4], a modularity framework which defines module composition operators merge, override, copy as and restrict. These operators inspired the sum, override, alias and exclusion operators on traits. Jigsaw models a complete framework for module manipulation providing namespaces, declared types and requirements, full renaming, and semantically meaningful nesting.

Ruby [28] introduced mixins as a building block of reusability, called modules. Moreover, modules can be applied to specific objects without modifying other instances of the class. However, object-specific modules suffer from the same composition limitation as modules applied to classes: they have to be applied linearly. Aliasing of methods is possible for avoiding name collisions, as well as removing method in the target object. However, objects or classes methods cannot be removed if they are not already implemented. This follows the concept of linearization of mixins. Talents can be applied without an order. Moreover, a talent composition delivers a new talent that can be reused and applied to other objects. Filters in Ruby provide a mechanism for composing behavior into preexisting methods. However, they do not provide support for defining how modules defined methods should be composed for a single object.

CLOS

CLOS [10] is an object-oriented extension of Lisp. Multiple inheritance in CLOS [25, 34] imposes a linear order on the superclasses. This linearization often leads to unexpected behavior because it is not always clear how a complex multiple inheritance hierarchy should be linearized [15]. CLOS also provides a mechanism for modifying the behavior of specific instances by changing the class of an instance using the generic function `change-class`. However, these modifications do not provide any composition mechanisms, render-

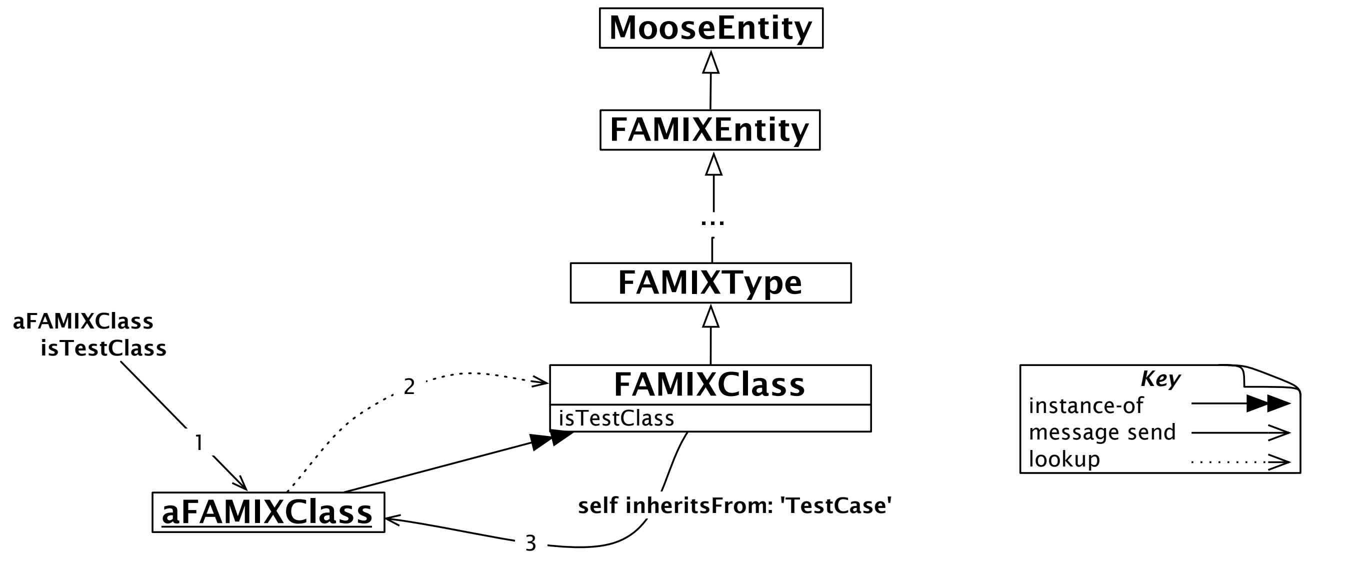


Figure 1. Default message send and method look up resolution.

