CONFRONTATION OF AN ASPECT-ORIENTED SOLUTION WITH AN OBJECT-ORIENTED SOLUTION, A CASE STUDY ON CACHING

Jeroen Boydens
KHB (Polytechnic Bruges-Ostend)
Department of Industrial Science and Technology
Zeedijk 101 B-8400 Oostende Belgium
K.U.Leuven
Department of Computer Science
Celestijnenlaan 200A B-3001 Leuven Belgium
Jeroen.Boydens@cs.kuleuven.be

Eric Steegmans
K.U.Leuven
Department of Computer Science
Celestijnenlaan 200A B-3001 Leuven Belgium
Eric.Steegmans@cs.kuleuven.be

ABSTRACT
This paper argues that the use of aspect-oriented programming techniques do not lead to clean object-oriented solutions. First, a caching example is implemented using an aspect-oriented approach. This approach is invisible for the original class designer. Then, this aspect-oriented solution is adjusted to throw exceptions, causing the client to break. This has the advantage that clients are now informed that extra information is available. Next, the paper solves the same problem using the object-oriented strategy and standard design patterns, such as the decorator pattern and the factory method pattern. The use of design patterns, leads to more disciplined solutions, but adds extra repetitive work. Finally, the paper compares both advantages and disadvantages of the aspect-oriented and the object-oriented solution.

KEYWORDS

1. INTRODUCTION
In software development, object-oriented programming is an established strategy. Design patterns are taking a leading role when structuring software systems. It is tedious to create complete separation of concerns, especially when multiple inheritance is not supported. Aspect-Oriented programming claims to be a solution to this problem, by weaving extra code statically or dynamically in the application (Filman 2005). The problem with this aspect-oriented solution is that there are very little restrictions on the weaving of aspect code. The original code behaviour can change, unnoticeable for the client of this code. We present an application that introduces caching, in which we would be tempted to use an aspect-oriented solution (Penchikala 2004). We will show that the overall structure resulting from an object-oriented approach is still superior. The object-oriented solution uses well-known design patterns, resulting in a much more disciplined way of working. However, contrary to the aspect-oriented solution, the object-oriented solution causes some repetitive work.
2. SAMPLE APPLICATION

When we create distributed applications, all data concerning a specific request is sent over the network. The request itself and all its arguments are serialised on departure at one side, and de-serialised on arrival at the other side, as illustrated in Figure 1. At the remote side this data is currently only used by the executing request. We can use a caching principle to update data used during one request, with values arriving in a following request. Since this last request brings more recent information, the first request can be notified of the fact that more accurate data is available.

Figure 1. Serialisation of local object in remote call.

This caching principle is used in a distributed banking application. Essential for the purpose of this paper is the fact that information is distributed across a number of nodes. The server for the central bank is located in the centre of the distributed application. It manages all bank accounts owned by customers. The customer records are distributed across the different branch offices. The branch offices use the accounts stored on the central bank, and keep track of extended customer information. The central bank needs information about all the customers, but is not willing to manage that huge amount of information. This head office does the management of the account information, and rarely uses all details of the customer-records. When the branch offices are communicating with the central bank, partial customer information is sent over to the head office. Since numerous applications are sending such information to the central bank, updated values may be arriving while handling current requests. The caching solution notifies the executing requests if a more recent version is available. In Boydens (2006) we describe this sample application using the caching strategy as illustrated in the sequence diagram in Figure 2.

Another solution to solve part of this problem would be to introduce distributed transactions for the customer records. We think the introduction of transactions to solve this problem is correct, but introduces a lot of complications. Distributed transaction servers are needed, isolation levels (access restrictions) for the data must be studied, and transaction abortion must be handled. In our example, we only need a lightweight solution for having customer information available at the head office. The heavy machinery of transactions seems to be overkill in this particular case.
We will not be focusing on cache validation and cache eviction policies. Data objects are removed from the cache when their time-to-live attribute are expired, or when the complete cache is cleared. The main goal of this paper is not on caching itself, but rather on the addition of a behaviour to an existing program. Caching is purely used as a sample behaviour.

### 2.1 Data Class Setup

We will consider a data class, named `Customer` having a number of properties: `id`, `firstName`, `middleInitial`, `lastName`, `email`, `street`, `zip`, `city`, `state`, ... One of the prerequisites of the class is that it should be `Clonable` and `Identifiable`. `Clonable`, since we will be putting copies of data objects in the cache. `Identifiable` since we do not want to add an extra property to the class to identify objects, but want to use an existing `id`-property instead.

