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
abergel@dcc.uchile.cl

Gabriela Arévalo

Abstract

an abstract

1. Introduction

Coping with emerging requirements is probably one of the most difficult challenge in software engineering [3].

Mondrian is an agile visualization engine. It is used in more than a dozen projects. New requirements set by the increasing set of clients have an impact on assumptions that were hold for years.

Mondrian use simple two-dimensions rendering to graphically visualize an arbitrary domain. One of strong assumption Mondrian holds is the structure of its multiple cache mechanisms. Mondrian's caches are instance of the memoization technique[1]. Sending twice the same message returns the same value if no side effect impact the computation.

Unfortunately, the new requirements of Mondrian defeats the purpose of some of the caches it defines. One example is the bounds computation to obtain the circumscribed rectangle of a two-dimensional graphical element. This cache is useful when Mondrian is used to visualize 3d.

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 the cache and reset it. We subsequently undertaken a major refactoring of Mondrian's core: caches are not externalized and structured along a class hierarchy.

We were able to modularize the cache while preserving the overall architecture. Performances did not suffer from the refactoring.

The paper makes the following contributions:

- identification of memoizing cross-cutting concern
- refactorization of these cross-cutting concerns into modular and pluggable aspects
- lesson learnt

2. Making Mondrian Evolve

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

2.1 Turning Mondrian into a framework

Mondrian[2] [2] is an agile visualization library. A domain specific language is provided to easily define interactive visualizations. Visualizations are structured along a graph structured, made of possibly nested nodes and edges. Mondrian is a crucial component which is used in more than a dozen independent projects. To meet clients performance requirements, Mondrian authors are paying a great attention to provide fast and scalable rendering. To that purpose, Mondrian contains a number of cache to avoid redundant code executions.

Mondrian is now on the verge to become a visualization engine framework versus a library as it is currently. Mondrian is now used in situations that were not originally planned. For example, Mondrian has been used to visualize the real-time behavior of animated robots[3], 3D visualizations[4], whereas Mondrian has been originally designed to visualize software

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

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

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

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

source code using plain 2D drawing [1]. 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:` realizes 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 invocation 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 it computation. A user-performed interaction that leads to an update of the visualization invalidate the visualization. These memoization have were gradually introduced over the long development of Mondrian (which started in 2006). Each unpredicted usage leaded to a performance problem was has been solved using a new memoization. There is about 32 memoizations in the current version of Mondrian.

These caches have been shaped along the common usage of Mondrian. Visualizations produced in Mondrian are *all* static, employ colored geometrical objects and interactions are offered by simply on these objects.

Extending the range of applications for Mondrian invalidate some of the caches. 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 necessary operations for accessing and resetting a cache.

As exemplified with the `absoluteBounds` method given above, the caches are implemented by means of dedicated instance variables defined in the `MOGraphElement` class. That is to say, each cache is associated with an instance variable.

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.

In *MOGraphElement* hierarchy only remain a reference to the *Cache* class that contains all the caches Alexandre ►*I do not understand. Did I just say that?*◄.

Significant overhead. This ~~kind of~~ modularization solely based on delegating message ~~looses performances~~→has a sever overhead at execution time because the access to the cache variable is not direct. This causes a delay Alexandre ►*delay?*◄ when attempting to access the values that keep the caches. So, the separation of this concern is not a trivial problem. Specifically, when this solution the caches mechanism was 3 to 10 times slower, being the delay proportional to the number of elements.

2.4 Requirement for refactoring

Alexandre ►*@@HERE*◄

- to understand where the cache are defined and used
- no cost of performance must be incurred, else it defeats the whole purpose of the work
- Readability must not be reduced

3. Refactoring

The goal of the refactoring is the separation of the *Cache Concern* from the main class of *Mondrian*: *MOGraphElement* and its subclasses (*MOEdge*, *MONode*, and *MORoot*, totaling more than 235 methods and 1000 number of lines of codes). The refactoring process begins with the identification of the caches. This initial identification of the caches is done with the information provided by the developers of Mondrian and it lies in to know the variables related to the caches and the places where they are used. 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 on what is stored. After this initial identification, the fragment of codes in which the caches are used are grouped together based on the purpose of its use. Each group is associated with different activities:

- Initialize and reset the cache
- Retrieve the cache value
- Store data in the cache

Alexandre ►*what is a group and a subgroup?*◄ Santiago ►*Los grupos son: {Initialize, Retrieve, Store} y los subgrupos son aquellas lineas de codigo que tienen el mismo patron para cada uno de los elementos de estos grupos. Por emplo, los sugrupos de Initialize son Lazy initialization y cache Initialization. La verdad que el concepto de grupo-subgrupo no es algo importante o que sume valor. Creo que simplemente podemos poner que la division*

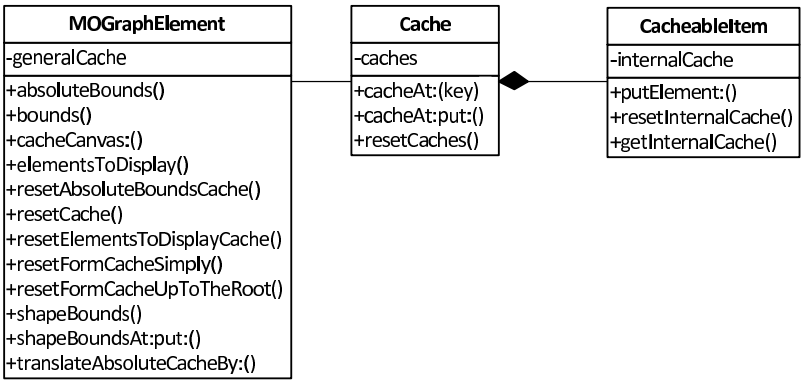

Figure 1. Cache behavior delegation. Alexandre ►*How is absoluteBounds written in that case?*◄

en grupos permitio detectar patrones.◄ Alexandre ►*I see. But why these groups and subgroups are not apparent in Figure 3? Shouldn't we have a class Initialization with two subclasses, LazyInitialization and CacheInitialization?*◄ Santiago ►*Cambie lo de grupo y subgrupo porque era confuso*◄ The task of group the fragments of code of the different caches is achieved with the goal of identifying possible strategies for refactoring. These groups allows the identification of code patterns that are repeated in the use of the caches. For each patter found a refactoring strategy is defined. These code patterns are described in the following subsections.

3.1 Pattern Description

This section presents the 5 code patterns we identified. Each pattern is described with a relevant typical occurrence, the number of occurrences of the pattern and a code example.

Reset Cache. A cache need to be invalidated when its content has to be actualized. We refer to this action as reset. The reset structure is *cache:=resetValue* where *resetValue* depends on the cache internal structure. Typically, the *resetValue* is nil or a new of a kind of Dictionary object. Eighteen occurrences of this pattern are found in the Mondrian code. 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. Next, the method *resetCache* is presented as an example. 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 need. This pattern shows the situation in which a verification is accomplished before access to a cache with the goal of avoid possible exceptions when the cache is nil. Typically, the structure of this pattern is: *if (cache==nil) cache:=newValue. return cache*. Five occurrences of this pattern are found in the Mondrian code. Next, the method *bounds* is presented as an example in which boundsCache is accessed.

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

Cache Initialization. This pattern represents a situation in which a value is assigned to a cache. The structure of the pattern is only an assignation: cache:=*aValue*. This pattern is found in three occasions. Next, the method *cacheCanvas* is presented as an example in which a value is assigned to cacheForm.