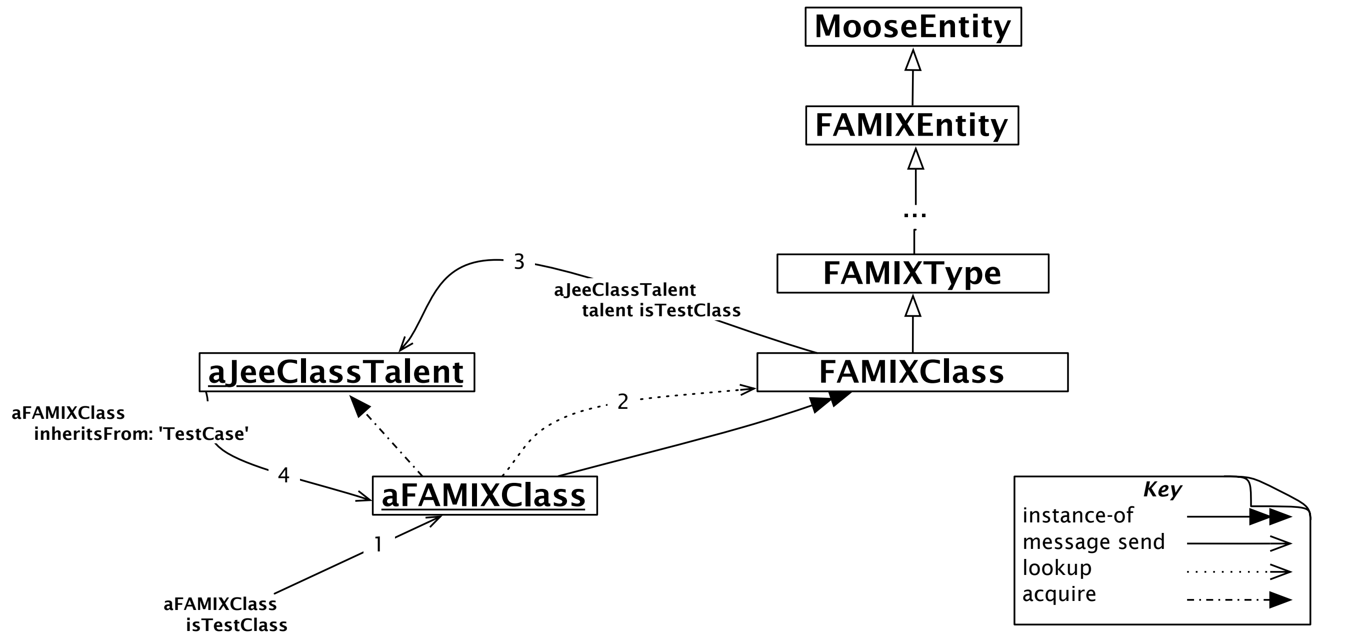


Figure 2. Talent modeling the Moose FAMIX class behavior for the method `isTestClass`.

ing this technique dependent on custom code provided by the user.

Traits

Traits [14, 40] overcome the limitations of previous approaches. A trait is a set of methods that can be reused by different classes. The main advantage of traits is that their composition does not depend on a linear ordering. Traits are composed using a set of operators — symmetric combination, exclusion, and aliasing — allowing a fair amount of composition flexibility. Traits are purely static since their semantics specify that traits can always be "flattened" to an equivalent class hierarchy without traits, but possibly with duplicated code. As a consequence traits can neither be added nor removed at run-time. Moreover, traits were not conceived to model object-specific behavior reuse.

Smith and Drossopoulou [41] proposed a mechanism for applying traits at runtime in the context of Java. However, only pre-defined behavior defined in a trait can be added at runtime. It is not possible to define and add new behavior at runtime.

Object Extensions

Self [46] is a prototype-based language which follows the concepts introduced by Lieberman [26]. In Self there is no notion of class; each object conceptually defines its own format, methods, and inheritance relations. Objects are derived from other objects by cloning and modification. Objects can have one or more prototypes, and any object can be the prototype of any other object. If the method for a message send is not found in the receiving object then it is delegated to the parent of that object. In addition, Self also has the notion of trait objects that serve as repositories for sharing behavior and state among multiple objects. One or more trait objects can be dynamically selected as the parent(s) of any object. Selector lookups unresolved in the child are passed to the parents; it is an error for a selector to be found in more than one parent. Self traits do not provide a mechanism to fine tune the method composition. Let us assume that two objects are dynamically defined as parents of an object. If the both parents object define the same method there is not a simple way of managing the conflict.

Object extension [11, 18] provides a mechanism for self-inflicted object changes. Since there is no template serving as the object's class, only the object's methods can access the newly introduced method or data members. Ghelli *et al.* [18] suggested a calculus in which conflicting changes cannot occur, by letting the same object assume different roles in different contexts.

Drossopoulou proposed Fickle [13], a language for dynamic object re-classification. Re-classification changes at run-time the class membership of an object while retaining its identity. This approach proposes language features for object re-classification to extend an imperative, typed, class-based, object-oriented language. Even though objects may be re-classified across classes with different members, they will never attempt to access non-existing members.

Cohen and Gil introduced the concept of object evolution [7]. This approach proposes three variants of evolution, relying on inheritance, mixins and shakeins [37]. The authors introduce the notion of evolvers, a mechanism for maintaining class invariants in the course of reclassification [13]. This approach is oriented towards dynamic reuse in languages with types. Shakeins provide a type-free abstraction, however, there are no composition operators to aid the developer in solving more complex scenarios.

Bracha *et al.* [6] proposed a new implementation of nested classes for managing modularity in Newspeak. Newspeak is class-based language with virtual classes. Class references are dynamically determined at runtime; all names in Newspeak are method invocations thus all classes are virtual. Nested classes were first introduced in Beta [27]. Classes declarations can be nested to an arbitrarily depth. Since all references to names are treated as method invocations any object member declaration can be overridden. The references in an object to nested classes are going to be solved when these classes are late bound to the classes definition in the active module the object it is in. Talents model a similar abstraction to modules, for dynamically composing the behavior of objects. However, Newspeak modules do not provide composition operators similar to talents. Composed talents can remove, alias, or override method definitions. Removing method definitions is not a feature provided by Newspeak modules. In Newspeak composition would be done in the module or in the nested classes explicitly.

Subjective Programming

Subjective behavior is essential for applications that must adapt their behavior to changing circumstances. Many different solutions have been proposed in the past, based, for example, on perspectives [42], roles [24], contextual layers [8], and force trees [9]. Depending on the active context, an object might answer a message differently or provide a modified interface to its users. These approaches mainly concentrate on dynamically modifying an object's behavior, however, there is no support for behavior reuse between object as it exists in traits or mixins.

Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [22] modularizes cross cutting concerns. Join points define all locations in a program that can possibly trigger the execution of additional cross-cutting code (advice). Pointcuts define at run-time if an advice is executed. Both aspects and talents can add new methods to existing classes. Most implementations of AOP such as AspectJ [23] support weaving code at more fine-grained join points such as field accesses, which is not supported by talents. Although AOP is used to introduce changes into software systems, the focus is on cross-cutting concerns, rather than on reflecting on the system.