Whenever an inspector or a mutator is called on the data object, a check will be executed to see if a more recent version is available in the cache. Every update done to this data object will also be forwarded to the cache. This means that whenever there is a read or write access to any of the properties of the data object, the cache is validated.

### 2.2 Data Class Refactored

We will refactor the data class by extracting its interface. We will then enhance this interface by specifying exceptions thrown by methods specific to the caching mechanism. This way, we can refer to our `Customer` objects using this `ICustomer` interface as a static type. In situations where the extra caching behaviour is not needed, references will be pointing to `Customer` objects. This `Customer` class does not throw the extra exceptions, and is still compatible to the original interface (Liskov 1994). But when we do need the extra behaviour we will use `CachedCustomer` objects, for which the specific exceptions can be thrown.

This technique is also known as the "dependency inversion principle" (Martin 1996a). We do not want to change the existing class by introducing extra exceptions to its methods. Instead, we encourage the clients of the class to use a more general interface, and to work with this abstraction instead of the concrete class. This way, different concrete classes can be used by the client, without affecting the client code. We are now able to change the underlying concrete classes at will without breaking the client code, as long as we make them abide to the extracted interface.
3. ASPECT-ORIENTED SOLUTION

In this Section we introduce those concepts of AOP which we use in the solution. For further details we refer to complete descriptions of AOP and related work (Kiczales 1997). The underlying idea in AOP is to separate crosscutting concerns. These are concerns that are spread all over the application.

We model the caching mechanism as a separate concern. Instead of making calls to the caching infrastructure on different places in the class, we create one single caching aspect. This caching aspect is woven into the class needing it. The new class can now be utilised with the added aspect. This is illustrated in Figure 3.

![Figure 3. Weaving of an aspect in an existing class](image)

3.1 Caching Pointcuts

Pointcuts are defined as a set of join points. Join points are those places in the original code where the aspect can intercept with the current code.

The first pointcut we need is the **getters** pointcut. This is the pointcut that identifies all places in our class where an inspector (get-method) is called. It is defined as follows:

```
pointcut getters(Customer aCustomer):
  target(aCustomer)
  && call (* *.get*()) && ...
```

This getters pointcut provides a reference to a `Customer` object as contextual information to its advices. The advice will be able to interact with the object on which the get-method was invoked. The verification of the current value with the cached one will be inserted by the advice at this pointcut.

The second pointcut we need is the **setters** pointcut. This is the pointcut that identifies all places in our class where a mutator (set-method) is called. It is defined as follows:

```
pointcut setters(Customer aCustomer):
  !call(Customer.new(..))
  && target(aCustomer)
  && call (void *.set*(*)) && ...
```

This pointcut is used by the advice to verify that we are updating the most recent value, and to update the cache with the latest values.

The final pointcut we need is the **constructor** pointcut. This is the pointcut that identifies all places in our class where a constructor is called. It is used to insert information concerning new `Customer` objects in the cache, at the time they are created. The definition is similar to the previous pointcuts.

3.2 Caching Inter-type Declarations

Inter-type declarations allow us to change the class to which an aspect applies. In this part of the aspect, we introduce extra types for the class `Customer`. Part of the declarations to introduce the caching aspect are:

```
declare parents: Customer implements Cached;
private static Cache Customer.sharedCache;
private int Customer.versionNr = 0;
```

The first declaration states that the `Customer` class is now additionally implementing the `Cached` interface. This interface is used by the aspect to put our object in the cache. This interface is later explained in
Section 4.2. It contains a way to identify objects, a way to verify version numbers of objects, and a way to clone objects. The clone operation is used to store deep clones of the data object in the cache.

The second declaration gives access to a shared reference of the cache. The last attribute declaration introduces a version number in each object put in the cache. We use this version numbers to decide on more recent values of the Customer objects. The attribute is added and every object has an initial value of zero for its version number.

Finally, to get correct behaviour, all mutator and inspector methods declared in the Cached interface are introduced in the class through inter-type declarations. These methods are not quoted in the code above.