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

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.
```

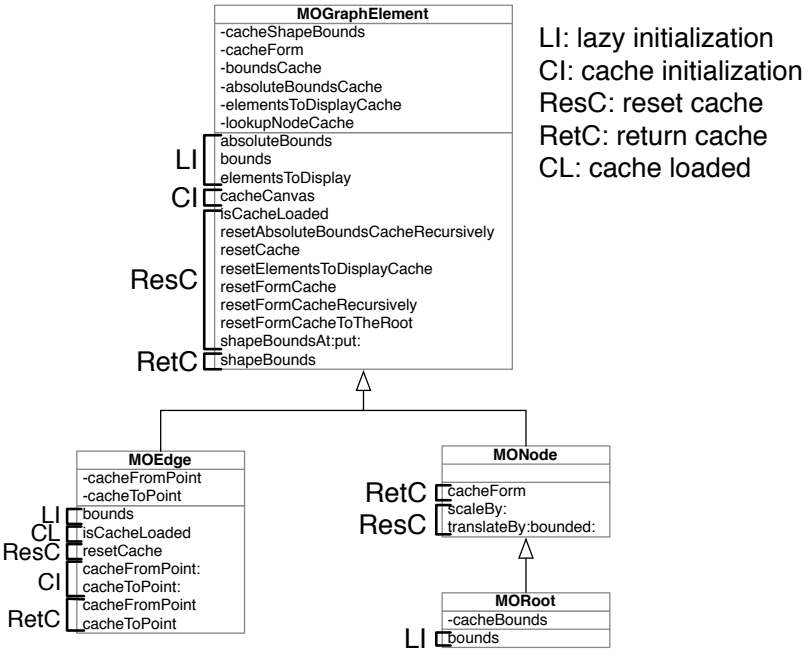

Figure 2. Pattern locations in the MOGraphElement hierarchy

Figure 2 shows the distribution of the caches over the main Mondrian classes.

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

3.2 Refactoring Strategy

Once the code patterns are identified, strategies to refactor them are established. The goal of the refactorization is the extraction of these patterns from the main code without changing the behavior of the system. The main constraint to refactoring the code is the preservation of a good performance of the cache mechanism.

Several alternatives were explored to encapsulate the *Cache Concern*. For example, one of the explored mechanisms was the separation of the concern by means of the definition of an exclusive class for managing caches. While this solution provided a good modular design, it introduced indirections to access to the caches (Section 2). Indirections caused a poor performance that slows down response times of the cache mechanism. For this reason, this is not a feasible solution.

After exploring a variety of options such as the use of proxies to intercept messages, an approach based on code injection was chosen. This solution has the advantage of encapsulating the concern in a new unit while the code that it is finally executed after the injection is similar to the original code of *Mondrian*. So, the performance is not affected. In order to encapsulate the source code related with the code patterns the *pragma* mechanism is used. *Pragmas* are the method annotation syntax implemented by Pharo.

The refactoring strategy used is: for each method that contains code related to the *Cache Concern*, the code related to the concern is extracted using a pragma that is defined in the method. The decision to define the pragma inside the method is in order to allow a better visibility of the code that is injected. The pragmas used have a structure according to each code pattern. In general, the pragmas structure is *<patternCodeName: cacheName>* where *cacheName* indicates the name of the cache that will be injected and *patternCodeName* indicates the pattern code to be generated. For example, the pragma *<LazyInitializationPattern: #absoluteBoundsCache>* indicates that the *Lazy Initialization* pattern will be injected for the cache *absoluteBoundsCache* in the method in which the pragma is defined. **Alexandre** ►*it also creates a variable in the class doesn't it?*◄ **Santiago** ►*No, the only addition into the original method is the pragma*◄

Once that the cache code is extracted into the pragmas, the code to be injected is automatically generated before the execution of the system. Specifically, the automatic injection of a pragma in a method is achieved following the next steps:

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

```
MOGraphElement>>absoluteBounds
   <LazyInitializationPattern: #absoluteBoundsCache>
   ^ 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.

a new method called `computeMOGraphElementAbsoluteBounds` is created.

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

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

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

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

In this way, the refactored cache code is executed with Mondrian.

In order to automatically generate the code to be injected, the injector code mechanism provides a *CachePattern* interface. In this way, each cache pattern has to implements this interface which allow the generation of the methods mentioned above. Basically each subclass is responsible of the definition of the pragma to be used and the generation of the code sentences to be injected related with the cache whereas the interface *CachePattern* creates the methods to be added to the system. This class hierarchy is shown in Fig. 3.

Next, the refactorings applied to each code pattern are presented.

Reset Cache. In order to refactor this pattern each statement that resets a cache was extracted using a pragma. The pragma contains the cache to be resetted. Owing to 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. As is shown in Fig. 3, this hierarchy is composed of the classes *AbstractResetCachePattern*, *BeforeResetCachePattern*, and *AfterResetCachePattern*. The pragmas 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 3.1 of the method *resetCache*, a pragma 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 pragmas used are the 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 pragma. 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* performs additional activities that do not involve the cache variables. For this reason they remain in the *resetCache* method.

