Karlsruhe Institute of Technology

An approach to domain-specific model optimization

Technical report of

Georg Hinkel

At the Department of Informatics
Institute for Program Structures
and Data Organization (IPD)

Date: 28th August 2013

Contents

1. Introduction

Model-driven engineering (MDE) is getting more accepted among both industry and academia. Thus, the need arises to optimize certain models in in-place model transformations among a certain domain-specifc cost functions. In general, such model-optimization tasks can easily be NP-hard problems and cannot be solved without using a backtracking operation.

This technical report is to describe NMF Optimizations, a framework and internal DSL to support creating domain-specific optimizations in a maintainable manner. However, these optimization tasks are then solved using brute-force algorithms. This makes such an optimization framework feasible for small model sizes, only.

NMF Optimizations as a framework is one of the contributions that are achieved through the master thesis "An approach to maintainable model transformations using internal DSLs" [Hin13]. The reason that the content presented in this report is not included in the master thesis is that the theoretical background of NMF Optimizations is only described very incomplete in this report. It remains subject of future work to improve and extend the background theory for the framework.

2. Example transformation

This transformation scenario has been taken from the Transformation Tool Contest 2013[1] (TTC). The goal of this case is to remove clones of attributes from a UML class diagram. Thus, the transformation involved in this scenario is an in-place transformation, as source and target domain are the same. Clones of attributes are identified as they have the same name and type. If a cloned attribute is identified, it should be removed, if possible. If two classes both having such a cloned attribute share a common base class, the attribute is moved to that base class (only in case no other class is affected by this), if both do not have a base class, a common base class is created and the attribute is moved there.

To be more precisely, the case description describes three rules to pull attributes to a base class:

1. If an entity *e* has two or more derived entities that all share an attribute with the same name and type, then these attributes should be merged into a new attribute of *e* with that name and type. Thus, the transformation deletes the copies of that attribute inside the derived classes. This refactoring called ”Pull up common attributes” [FB99].
2. If an entity *e* has two or more derived entities that share a common attribute with the same name and type, where not all of the derived entities share this attribute, a new class should be generated, such that is inserted in the inheritance hierarchy right between *e* and those derived entities and the common attribute is then to be pulled to that new entity. This refactoring is called ”Extract superclass” in [FB99].
3. If there are two or more root classes that share an attribute with the same name and type, then a new class is to be created that these former root classes inherit and the common attribute is to be pulled to the new entity.

Furthermore, the case description explicitely states the requirement that the transformation produces as few as possible new classes. The case description further suggests to apply these rules with descending priority to minimize the number of classes cxreated by the transformation.

Besides being an in-place transformation, this case further yields another typical transformation problem. That is that there might be multiple rules to eliminate cloned attributes

---

[1] `http://planet-sl.org/community/_/ttc/ttc2013/cases/ClassDiagramRestructuring/TTC2013-ClassDiagramRestructuring-Description.pdf`

applicable at the same time, which exclude each other. The case requires a solution transformation to perform these changes, such that as few as possible classes are created.

In this way, the transformation task can also be considered as an optimization task, where we do not assume that executing the rules in descending priority will lead to the same result.

The original case description in the TTC 2013 also contains two examples of possible inputs. The first scenario is presented in figure 2.1.

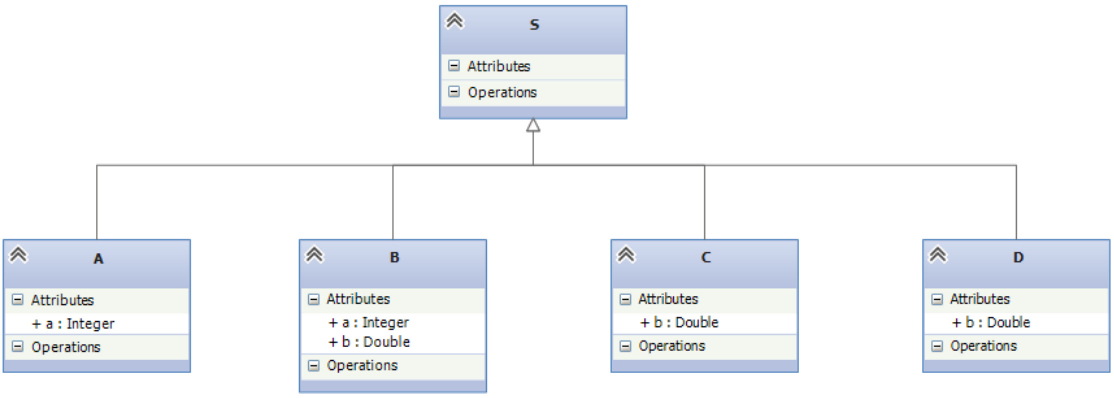

Figure 2.1.: The initial class diagram of test case 1

In this scenario, it would be possible to extract a superclass for either *a* or *b*. If a superclass for the attribute *a* is extracted, it is further possible to extract a superclass for *b* for the two classes *C* and *D*. However, this resulted in two newly created classes whereas only two attributes are deleted. Thus, the correct solution immediately extracts a superclass for the attribute *b*. In this way, only one new class is created, but also two attributes are deleted. The solution is shown in figure 2.2.