Aspects are concerns that cannot be cleanly encapsulated in a generalized abstraction (i.e., object, method, mixin). This means that in contrast to talents, aspects are neither designed nor used to build dynamic abstraction and components from scratch, but rather to alter the performance or semantics of the components in systemic ways.

6. Discussion

In this section we discuss other benefits that talents bring to Smalltalk.

6.1 State

Bifröst structural meta-objects provide features for adding and removing state from a single object. Theoretically, talents can provide something that trait cannot, state. Moreover, talents can provide operators for composing state adaptations. This composition is not present in object-specific techniques like mixins and Newspeak modules.

However, stateful traits [2] have shown that state composition is not simple to achieve.

6.2 Scoping

Scoping talents dynamically is of key importance because it allows us to reflect in which context the added features should be active and also to control the extent of the system that is modified. An object might require to have certain features in one context while having others features in a different context. Let us analyze an example to understand the motivation for scoping talents.

A bank financial system is divided in two main layers: the domain and the persistency layer. The domain layer models the financial system requirements and features. The persistency layer deals with the requirements of persisting the domain abstraction in a relational database. When testing the domain behavior of this application we do not want to trigger database-related behavior. Normally, this is solved through mocking or dependency injection . However, these solutions are not simple to implement in large and legacy systems which are not fully understood, and where any change can bring undesired side effects. Scoped talents can solve this situation by defining a scope around the test cases. When the tests are executed the database access objects are modified by a talent which mocks the execution of database related actions. In a highly-available system which cannot be stopped, like a financial trading operation, scoped talents can help in actions like: auditing for the central financial authority, introducing lazy persistency for updating the database, logging. This is similar to the idea of modules in Newspeak.

6.3 Flattening

Flattening is the technique that folds into a class all the behavior that has been added to an object. There are two types of flattening in talents:

Flattening on the original class. Once an object has been composed with multiple talents it has a particular behavior. The developer can analyze this added behavior and from a modeling point of view realize that all instances of the object's class should have these changes. This kind of flattening applies the talent composition to the object's class.

Flattening on a new class. On the other hand the developer might realize that the new responsibilities of the object is relevant enough to be modeled with a separate abstraction. Thus a new class has to be created cloning the composed object behavior. This new class will inherit from the previous object class. Deleted methods will be added with a `shouldNotCallMethod` exception to avoid inheriting the implementation.

7. Examples

In this section we present a number of example applications of talents.

7.1 Mocking

Let us assume that we need to test a class which models a solvency analysis of the assets of a financial institution customer. The method we need to test is `SolvencyAnalysis>>isSolvent: aCustomer`. This method delegates to `SolvencyAnalysis>>assetsOf: aCustomer` which executes a complex calculation of the various assets and portfolios of the customer.

We are only interested in isolating the behavior of `isSolvent:` , we are not interested in the complexities of `assetsOf:`

```
1 SolvencyAnalysisTest>>testIsSolvent
2    | aCustomer anAnalysis |
3    aCustomer := Customer named: 'test'.
4    anAnalysis := SolvencyAnalysis new.
5    anAnalysis method: #assetsOf: shouldReturn: 1.
6    self assert: (anAnalysis isSolvent: aCustomer).
7    anAnalysis method: #assetsOf: shouldReturn: -1.
8    self deny: (anAnalysis isSolvent: aCustomer).
```

We added the method `method:shouldReturn:` to the class `Object` which creates a talent with a method named as the first argument and with the body provided by the second argument. In line 5 and 7 you can see the use of this behavior. If the method `assetsOf:` return a positive amount then the customer is solvent otherwise not.

7.2 Compiler examples

Cohen and Gil provide an example in the context of object evolution [7]. In many compiler designs, the parser generates an Abstract Syntax Tree (AST) from the source code; the back-end then processes this tree. Often, the parser does not have the knowledge required for classifying a given AST node at its most refined representation level. For example, in the Smalltalk compiler's parser a variable access is modeled as an `ASTVariableNode`, there is no distinction between instance variables, class variables, globals or temporals. As the compiler advances through its phases these AST nodes are going to be classified in an abstraction called the lexical scope tree. However, when analyzing the AST structure we require information about types of variables and scopes. Using talents we can add behavior to AST variable nodes to specify them as instance variable, class variables and temporals. We can also remove methods from AST nodes that do not make sense for the new specification of the node. The talent composition mechanism will be particularly useful in merging these different talents on AST nodes.

7.3 State Pattern

The state pattern [17] models the different states a domain object might have. When this object needs to do something then it delegates the decision of what to do to its state. A class per object state is created with the required behavior. Sometimes, multiple instances of each state are created and sometimes a singleton pattern is used.

Instead of having a state abstract class and then concrete subclasses for each of the more specific states we could use talents. We will have a single state class and then create as many instances as different states are. We can model each specific state with a different talent that is applied to the state's instances, thus avoiding the creation of multiple state specific classes.

8. Conclusion and Future Work

This paper presented talents, a dynamic compositional model for reusing behavior. Talents are composed using a set of operations: composition, exclusion and aliasing. These operations provide a flexible composition mechanism while avoiding the problems found in mixins and traits.

Talents are most useful in composing behavior for different extensions that have to be applied to the same base classes, thus dynamically adapting the behavior of the instances of these classes seems natural to obtaining a different protocol.

Managing talents can currently be complicated since the development tools are unaware of them. We plan on developing a user interface which takes talents into account both helping in their definition and composition.