3.3 Caching Advices

We can use the declared pointcuts to weave extra code, called advice, in our application code (Hilsdale 2004). We will insert advice before every getter-call, to compare the current version with the cached version. We will also insert advice around the calls to the setters. Before a setter gets called, we verify that the current value is in conformance with the cached value. After the setter has been called, the cache must be updated. Care must be taken when the set-requests are running in a multi-threaded environment, since different threads could be updating the same cached values. The synchronisation blocking is added in the final code, but not displayed in the following advice. First, in Section 3.3.1 a standard AOP approach is taken. This approach is invisible for the client. Next, in Section 3.3.2 an alternative approach is taken. Advises are produced using a safe strategy: informing the client with exceptions when current values are out-of-date.

3.3.1 Standard Getters and Setters Advice

This advice invisibly introduces the caching aspect to the client. It uses the getters pointcut. Before the original get-method is executed, the following advice verifies the version of the target object in the cache. Whenever a get-method is called, the cached version is checked and an update may occur. The get-method then proceeds to its normal implementation.

```java
before(Customer aCustomer):getters(aCustomer){
    // verify versionNr complies to Nr in cache
    Customer customerInCache = (Customer) getCustomerFromCache(aCustomer);
    if (customerInCache.getVersionNr() > aCustomer.getVersionNr())
        update(customerInCache, aCustomer); // update(from, to)
    // proceed with the rest of the getter on the updated object
}
```

The update of the data object is handled through reflection in the update method. We iterate over all available set-methods of the object since we do not need to consider the read only properties. Each set-method is called on the to-object with a value received from the corresponding get-method on the from-object.

A comparable advice can be defined for the setters pointcut. Before the original set-method is executed, the current version is compared with the version in the cache. The advice then increases its version number, and all information is updated in the current target object. Then, the original implementation of the set-method is running, using a proceed instruction. After the set-method has completed, the cache is updated with the new values.

```java
around(Customer aCustomer):setters(aCustomer){
    // get version from cache
    Cached changingCustomer = getCustomerFromCache( aCustomer );
    changingCustomer.setVersionNr( changingCustomer.getVersionNr() + 1);
    update(changingCustomer, aCustomer); // update(from, to)
    proceed(aCustomer);
    putCustomerInCache(aCustomer);
}
```

3.3.2 Getters Advice and Setters Advice Throwing Exceptions

The advices described in Section 3.3.1 can result in conflicts. Consider a client using multiple properties of the same data object in a visualisation. When an update of the underlying data object is done, the complete user interface should be refreshed. The client is unaware of the fact that other fields are out of date and an
inconsistent state is displayed. Therefore the alternative advices throw an exception to client code when the version number is not in conformance with the number in the cache. The exception breaks the client code, but informs the developer that presumably other fields should be updated too. If the version of the target object is older than the version of the cached object, then a more up-to-date version is available in the cache. The advice notifies the client by throwing an unchecked exception. The client can then decide to update his version with the version from the cache. Only unchecked exceptions can be thrown since we are not able to change the signature of an existing method by using an aspect.

The same conflicts can occur for the around advice of Section 3.3.1. A similar around advice throws unchecked exceptions. If the cached version number is higher, an exception is thrown, because the client would be changing an out of date object.

4. OBJECT-ORIENTED SOLUTION

We will now describe the object-oriented solution for the same problem scenario. The solution is based on the extracted interface so we are able to use the "Dependency Inversion Principle" (Martin 1996a). Recall that we enrich this interface with checked exceptions, to signal conflicts with cached versions. The original class implements this interface without throwing the exceptions. A new CachedCustomer class implements this interface with the added exception behaviour. The new class uses a decorator pattern (Gamma et al. 1995) to add the extra behaviour to the get-methods and set-methods. We use the factory method pattern to create new instances. The factory-method returns an interface-type, so the client does not need to be adjusted when we add the caching behaviour or not. Using the factory method we can control which other decorations are added at runtime.

4.1 Cached Decorator

We use the merged decorator pattern as described by Gamma et al. 1995, as a basis in the object-oriented solution. The merged decorator class CachedCustomer is responsible for both the default implementation of the hook methods and the specific decoration operations. The standard decorator implemented these operations in the concrete subtypes. This reduced form lets us add only one concrete decoration per interface. But nothing prevents us from implementing more concrete decorations totally separate from each other. The general class diagram of the object-oriented solution is illustrated in Figure 4. In this illustration the Customer class displays only one property because stating all properties would overload the illustration. It is obvious that in the coded solution all properties are still considered.