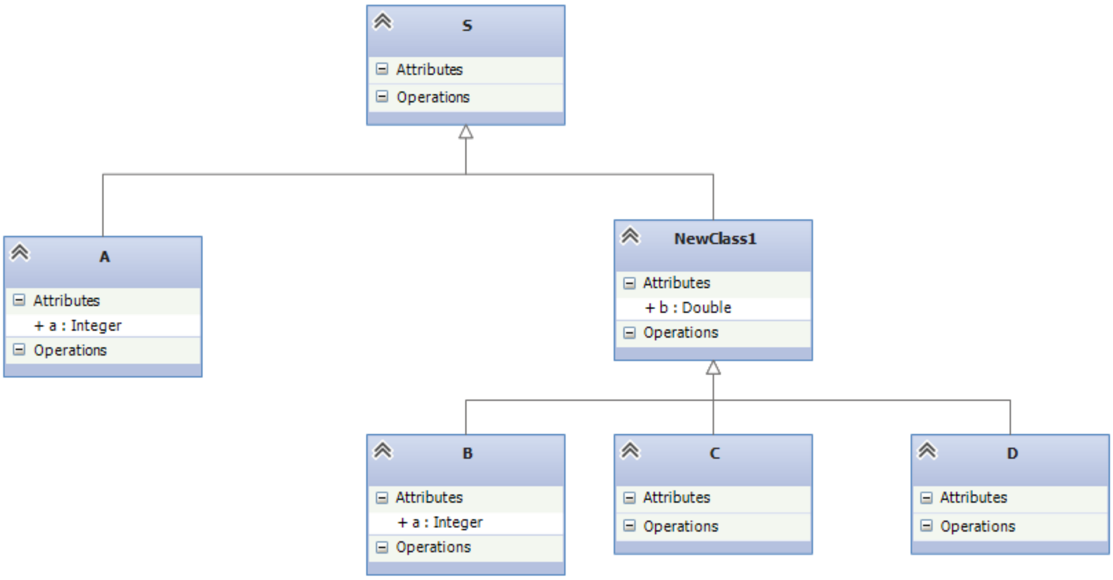

Figure 2.2.: The solution for the first class diagram restructuring test case

While this test case is rather simple, the second test case is more complex. While in the first test case, only the second refactoring rule to extract superclasses has been applied, the second test case is a scenario for the other two rules. The initial scenario of this test case is shown in figure 3.3.

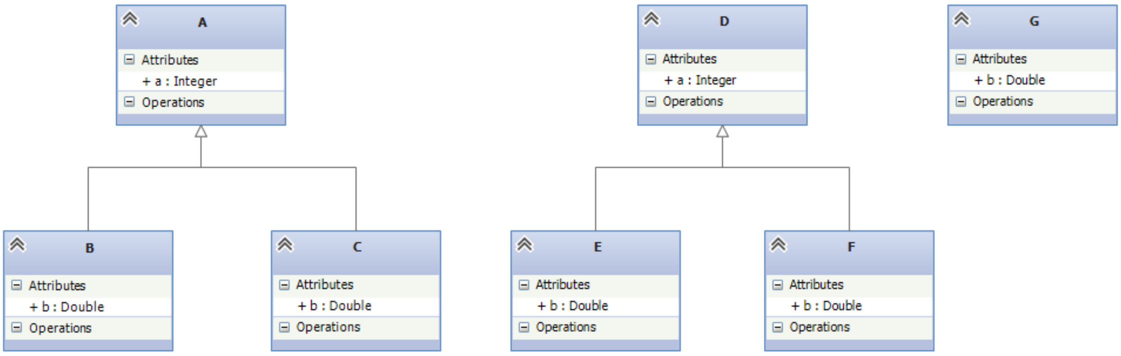

Figure 2.3.: The initial class diagram for the second restructuring test case

The correct solution for this test case is presented in figure 2.4.