We plan on providing a more mature implementation of the talents scoping facilities. This technique shows great potential for the requirements of modern applications, such as dynamic adaptation and dependency injection for testing, database accesses, profiling, and so on.

Acknowledgments

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). We also like to thank Simon Denier for his feedback on earlier drafts of this paper.

References

[1] D. Ancona, G. Lagorio, and E. Zucca. Jam — a smooth extension of Java with mixins. In *ECOOP 2000*, number 1850 in Lecture Notes in Computer Science, pages 145–178, 2000.

[2] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful traits and their formalization. *Journal of Computer Languages, Systems and Structures*, 34(2-3):83–108, 2008. ISSN 1477-8424. doi: 10.1016/j.cl.2007.05.003.

[3] A. H. Borning and D. H. Ingalls. Multiple inheritance in Smalltalk-80. In *Proceedings at the National Conference on AI*, pages 234–237, Pittsburgh, PA, 1982.

[4] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Dept. of Computer Science, University of Utah, Mar. 1992.

[5] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, volume 25, pages 303–311, Oct. 1990.

[6] G. Bracha, P. von der Ahé, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda. Modules as objects in Newspeak. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 405–428, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14106-4, 978-3-642-14106-5. doi: 10.1007/978-3-642-14107-2_20.

[7] T. Cohen and J. Y. Gil. Three approaches to object evolution. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 57–66, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-598-7. doi: 10.1145/1596655.1596665.

[8] P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming: An overview of ContextL. In *Proceedings of the Dynamic Languages Symposium (DLS) '05, co-organized with OOPSLA'05*, pages 1–10, New York, NY, USA, Oct. 2005. ACM. ISBN 1-59593-283-6. doi: 10.1145/1146841.1146842.

[9] B. Darderes and M. Prieto. Subjective behavior: a general dynamic method dispatch. In *OOPSLA Workshop on Revival of Dynamic Languages*, Oct. 2004.

[10] L. G. DeMichiel and R. P. Gabriel. The Common Lisp object system: An overview. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings ECOOP '87*, volume 276 of *LNCS*, pages 151–170, Paris, France, June 1987. Springer-Verlag.

[11] P. Di Gianantonio, F. Honsell, and L. Liquori. A lambda calculus of objects with self-inflicted extension. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 166–178, New York, NY, USA, 1998. ACM. ISBN 1-58113-005-8. doi: 10.1145/286936.286955.

[12] R. Dixon, T. McKee, M. Vaughan, and P. Schweizer. A fast method dispatcher for compiled languages with multiple inheritance. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 211–214, Oct. 1989.

[13] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic object re-classification. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 130–149, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-42206-4.

[14] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, Mar. 2006. ISSN 0164-0925. doi: 10.1145/1119479.1119483.

[15] R. Ducournau, M. Habib, M. Huchard, and M. Mugnier. Monotonic conflict resolution mechanisms for inheritance. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, volume 27, pages 16–24, Oct. 1992.

[16] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183. ACM Press, 1998. ISBN 0-89791-979-3. doi: 10.1145/268946.268961.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, Reading, Mass., 1995. ISBN 978-0201633610.

[18] G. Ghelli. Foundations for extensible objects with roles. *Inf. Comput.*, 175(1):50–75, 2002.

[19] T. Gîrba. *The Moose Book*. Self Published, 2010.

[20] A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983. ISBN 0-201-13688-0.

[21] S. E. Keene. *Object-Oriented Programming in Common-Lisp*. Addison Wesley, 1989.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag. doi: 10.1007/BFb0053381.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings ECOOP 2001*, number 2072 in LNCS, pages 327–353. Springer Verlag, 2001.

[24] B. B. Kristensen. Object-oriented modeling with roles. In J. Murphy and B. Stone, editors, *Proceedings of the 2nd International Conference on Object-Oriented Information Systems*, pages 57–71. Springer-Verlag, 1995.

[25] J. A. Lawless and M. M. Milner. *Understanding Clos the Common Lisp Object System*. Digital Press, 1989.

[26] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 214–223, Nov. 1986. doi: 10.1145/960112.28718.

[27] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison Wesley, Reading, Mass., 1993. ISBN 0-201-62430-3.

[28] Y. Matsumoto. *Ruby in a Nutshell*. O'Reilly, 2001. ISBN 0596002149.

[29] J. McAffer. Engineering the meta level. In G. Kiczales, editor, *Proceedings of the 1st International Conference on Metalevel Architectures and Reflection (Reflection 96)*, San Francisco, USA, Apr. 1996.

[30] T. Mens and M. van Limberghen. Encapsulation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems. *Object Oriented Systems*, 3(1):1–30, 1996.

[31] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

[32] D. A. Moon. Object-oriented programming with Flavors. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 1–8, Nov. 1986.

[33] O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York NY, 2005. ACM Press. ISBN 1-59593-014-0. doi: 10.1145/1095430.1081707. Invited paper.

[34] A. Paepcke. User-level language crafting. In *Object-Oriented Programming: the CLOS perspective*, pages 66–99. MIT Press, 1993.

[35] F. Perin. MooseJEE: A Moose extension to enable the assessment of JEAs. In *Proceedings of the 26th International Conference on Software Maintenance (ICSM 2010) (Tool Demonstration)*, Sept. 2010. doi: 10.1109/ICSM.2010.5609569.

[36] F. Perin, T. Gîrba, and O. Nierstrasz. Recovery and analysis of transaction scope from scattered information in Java enterprise applications. In *Proceedings of International Conference on Software Maintenance 2010*, Sept. 2010. doi: 10.1109/ICSM.2010.5609572.