After the code injection the resetCache method is transformed into:

```
MOGraphElement>>resetCache
   absoluteBoundsCache:=nil.
```

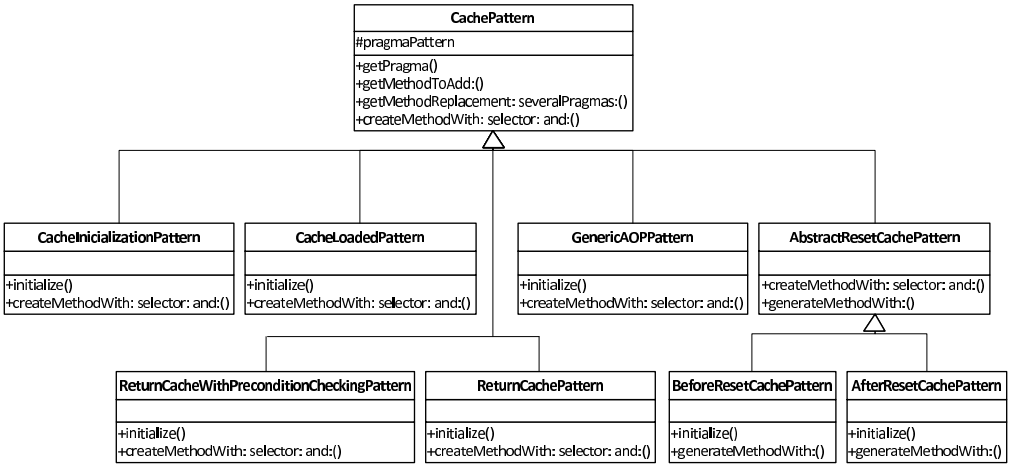

Figure 3. Pattern hierarchy.

```
   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 encapsulated into a pragma 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 *bounds* method presented in the previous section, the code related to the cache is extracted using the pragma 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.
```

In this way, the code to be generated for 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 assignation, the first section of the assignation (*cacheName:=*) will be generated automatically by the code injection mechanism using a pragma defined as <CacheInitializationPattern: cacheName>. So, only the value at which is initialized remains in the method. In the case of the example presented in Section 3.1, 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 pragma. The pragma is defined as <ReturnCachePattern: cacheName>. Following, the refactored code for 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 a pragma defined as <CacheLoadedPattern: cacheName>. The code generated contains a sentence in which the checking is done for all the caches defined in the pragmas of this pattern contained in a method. In the case of the example presented in Section 3.1, the refactored code is shown below:

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

With the use of these patterns the *Cache Concern* is refactorized properly in more than 85% of the methods of the `MOGraphElement` hierarchy that uses one or more caches. **Alexandre** ►*Does it mean that 85% of Mondrian methods use a cache?*◄ **Santiago** ►*No, I mean that from all the methods that uses a cache 85% could be refactored with the pattern structure*◄ The main reason because some of the uses of the caches are not encapsulated by means of cache patterns are (1) the code belongs to a cache pattern but the code related with the cache is too mixed with the main concern, or (2) the code does not match with any of the patterns described. For example the 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 been 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 patterns are also refactored by means of pragmas. For these cases a *Generic AOP* pattern is used. The pragmas used have the structure *<cache: cacheName before: " after: ">* where *cache* indicates the name of the cache that will be injected. The clauses before and after 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 pragma, a call to the new method will be added, and the code contained in the after clause of the pragma 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 can be seen, all the setences with references to the cache *lookupNodeCache* are encapsulated into the before clause of the pragma.

4. Results

Alexandre ▶*We also need to provide some benchmarks. There is a class in mondrian that does exactly this*◀

Alexandre ▶*We need to be stronger on this section*◀

The use of the presented patterns could be 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 currently Mondrian caches (and also those that may be introduced in 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, 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 is smalls fragments of code. These fragments of code are encapsulated by the patterns 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 during the refactoring process was not to 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 observed between the system before and after applying refactorings 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. The details of the benchmarks results are shown in Figure 4 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 can be seen, as was expected, there are not remarkable variations during these displaying.

Using cache in the main logic This experience has been the opportunity to rethink 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 rely on the cache to implement its semantics. This is obviously wrong and this is situation is marked as a defect[5].

