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 avoid redundant computation. These caches are structured along Mondrian purpose: generating static two-dimensional visualizations. Particularly in the case of Mondrian, we have noticed that the caches are meaningless for the evolution being made on Mondrian. Using aspect-oriented programming, we have refactored these caches into well defined aspects to address the evolution 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 [12].

This paper presents a solution to a maintenance problem we recently faced 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.

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

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

[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 [11]. 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 [8]. 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 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 [**?**]. These code patterns are described in the following subsections.

3.2 Pattern description

We identified 5 code patterns based on Mondrian source code and we describe them 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).

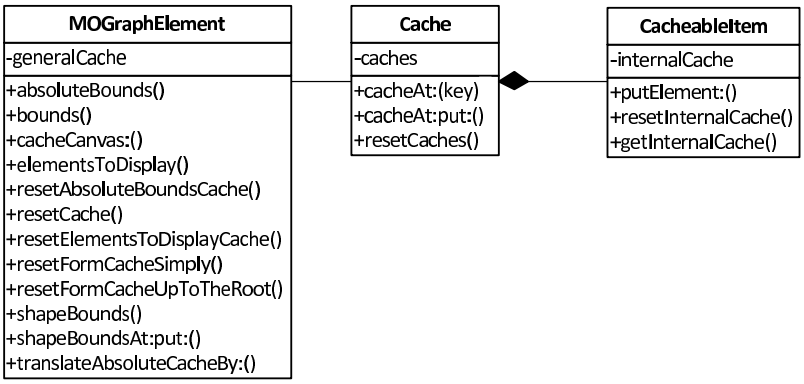

Figure 1. Cache behavior delegation.

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.

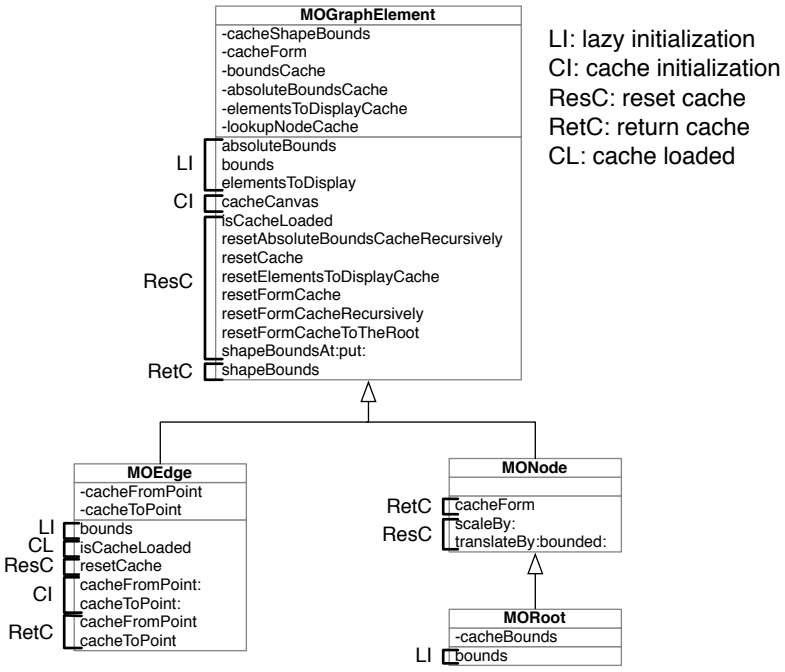

Figure 2. Pattern locations in the MOGraphElement hierarchy

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:

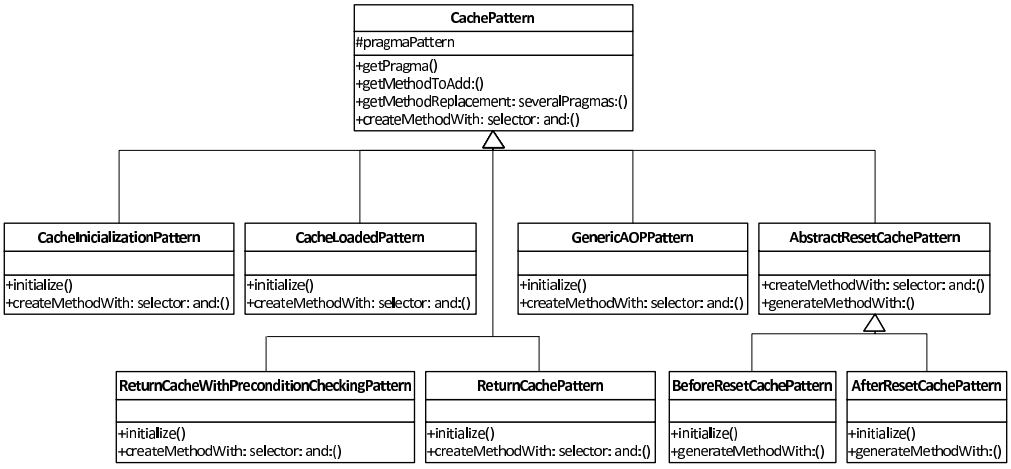

Figure 3. Pattern hierarchy.

```
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 because (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 presented patterns are 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 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.

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

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, 13, 15]. 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. Similarly to the last work, Van der Rijst et al. [10, 14] propose a migration strategy based on crosscutting concern sorts. With this approach the crosscutting concerns are described by means of concern sorts. In order to refactor the code, each specific crosscutting concern 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 crosscutting 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. Similar to us, this approach allows the reuse of the description of a crosscutting concern however, it does not mention how the code should be refactored.

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 [9] propose a web cache to evaluate an aspect-oriented approach based on XML annotations.

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

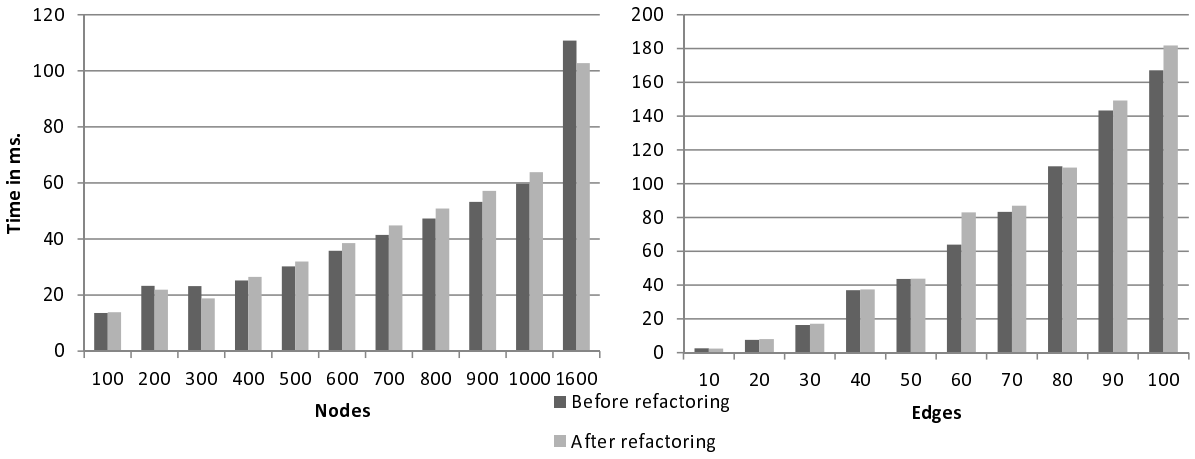

Figure 4. Benchmark of performance.

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] 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`.

[9] 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`.

[10] 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.

[11] 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`.

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

[13] 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`.

[14] 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.

[15] 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.