[37] A. Rashid and M. Aksit, editors. *Transactions on Aspect-Oriented Software Development II*, volume 4242 of *Lecture Notes in Computer Science*, 2006. Springer. ISBN 3-540-48890-1.

[38] J. Ressia, L. Renggli, T. Gîrba, and O. Nierstrasz. Run-time evolution through explicit meta-objects. In *Proceedings of the 5th Workshop on Models@run.time at the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010)*, pages 37–48, Oct. 2010. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-641/.

[39] C. Schaffert, T. Cooper, B. Bullis, M. Killian, and C. Wilpolt. An Introduction to Trellis/Owl. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 9–16, Nov. 1986.

[40] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behavior. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003. ISBN 978-3-540-40531-3. doi: 10.1007/b11832.

[41] C. Smith and S. Drossopoulou. Chai: Typed traits in Java. In *Proceedings ECOOP 2005*, 2005.

[42] R. B. Smith and D. Ungar. A simple and unifying approach to subjective objects. *TAPOS special issue on Subjectivity in Object-Oriented Systems*, 2(3):161–178, 1996. doi: 10.1002/(SICI)1096-9942(1996)2:3%3C161::AID-TAPO3%3E3.0.CO;2-Z.

[43] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, Reading, Mass., 1986. ISBN 0-201-53992-6.

[44] P. F. Sweeney and J. Y. Gil. Space and time-efficient memory layout for multiple inheritance. In *Proceedings OOPSLA '99*, pages 256–275. ACM Press, 1999. ISBN 1-58113-238-7. doi: 10.1145/320384.320408.

[45] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings of International Symposium on Principles of Software Evolution (ISPSE '00)*, pages 157–167. IEEE Computer Society Press, 2000. doi: 10.1109/ISPSE.2000.913233.

[46] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 227–242, Dec. 1987. doi: 10.1145/38765.38828.

Towards Structural Decomposition of Reflection with Mirrors

Nick Papoulias[1,2] Noury Bouraqadi[2] Marcus Denker[1]
Stéphane Ducasse[1] Luc Fabresse[2]

[1]RMoD Project-Team, Inria Lille–Nord Europe / Université de Lille 1
[2]Université Lille Nord de France, Ecole des Mines de Douai
{nick.papoulias,noury.bouraqadi,luc.fabresse}@mines-douai.fr,
{marcus.denker,stephane.ducasse}@inria.fr

Abstract

Mirrors are meta-level entities introduced to decouple reflection from the base-level system. Current mirror-based systems focus on functional decomposition of reflection. In this paper we advocate that mirrors should also address structural decomposition. Mirrors should not only be the entry points of reflective behavior but also be the storage entities of meta-information. This decomposition can help resolve issues in terms of resource constraints (e.g. embedded systems and robotics) or security. Indeed, structural decomposition enables discarding meta-information.

1. Introduction

Reflective Object Oriented languages provide a conceptual and design advantage due to their dynamic nature. This advantage often comes at the price of resource intensive solutions for our computational problems. In areas where there are resource constraints, reflective languages aid the design process, but their reflective nature is a burden upon deployment. Moreover security is an issue in the presence of reflective facilities and meta-information.

Mirrors [6] first introduced in the language Self [18] try to remedy this situation. Mirrors provide the decoupling of reflection from the base system through functional decomposition [3]. This means that reflective methods on objects are migrated to separate entities (called Mirrors) that conform to a specific behavior. This behavior can be described via interfaces or abstract classes, allowing different implementations. Thanks to this functional decomposition, it is possible to later discard the reflective behavior of a language.

On the contrary in classical implementations of reflection (as in Smalltalk) there is no clear specification as to where and how meta-information and reflective functionality is stored, can be accesed, and how exactly (semantically or otherwise) it interacts with the base system.

A clear example of this problematic situation is given in [6] and it's generality easily encompasses nearly all reflective OO languages in current use today. To illustrate this we provide and discuss the same example for Smalltalk in Figure 1.

What the code-snippet in Figure 1 illustrates is that the meta-linguistic part of the system, specifically in this example the acquisition of class and superclass references, is implemented and accessed as ad-hoc functionality inside the base system of the language. *Base and Meta level functionality is thus indistinguishably mixed in program code.*

```
myCar := Car new.
carClass := myCar class.
anotherCar := carClass new.
carSuperclass := carClass superclass.
```

Figure 1. The initial example of Bracha and Ungar in Smalltalk

Mirror based reflection deviates from this perspective by proposing a specific design pattern for meta-level access that can exhibit certain desired characteristics such as:

Encapsulation The ability of mirrors to encapsulate the reflective behavior and meta-information of objects, allowing different implementations.

Stratification The ability of mirrors to functionally decompose reflection from the base system, allowing the meta-level to be discarded when not needed.

Ontological Correspondence The ability of mirrors to describe and reflect a language in its entirety. This is a derived property that depends solely on the completeness of the mirror implementation.

Although mirrors support multiple implementations through encapsulation, there always exists some original source of meta-information in reflective languages. This original source may well reside in the base system of the language, in abstract syntax trees, in separate source files, in system dictionaries, in the object representation or inside the virtual-machine.

Since mirrors describe only the functional decomposition of reflection there is no specification for the stratification of the meta-information itself. This fact impacts the level of stratification that mirrors can offer.

In this paper, we propose a solution to this problem by means of structural decomposition for reflection. Structural decomposition is achieved by extending mirrors to be the storage entities of meta-information. We also distinguish between the high-level language meta-information and low-level meta-information from the perspective of the VM. We argue that although low-level information is the limit of stratification, it can be clearly decomposed from the base object model of a language.