The extracted interface ICustomer contains those methods that will be decorated. This interface is now also an extension of the Identifiable interface. The Identifiable interface enforces the introduction of an inspector in the Customer class returning a unique identification for each customer. The interface ICustomer also associates checked exceptions (CacheVersionException) with getters and setters for all the properties ascribed to customers. The client code should be based on the interface type, not on the concrete Customer class. We realise that these checked exceptions will break existing client code. However, in this way the client is aware that things may go wrong in manipulating customer information. Existing client code was written assuming that customer information was correct. Obviously, if this assumption is no longer true, the client code must be revisited.
The class **CachedCustomer** is the concrete decorator class. The class keeps a reference to a concrete **Customer** object through its static **ICustomer** reference. This lets us apply a number of decorators on top of each other. It throws checked exceptions when the decorated properties are accessed and more recent versions are available in the cache. It is up to the client to decide what to do in this situation. The client can abort the whole operation it is currently executing, or it could refresh its current state and continue.

### 4.2 Cached Interface

As illustrated in Figure 4, the **Cached** interface extends the **Identifiable** interface and the **Clonable** interface. It extends the **Identifiable** interface so the cache is able to use the **getId** request to identify its objects in the cache. It extends the **Clonable** interface because objects are put in the cache using deep cloning. The methods in this interface are used by the decorator pattern to request the current version of the cached object, and to change this version number if the cache is updated.

### 4.3 Applying the Decorations

In the **CachedCustomer** class, the concrete decorations are implemented. The decoration depends on the fact whether it is a decoration for an inspector, or whether it is a decoration for a mutator.

The getters decoration will first verify if the current version is still in conformance to the cached version. It will throw a checked exception if it detects a conflict. If no conflict is detected, the value is fetched from the object in memory and this value is returned to the caller. This is illustrated below for the method to fetch the e-mail of a customer. Similar decorations must be worked out for the getters of all the characteristics.

```java
public String getEmail() throws CacheVersionException {
    CachedCustomer aCachedVersion =
        (CachedCustomer) getSharedCache().getCached(this.getId());
    if (aCachedVersion.getVersionNr() > this.getVersionNr())
        throw new CacheVersionException("outdated Customer");
    return aCachedVersion.email; // proceed
}
```

The setters decoration is more complicated. Firstly, the verification must be done if the in-memory version is still in conformance with the cached version. Again a checked exception is thrown to the caller in
case of a conflict. Secondly, the update is executed on the current object in memory. And finally, the cached object is updated.

```java
public void setEmail(String email) throws CacheVersionException {
    CachedCustomer aCachedVersion =
        (CachedCustomer) getSharedCache().getCached(this.getId());
    if (aCachedVersion.getVersionNr() > this.getVersionNr())
        throw new CacheVersionException("outdated Customer");
    this.getCustomer().setEmail(email); // proceed
    getSharedCache().putCached(this);
}
```

In the `putCached` request, the version number of the cached object is updated and a copy of the updated object is registered in the cache. In this request, all necessary actions are taken to make sure the update takes place in isolation of concurrent threads. Again a checked exception can be thrown if, by the time of the update, the cached version has already been updated by another thread.

### 4.4 Factory Method

We use a factory method pattern (Gamma et al. 1995) to define a unique interface for creating an object but deferring the actual instantiation to a subclass. Using this factory-method we can control which decorations must be added to the instantiating object. The factory method returns an object of static type `ICustomer`. Using a factory method, we can be sure that all requested decorations are added.

We could create `Customer`-objects decorated with a number of decorations such as: caching, logging and tracing. Another advantage of this approach is that duplicated decorations are avoided, which is not the case when the developer himself must add these decorations.

If multiple different decorations are added, the `ICustomer` interface is upgraded to throw exceptions of type `DecoratorException`. This type then becomes a super type of `CachedVersionException`. This way, client code is required to catch the correct exceptions.

### 5. COMPARISON

#### 5.1 Strengths and Weaknesses of the Aspect-Oriented Solution

One of the greatest advantages of the aspect-oriented solution is that duplicated code is completely avoided. We define the getter and setter advice only once and apply it to all the get- and set-methods. The pointcut declaration distributes this advice to the different instances of the methods.

A second advantage of the aspect-oriented solution is that the code to add the caching concept is totally separated from the original business code. In the example, a caching concept is added without changing the current client code. This separation is only available at design time, since the aspect is weaved in the code at run time. This means that the developer is unable to predict the outcome of the software he writes. Instead of letting the developer control which operations can be overridden in subclasses.

