Memoization Aspects: a Case Study

Santiago A. Vidal[1,2] Claudia A. Marcos[1] Alexandre Bergel[3]
Gabriela Arévalo[2,4]

[1]ISISTAN Research Institute, Faculty of Sciences,
UNICEN University, Argentina

[2]CONICET (National Scientific and Technical Research Council)

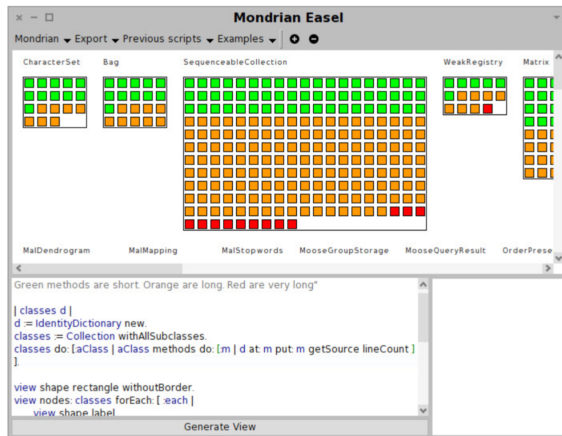
[3]PLEIAD Lab, Department of Computer Science (DCC),
University of Chile, Chile

[4]Universidad Nacional de Quilmes, Argentina,

International Workshop on Smalltalk Technologies (ESUG 2011)

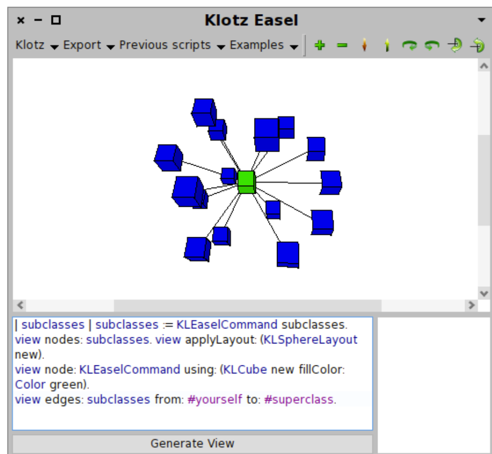
Mondrian

- Mondrian is an agile visualization engine implemented in Pharo, and has been used in more than a dozen projects



Dealing with Mondrian Evolution

- Mondrian has several caches
- Each unpredictable usage led to a performance problem that has been solved using a new memoization.



Mondrian Computations

Memoization

An optimization technique used to speed up an application by making calls that avoid repeating the similar previous computation

Mondrian caches are instances of the memoization technique

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

Proposal

Problems

- The caches that are used intensively when visualizing software are not useful and may even be a source of slowdown and complexity in other contexts.
- The legibility of the methods with memoization has been affected.

Proposal

Problems

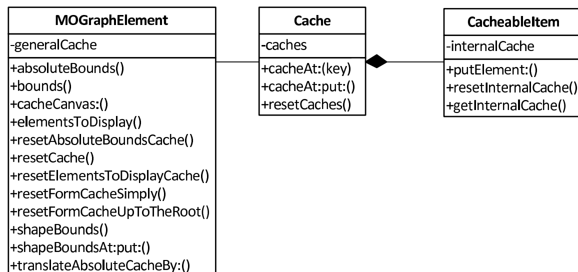
- The caches that are used intensively when visualizing software are not useful and may even be a source of slowdown and complexity in other contexts.
- The legibility of the methods with memoization has been affected.

Goals

- Identification of memoizing crosscutting concerns
- Refactorization of these crosscutting concerns into modular and pluggable aspects

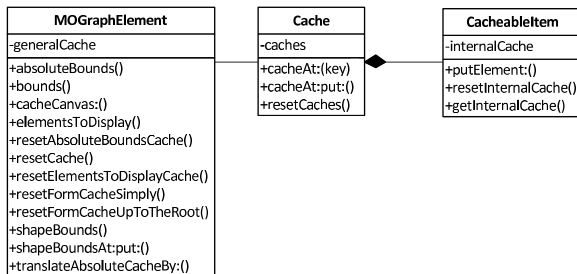
A Naive Solution

- General operations for accessing and resetting a cache



A Naive Solution

- General operations for accessing and resetting a cache



Problem

- Significant overhead (3 to 10 times slower)

Requirements for Refactoring

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

Identifying caches

- The caches are mostly identified by browsing the methods in which the cache variables are referenced and accessed.

Identifying caches

- The caches are mostly identified by browsing the methods in which the cache variables are referenced and accessed.
- 9 caches were found.

Identifying caches

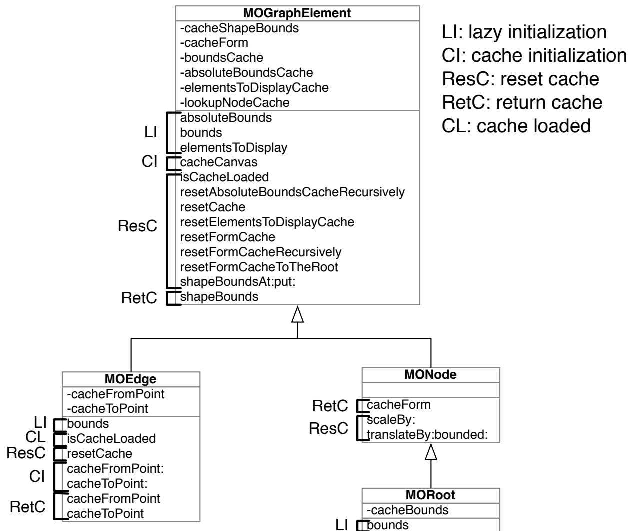
- The caches are mostly identified by browsing the methods in which the cache variables are referenced and accessed.
- 9 caches were found.
- The caches were grouped together based on the purpose of its use:
  - Initialize and reset the cache
  - Retrieve the cache value
  - Store data in the cache

Identifying caches

- The caches are mostly identified by browsing the methods in which the cache variables are referenced and accessed.
- 9 caches were found.
- The caches were grouped together based on the purpose of its use:
  - Initialize and reset the cache
  - Retrieve the cache value
  - Store data in the cache

These groups allow the identification of code patterns.