In Section 2 we discuss stratification of reflection in the context of resource and deployment constraints. In Section 3 we provide a reference model for structural decomposition of reflection. Finally in Section 4, we describe and validate a prototype for our proposal.

2. Discussion on Structural Decomposition

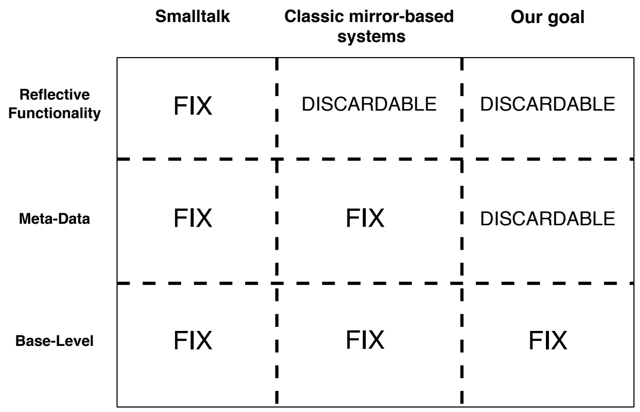
2.1 Functional and Structural Decomposition

Both in literature [6] and in state-of-the-art implementations of mirrors [4] the emphasis is given on the functional decomposition of reflection. This means that there is no specific description for the origin of meta-information or their storage point in the system. It follows that the stratification property of mirrors discussed in literature [6] does not apply to meta-data.

This is why the usual strategy for mirror instantiation is to collect the meta-information from the system as a whole (through primitives, separate source files or otherwise) [14]. Mirrors are not the original source of meta-information.

In existing mirror-based systems, one can only stratify the reflection functionality not the actual meta-data (see left part of Figure 2). This situation raises two issues. On the one hand, unused meta-data cannot be discarded to free system resources. On the other hand, undesirable access to this meta-data is possible since they still reside within the system. The impact of these facts should not be underestimated. Meta-information are widely used in reflective OO languages such as: global resolution names, names of instance variables and methods, system categories and packaging information, the code for each module, sub-module in the language, abstract syntax trees, executional, profiling and tracing information.

Structural decomposition of reflection through mirrors would help to extend the property of stratification to meta-data. Thus, one can effectively discard meta-data (see right part of Figure 2). As an analogy to this, in c-like languages one is able to discard not only debugging facilities but also meta-information [12, 17] upon deployment for efficiency reasons.



	Smalltalk	Classic mirror-based systems	Our goal
Reflective Functionality	FIX	DISCARDABLE	DISCARDABLE
Meta-Data	FIX	FIX	DISCARDABLE
Base-Level	FIX	FIX	FIX

Figure 2. Our goal, towards structural decomposition with mirrors

2.2 Low-level vs High-level Meta-information

Bracha and Ungar [6] make the distinction between low-level and high-level mirrors. We discuss this distinction in the light of structural decomposition. Although structural decomposition of all meta-information is desirable, not all meta-information should be discarded. The run-time engine does require some meta-information to actually run the base-level. We qualify such meta-information as low-level. This information usually includes: class reference, superclass reference and unique indexes for each method.

To sum up we believe that all meta-information should be decomposed and thus materialized in mirrors. Furthermore there should be a distinction as in [6] between low-level and high-level mirrors. Low-level mirrors cannot be discarded, since they encapsulate low-level meta-data. On the opposite all high-level meta-information should be materialized in high-level mirrors. Thus, it can be discarded.

3. Reference Model for the Structural Decomposition of Reflection

Aiming for simplicity and generality in our model, we chose to derive it from ObjVLisp [8]. We extended it by adding a meta-level with both an abstract and a concrete specification of mirrors. In this object model:

- Every object in the system has a Mirror, including classes and meta-classes.
- Mirrors are meta-level entities that hold all the meta-information of the respective object that they reflect on, and they are the sole provider of all reflective functionality on that object.
- All other entities in the language (including meta-classes) can provide reflective functionalities only explicitly through mirrors.

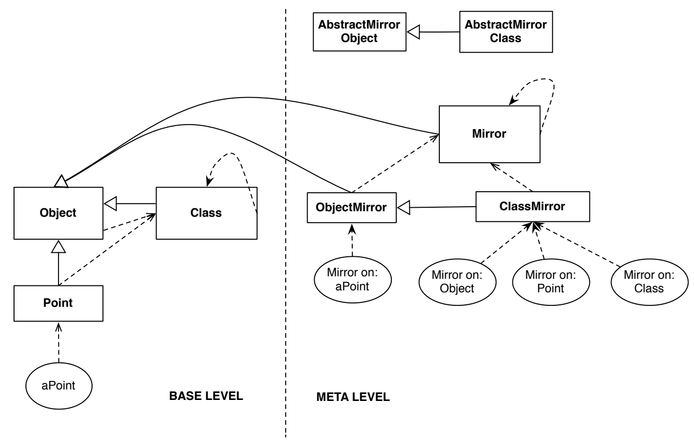


Figure 3. The MetaTalk Object Model

- Dynamic class definition can only be done through mirrors.

In our object model the base level is self-contained and can function independently. It can only instantiate terminal objects, without their meta-level counter parts. This allows the base system to operate seperately and effectively prevents dynamic addition of behavior in the absence of Mirrors.

All reflective functionality and meta-information is accessible through mirrors. This information also includes low-level meta-information which is part of an object's representation but cannot be accessed from the base level.