A disadvantage of this solution is that the aspect-oriented solution functions in a very intrusive way. At design time we specify the join points of the aspect. This makes the aspect and the class highly coupled (Meyer 1997, Martin 1996b). If anything changes in the class, the aspect must be revisited. Since the class creator does not expect an aspect to be inserted, he could make changes to his code that would result in an unexpected aspect weaving.

A second disadvantage is that the client code is not able to see if the concept is being added. The client code can see no difference between `a Customer` and `a CachedCustomer`.

A third disadvantage is that the concept is always added or never added. The weaving of the aspect changes the class, leaving the client no options. All polymorphisms, concerning the concept implemented by the aspect, gets lost.

Finally, since the concept was added without changing the current client code, it seemed that the aspect was not breaking the client code. The client code still runs, but the original contracts of the methods are
broken. By introducing the aspect the application behaves in a different way. In one case invoking a setter with a value for the attribute is not registered, in the other case exceptions are thrown.

5.2 Strengths and Weaknesses of the Object-Oriented Solution

The most important advantage of the object-oriented solution is that it works in a very disciplined way, using well-known design patterns. The use of the decorator design pattern results in a clean separation of code concerning the crosscutting caching behaviour.

Another advantage is that a client is able to see if a specific concept is added by interrogating the dynamic types of the object. In the sample solution the factory method returned an ICustomer type which can be a CachedCustomer or a Customer. In the context of multiple decorations, the dynamic type could be Cached, Traced, Logged, ... Using the object-oriented solution, the client can even dynamically reorder these decorations. According to Hannemann and Kiczales (2002), this is not possible in an aspect-oriented implementation of the pattern.

Another advantage is that the client of the object-oriented solution is informed that, by using the ICustomer interface, exceptions will be thrown. Thus client code will not break when using the decorated objects. Instead the client is informed at compile time to take the checked exceptions into account.

Finally, the concept is added through a factory method on request by the client. This way, the client is able to cache one object, and not cache another.

A disadvantage of this object-oriented solution is that it introduces slight performance degradation, because the decoration inserts an object indirection. Since the server performs this indirection locally, the delay is unnoticeable for remote clients compared to the delay introduced by the remoting infrastructure.

The most important disadvantage of the object-oriented solution is the repetitive work needed. In the sample code in Section 4.3, we illustrated the decoration for one property. We must repeat this decoration for every property. The introduction of a dynamic proxy can solve this repetitiveness, since reflection offers the necessary tools. But experiments show that this reflective code is difficult to read, since all static type information is lost. This dynamic proxy introduces even more delays; the pointcut algorithm is evaluated at runtime.

6. CONCLUSION AND FUTURE WORK

In this paper we presented a caching framework for use in distributed applications. The strategy shown for caching can also be applied for other requirements like tracing, logging, ... We implemented the framework in two different ways: an aspect-oriented solution and an object-oriented solution. We then compared the advantages and disadvantages of both solutions. The object-oriented solution uses well-known design patterns and imposes a very disciplined approach. The aspect-oriented solution avoids the addition of code at multiple locations, but it breaks the contract of the methods. In modern software development it is unacceptable that these contracts, originally guaranteed by the developer, can be neglected or even broken. In our discussion we did not mention quantitative criteria such as the number of classes necessary to implement the solution, the number of methods inside these classes, the total amount of lines of code to implement the caching framework. Instead, we compared both implementations by a number of qualitative observations. We are convinced that these qualitative observations are important in terms of code readability, maintenance and extension of software applications. This paper is based on a typical phenomenon noted in various applications. Therefore, we can extrapolate our conclusion that the object-oriented solution for the caching framework is superior.

We see a number of ways how we could continue research on this topic. In the sample application we introduced a separate caching concept. Using the same strategy we could introduce other crosscutting concerns, such as a logging concept. Another topic we could search for is an approach to replace the pointcut by a new concept in the object-oriented solution. Decoration is now added per method, but if we could find a way to declare that a certain decoration must be applied to a number of methods, the repetitive work would be reduced. This could be realized by introducing a new language construct.
ACKNOWLEDGEMENT

We want to thank Johan Calu and Gwendolyn Rogge from KHBO, and Marko van Dooren, Sven De Labey and Koen Vanderkimpen from K.U.Leuven for their comments and contributions.

REFERENCES