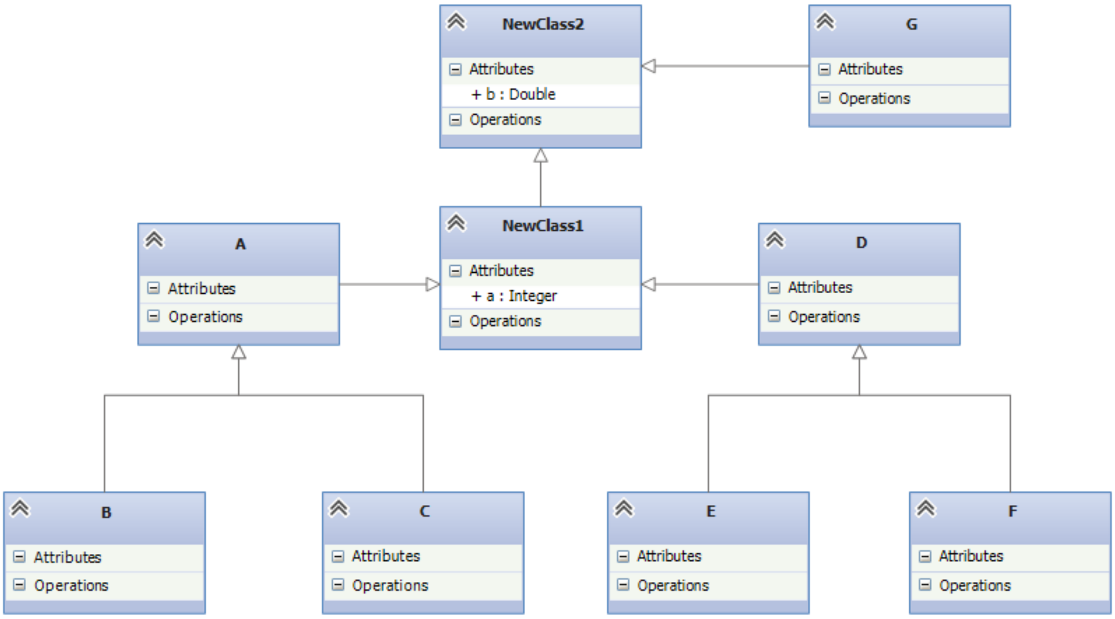

Figure 2.4.: The solution for the second class diagram restructuring test case

In the TTC2013, two solutions for this case were presented, both using heuristics to solve the case without using a backtracking mechanism [Hor13, SR13]. These solutions were sufficient to solve the provided test cases. However, counter-examples could be found that actually show that the heuristics do not solve the optimization problem in every case.

3. NMF Optimizations

This chapter presents NMF Optimizations, a framework to address the problems arising from non-deterministic optimizations. Like NMF Transformations, NMF Optimizations is also designed as an internal DSL to overcome these issues.

NMF Optimizations is a framework written in C# to provide support for optimizations. It aims to support optimizing models using modifications that perform domain-specific operations on the target model. The optimal outcome is determined via a a cost function. However, the modifications may exclude each other, which is what makes the optimization problem NP-hard. NMF Optimizations aims to form a model for such optimizations and therefore specify such optimizations declaratively using an internal DSL.

3.1. Terminology

This section introduces the main terminology of NMF Optimizations.

3.1.1. Optimization

An optimization in NMF Optimizations is a concrete optimization task. Other than a transformation in NMF Transformations, it is context-sensitive and thus not thread-safe. As a reason, the initialization overhead of an optimization is far less than the overhead for a transformation. However, one needs to be aware of this fact when combining NMF Transformations and NMF Optimizations to form transformations that make use of the optimization framework of NMF Optimizations.

Optimizations furthermore have a property whether they require that optimization modifications (see section 3.1.3) that perform the actual model changes should be forced. This means that the optimization can specify whether it should be allowed that solutions to this optimization still have modifications available.

If one for example consideres the restructuring of class diagrams mentioned in section 2 as optimization problem, where the changes to the class diagrams are represented by optimization modifications, in the possible solution scenarios there is no further modification available, as every further modification would definitely decrease the cost of the whole model. In this case, it is useful to specify that the optimization forces modifications to allow optimizers to prune several scenarios.

3.1.2. Optimization Items

Optimization items are elements that participate in the optimization. Participating in this context means that an element is important for the cost function of how the

3.1.3. Optimization Modification

Optimization modifications are the heart and soul of NMF OPTIMIZATIONS, as they define the possible operations on the models. These operations are represented by the method `Apply`. However, as modifications may exclude each other and optimization algorithms need to undo several operations, these modifications also have a `Reverse` method which is to reverse the modification.

Furthermore, modifications can specify an impact. This can be used for optimizers to determine whether two modifications will exclude each other. Modifications are assumed to not exclude each other if their impact is disjunct.