4. Implementation and Validation

4.1 Implementation

For validating the structural decomposition of meta-information in a mirror based reflection system we implemented the experimental class-based language MetaTalk. MetaTalk focuses on providing these characteristics to a dynamically typed OO language inspired by Smalltalk [10], Resilient (a Smalltalk descendant targeting embedded devices) [1] and ObjVLisp [8].

MetaTalk follows the guidelines of our reference object model described in Section 3. It was implemented from top to bottom (object-representation, compiler, and virtual-machine) in the open-source, smalltalk-inspired environment Pharo [2]. Our compiler relies on *PetitParser* [16]. Compilation by definition is taking place in the presence of meta-information. Only the meta-level has access to the compiler. Thus it can be discarded from the system in the absence of mirrors.The code for our open-source language prototype can be found in the SqueakSource repository[1].

[1] http://www.squeaksource.com/MetaTalk/

4.2 Validation

For the validation of structural decomposition one needs to test program execution both in the presence and in the absence of mirrors. Base level functionality should be semantically identical in both cases. On the contrary access to the meta level should only be possible in the presence of mirrors. If base level functionality is validated to be identical in both cases, this means that mirrors can be safely discarded when not needed. Moreover access to the meta-level should be validated to raise an error or an exception in the absence of mirrors.

Following this strategy for the validation of our prototype we took the following successive steps:

1. We compiled our kernel from sources, and validated its sound execution in the complete absence of the meta-level.
2. We then compiled our meta-kernel from sources, providing the system with its reflective functionality.
3. Subsequently we allowed global access to mirrors by invoking: *Baselevel reflect: true* , which is signaling the VM to permit mirrors to be pushed in the execution stack. From this point on and for the rest of the life of the system, no further compilation from sources can take place.
4. Then we dynamically created new classes, a superclass and a subclass through the meta-level (which by now is the only way to introduce new functionality to the system).
5. We tested the newly created classes for both their base and meta level functionality (via mirrors).
6. Subsequently we forbide global access to mirrors by invoking: *BaseLevel reflect: false.*
7. Finally we repeated step 5 of the process, verifying that: (a) base level functionality of the newly created classes was not by anyway altered by the absence of the meta-level, thus concluding that the meta-level could be safely discarded; (b) the subsequent attempt to access the meta-level signaled a terminal error by the vm.

Steps 4 through 7 are seen in Figures 4 and 5, respectively while the standard output generated in these steps can be found in Figure 6.

5. Related Work

The work of Bracha and Ungar in [6] discusses the functional decomposition of reflection via mirrors. We believe that we have succeeded in showing that structural decomposition is also essential for extending the property of stratification to meta-data.

The work of Lorenz and Vlissides [13] on pluggable-reflection, concerns reflection as a uniform interface in the presence of multiple sources of meta-information. It is re-

```
newClass := ((Mirror on: Object) subClass: 'PointX' instanceVariableNames: {'x' . 'y'.}) baseObject.

(Mirror on: newClass) atMethod: 'initialize' put: 'x := 0. y := 0.'.
(Mirror on: newClass) atMethod: 'x' put: '^ x.'.
(Mirror on: newClass) atMethod: 'y' put: '^ y.'.
(Mirror on: newClass) atMethod: 'x: aNumber' put: 'x := aNumber asNumber.'.
(Mirror on: newClass) atMethod: 'y: aNumber' put: 'y := aNumber asNumber.'.
(Mirror on: newClass) atMethod: 'asString' put: '^ x asString , ''@'' , y asString.'.

newSubClass := ((Mirror on: newClass) subClass: 'Point3DX' instanceVariableNames: 'z'.) baseObject.

(Mirror on: newSubClass) atMethod: 'initialize' put: 'super initialize. z := 0.'.
(Mirror on: newSubClass) atMethod: 'z' put: '^ z.'.
(Mirror on: newSubClass) atMethod: 'z: aNumber' put: 'z := aNumber.'.
(Mirror on: newSubClass) atMethod: 'asString' put: '^ super asString , ''@'' , z asString.'.
```

Figure 4. Step 4 of the validation process

```
'Validation process...' print.

'Base level functionality, in the presense of mirrors:' print.
p3D := newSubClass new.
p3D z: 30.
p3D x: 1.
p3D print.

'Meta level functionality, in the presense of mirrors:' print.
p3D := newSubClass new.
(Mirror on: p3D) perform: 'z:' withArguments: 30..
(Mirror on: p3D) perform: 'x:' withArguments: 1..
(Mirror on: p3D) print.

BaseLevel reflect: false.
'Base level functionality, in the absense of mirrors:' print.

p3D := newSubClass new.
p3D z: 30.
p3D x: 1.
p3D print.

'Meta level functionality, in the absense of mirrors -- should signal an error by the vm.' print.
p3D := newSubClass new.
(Mirror on: p3D) perform: 'z:' withArguments: 30..
(Mirror on: p3D) perform: 'x:' withArguments: 1..
(Mirror on: p3D) print.
```

Figure 5. Steps 5, 6 and 7 of the validation process.

lated to the encapsulation property of mirrors, allowing multiple implementations but not stratification. Their approach to the reflection interface is extralingual and does not concern the reflective system inside the language or it's decomposition from the base level.

A comprehensive comparison of mirror-based systems with other approaches for reflection in general, is given in [6].

Specifically for Smalltalk the language itself strongly couples the base and the meta-level. A declarative model for the definition of Smalltalk programs as is presented in [19]. can be used as a basis for the execution of Smalltalk programs that do not use reflection. In our work using mirrors and structural decomposition, we have shown that even in an imperative model for a language, reflection can be decoupled and stratified when it is not needed. Furthermore in our validation prototype the kernel is compiled from sources and can be used separately as a non reflective declarative system.