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 of it by mean of cache patterns almost unviable. So, this method was restructured using the Generic AOP pattern.

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

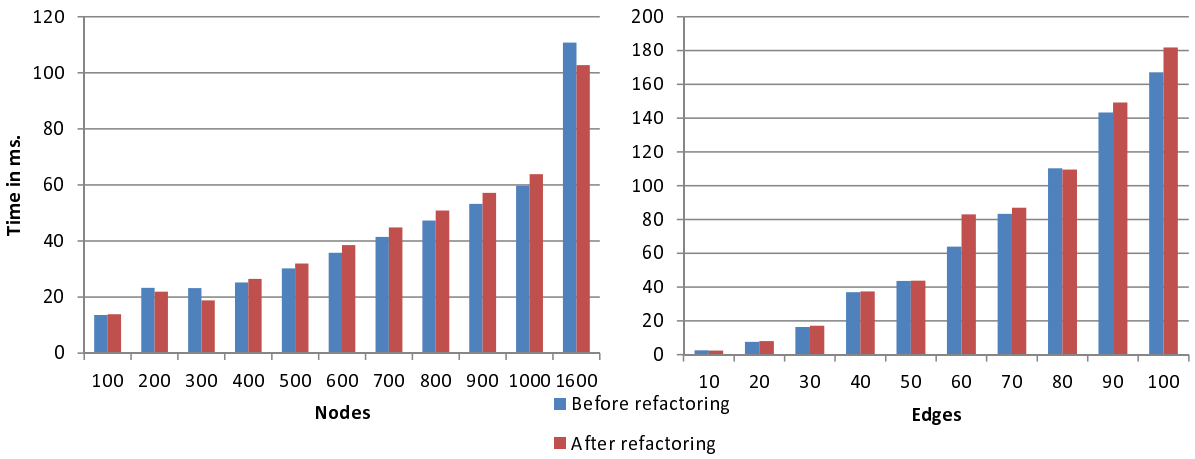

Figure 4. Benchmark of performance.

5. Discussion

Alexandre ▶*What are the benefits of the approach? Can I unplug cache?*◀

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 was shown, in this case the caches are resetted at the beginning of the method and after that the method *resetElementsToLookup* and *resetMetricCaches* are invocated in contrast with the original method in which the former was invocated 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 checked in a manual way and by mean of automatic test.

Alexandre ▶*How many occurrences of statement reordering?*◀

Importance of the tests However, in practice, this has not caused any noticeable problem. The extensive test set of Mondrian remained green after the instrumentation.

6. Related Work

Most of the work in AOP has been focused in Java and AspectJ. An approach called AspectS has been proposed for Smalltalk-Squeak [**?**]. This is a general purpose AOP language with dynamic weaving. **Santiago** ▶*No estoy seguro de como justificar que no utilizamos AspectS para los caches. Si funciona en squeak funciona en pharo? Una posibilidad es el overhead que implica la definicion de los pointcuts y aspectos y que no se garantizaba la misma performance en la ejecucion*◀ Several approaches have been presented in order to refactor and migrate OO systems to AO ones. Some of these approaches use a low level of granularity focusing in the refactorization of simple languages elements such as methods or fields [**? ? ? ? ?**]. On the other hand, other approaches are focused in a high level of granularity. This kind of approach tries to encapsulate into an aspect an architectural pattern that represents a CCC. That is, these approaches are focused on the refactorization of a specific type of concern. Our work is under this category. Others works that deals with the refactorization in a high level of granularity are discussed next. Da Silva et al. [**?**] 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 an specific concern and it is encapsulated into an aspect by means of a set of fixed refactorings. Van der Rijst et al.. [**? ?**] propose a migration strategy based on crosscutting concern sorts. With this approach the CCCs 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. [**?**] present a role-based refactoring approach. Toward this goal the CCCs 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. [**?**] 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 [**?**] propose a web cache to evaluate an aspect-oriented approach based on XML annotations.

7. Conclusion

Acknowledgments

We gratefully thanks ...

References

[1] M. Lanza and S. Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Soft-*

ware Engineering (TSE), 29(9):782–795, Sept. 2003. doi: 10.1109/TSE.2003.1232284. URL `http://scg.unibe.ch/archive/papers/Lanz03dTSEPolymetric.pdf`.

[2] 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`.

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