Patterns identified



LI: lazy initialization
CI: cache initialization
ResC: reset cache
RetC: return cache
CL: cache loaded

Pattern description

Lazy Initialization: In some situations it is not relevant to initialize the cache before it is actually needed.

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

Reset Cache: A cache has to be invalidated when its content has to be updated.

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

Cache Concerns as Aspects

- The goal of the refactorization is the separation of these patterns from the main code without changing the overall behavior.

Cache Concerns as Aspects

- The goal of the refactorization is the separation of these patterns from the main code without changing the overall behavior.
- Aspect weaving is achieved via a customized AOP mechanism based on code annotation and source code manipulation.

Cache Concerns as Aspects

- The goal of the refactorization is the separation of these patterns from the main code without changing the overall behavior.
- Aspect weaving is achieved via a customized AOP mechanism based on code annotation and source code manipulation.
- Refactoring strategy: for each method that involves a cache, the part of the method that deals directly with the cache is removed and the method is annotated.

Cache Concerns as Aspects

- The goal of the refactorization is the separation of these patterns from the main code without changing the overall behavior.
- Aspect weaving is achieved via a customized AOP mechanism based on code annotation and source code manipulation.
- Refactoring strategy: for each method that involves a cache, the part of the method that deals directly with the cache is removed and the method is annotated.
- The annotation structure is <patternCodeName: cacheName>
    - <LazyInitializationPattern: #absoluteBoundsCache>

Injection Mechanism I

For every annotation a method may have, the code injector performs the needed source code transformation to use the cache.

1. A new method is created with the same name as the method that contains the annotation but with the prefix "compute" plus the name of the class in which is defined.

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

MOEdge>>computeMOEdgeBounds
```

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

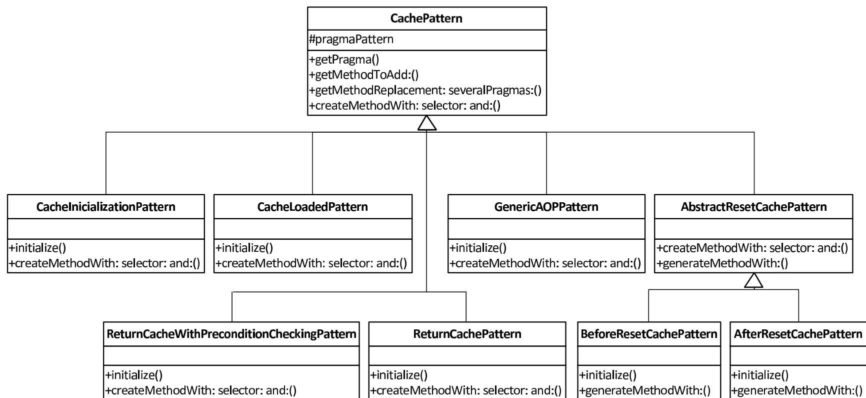
```
MOEdge>>computeMOEdgeBounds
   ^ self shape computeBoundsFor: self.
```

Injection Mechanism II

3. The code inside the original method is replaced by the code automatically generated according to the pattern defined in the annotation

```
MOEdge>>bounds
   boundsCache ifNotNil: [ ^ boundsCache].
   ^ boundsCache:= computeMOEdgeBounds
```

Injection Mechanism III

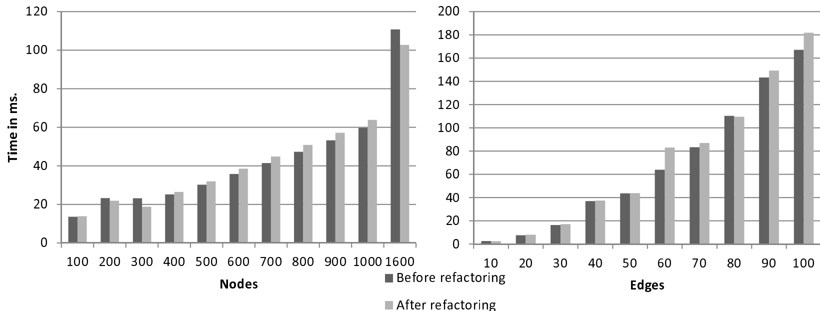


Maintainability

The contribution of this approach is twofold:

1. The mechanism of encapsulation and injection can be used to refactor the current Mondrian caches improving the code reuse.
2. The code legibility is increased because the Cache Concern is extracted from the main concern leaving a cleaner code.

Performance



Patterns identified

LI: lazy initialization
CI: cache initialization
ResC: reset cache
RetC: return cache
CL: cache loaded

Injection Mechanism I

For each annotation a method may have, the code injector performs the needed source code transformation to use the cache.

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.

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

MGEdge>>computeMGEdgeBounds
```

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

```
MGEdge>>computeMGEdgeBounds
  ^ self shape computeBoundsFor: self.
```

Injection Mechanism III

Performance