The Resilient platform [1] targeting embedded devices, offers remote reflection, by separating the programming from the running environment, and greatly succeeds in minimizing the deployment footprint. The reflection scheme though is not mirror based and cannot provide encapsulation. From the point of view of stratification and structural decomposition in such a scenario we believe that the deployment footprint can be reduced even more.

Implementations of mirror - based reflective sub-systems, that in general terms follow the premises of [6] already exist in languages as diverse as Java [11], Self [14] , StrongTalk [5], Newspeak [4] and AmbientTalk [15] with *on-going* smaller or larger efforts to implement them for C++ [7], Scala [9], Javascript [20] and possibly other platforms.

From the point of view of structural decomposition, and although our focus is on dynamic OO languages, the ongoing effort of the C++ community for mirror - based reflection, presents some interest. The static nature of the language suggests that structural decomposition in these languages should indeed be possible.

But, it is exactly this nature of the language that forces the implementation of reflective facilities to resort to adhoc mechanisms for generating the meta-data. In the case of C++ these come in the form of explicit registration macros for each and every name-space, type or class defined. Hinting that again structural decomposition and stratification of the reflective facilities could come only in the form of recompilation without the supporting libraries and macros.

6. Conclusion and Future Work

We have raised the question of structural decomposition of reflection and meta-information, in the context of mirror-

```
MetaTalk>>> Validation process...

MetaTalk>>> Base level functionality, in the presense of mirrors:
MetaTalk>>> 1@0@30

MetaTalk>>> Meta level functionality, in the presense of mirrors:
MetaTalk>>> Mirror on: a Point3DX 1@0@30

MetaTalk>>> Base level functionality, in the absense of mirrors:
MetaTalk>>> 1@0@30

MetaTalk>>> Meta level functionality, in the absense of mirrors -
- should signal an error by the vm.
...
<'metatalk-vm-exception: meta level access is disabled'>
```

Figure 6. Standard output generated from steps 5 to 7 of the validation process.

based systems. We showed that the property of stratification for mirrors, can be weak if structural decomposition is not taken into account. We provided a solution with a reference model where mirrors are the initial source of meta-information. Finally we validated this solution through a prototype supporting both functional and structural decomposition of reflection.

In terms of future work, we would like to provide further validation and metrics for structural decomposition. We want to give a more detailed specification of the system for implementors. Furthermore we would like to advance our prototype towards ontological correspondence and examine the role of structural decomposition in the context of behavioral reflection [15].

References

[1] Jakob R. Andersen, Lars Bak, Steffen Grarup, Kasper V. Lund, Toke Eskildsen, Klaus Marius Hansen, and Mads Torgersen. Design, implementation, and evaluation of the resilient smalltalk embedded platform. In *Proceedings of ESUG International Smalltalk Conference 2004*, September 2004.

[2] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[3] Gilad Bracha. Linguistic reflection via mirrors (talk at hpi potsdam). http://bracha.org/Site/Talks.html, 2010.

[4] Gilad Bracha. Newspeak programming language draft specification version 0.06. http://bracha.org/newspeak-spec.pdf, 2010.

[5] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, volume 28, pages 215–230, October 1993.

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

[7] Matus Chochlik. Mirror c++ reflection utilities. http://kifri.fri.uniza.sk/ chochlik/mirror-lib/html/, 2011.

[8] Pierre Cointe. Metaclasses are first class: the ObjVlisp model. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 156–167, December 1987.

[9] Yohann Coppel. Reflecting scala. http://lamp.epfl.ch/teaching/projects/archive/coppel_report.pdf, 2008.

[10] Adele Goldberg and Dave Robson. *Smalltalk-80: The Language*. Addison Wesley, 1989.

[11] Sun microsystems, java platform debugger architecture. http://java.sun.com/products/jpda/.

[12] David MacKenzie Julia Menapace, Jim Kingdon. The "stabs" debug format, 2004.

[13] David H. Lorenz and John Vlissides. Pluggable reflection: decoupling meta-interface and implementation. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 3–13, Washington, DC, USA, 2003. IEEE Computer Society.

[14] Jacques Malenfant, Christophe Dony, and Pierre Cointe. Behavioral Reflection in a prototype-based language. In A. Yonezawa and B. Smith, editors, *Proceedings of Int'l Workshop on Reflection and Meta-Level Architectures*, pages 143–153, Tokyo, November 1992. RISE and IPA(Japan) + ACM SIGPLAN.

[15] Stijn Mostinckx, Tom Van Cutsem, Stijn Timbermont, and Eric Tanter. Mirages: Behavioral intercession in a mirror-based architecture. In *Proceedings the ACM Dynamic Languages Symposium (DLS 2007)*, October 2007.

[16] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, Malaga, Spain, June 2010.

[17] Stan Shebs Richard Stallman, Roland Pesch. *Debugging with GDB*. Gnu Press, 2003.

[18] Randall B. Smith and David Ungar. Programming as an experience: The inspiration for self. In W. Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 303–330, Aarhus, Denmark, August 1995. Springer-Verlag.

[19] Allen Wirfs-Brock. A declarative model for defining smalltalk programs. invited talk at oopsla 96. http://www.smalltalksystems.com/publications/_awss97/SSDCL1.HTM, 1996.

[20] Allen Wirfs-Brock. A prototype mirrors-based refection system for javascript. https://github.com/allenwb/jsmirrors, 2010.

Author Index