Because optimization modifications are domain-specific, they must be specified by the clients. This is done by implementing the appropriate interface `IOptimizationModification`. This interface consists of two methods and a property. The two methods are `Apply` and `Reverse` with the obvious functionality. As modifications can overlap and the effect of a change sometimes is not predictable, it is necessary that changes in the model must be reversible. However, a modification is not reversed unless it has been applied before. Thus, a modification can save the original state and apply it in the `Reverse` method.

Furthermore, a modification consists of the property `Impact`. This property is used to predict, whether two modifications exclude each other. The rationale behind this property is that if two modifications have disjunct impacts, they will not interfere each other. However, this is a conservative estimation. If two modifications have an intersection in their impact, they do not necessarily exclude each other.

Optimization modifications are the artifacts of an optimzation specification that involve most of the complexity.

3.1.4. Modification Patterns

Optimization modifications describe small portions of possible modifications on the target model. To be able to apply these modifications, it is necessary to specify when such a modification is applicable and to derive the necessary parameters of a modification. This specification is done via modification patterns in NMF OPTIMIZATIONS. These patterns specify when a modification is applicable through the query-syntax and dedicated monad implementations behind it.

Modification patterns can either be static or dynamic. The main difference between these two types is the way when new modifications are determined. While a static modification pattern only determines once which modifications are available for a certain model, dynamic modification patterns update the available modifications as the model is changed during the optimization.

The ways how to specify these modification patterns is presented in section 3.4.

3.1.5. Optimization Scenario

An optimization scenario is a possible outcome of the optimizer and as such a unit for evaluation. Optimization scenarios work on the same single model and change this model with the modifications. Scenarios are also responsible for determining the available modifications in their context.

3.1.6. Optimizer

An optimizer in NMF Optimizations is an algorithm that solves an optimization task described by an optimization. As the optimization strategy is not tightly coupled to the optimization models, the algorithm can easily be replaced. By default, NMF Optimizations contains several implementations for optimizers. The `GreedyOptimizer` is an implementation for a heuristical local optimizer. It just always applies the most promising modification. In contrast, the `BruteForceOptimizer` looks for all possible combinations of modifications and looks for the best.

For more detail how these optimizers work, see section 3.5.

3.1.7. Scenario Evaluator

A scenario evaluator is a component that evaluates optimization scenarios. This procedure is moved to a dedicated component to allow more sophisticated cost functions.

However, there are situations where it is not possible to determine a cost for the single items or the cost of the whole model cannot be calculated as the sum of the contained elements. An example for such a cost function is a cost function based on performance measurements, where the response time of a whole system cannot be just summed up from the response times of its components, but instead the components have complex interactions.

Thus, the evaluation of a scenario is moved to a dedicated component to allow for such a component to see the big picture.

3.2. Optimization problem to be solved by NMF Optimizations

Unlike NMF Transformations, where plenty transformation engines already exist, NMF Optimizations covers a field that has not been widely spread on both industry and research. Thus, in this section, the optimization that NMF Optimizations uses to describe domain-specific optimizations is introduced.

Therefore, consider a set S of possible scenarios, equipped with a cost function $cost : S \to \mathbb{R}_{\geq 0}$. An optimization modification now is a (partial) mapping $m : \mathbb{D}_m \subset S \to R(m) \subset S$ with an inverse (partial) mapping $m^{-1} : R(m) \to \mathbb{D}_m$, where $\mathbb{D}_m$ denotes the subset of scenarios where this modification is applicable and $R(m)$ denotes the range of this mapping. Note that an optimization scenario is not restricted to a single scenario. Instead, it can be valid on multiple scenarios. Denote $Mod(S)$ as the space of such modifications on the scenarios in S. Then we further have a function $mods : S \to \mathcal{P}(Mod(S))$ that for each scenario returns the possible modifications.

We call the tuple $(S, cost, mods)$ an optimization. Furthermore, we define a valid modification as a modification m that suffices the Valid Modification Property:

$$\forall s \in \mathbb{D}_m : m \in mods(s). \tag{3.1}$$

The Valid Modification Property is fulfilled, if a modification is available for every scenario it is defined on.

Being functions, optimization modifications can be composed. Let $m_1, m_2 \in Mod$ be valid modifications. If $R(m_1) \subset \mathbb{D}_{m_2}$ and thus for each scenario $s \in \mathbb{D}_{m_1}$ holds that $m_2 \in mods(m_1(s))$ we define a valid composition m_{12} as

$$m_{12} = m_2 \circ m_1. \tag{3.2}$$

The valid composition of m_1 and m_2 is just to apply m_2 after m_1, provided the m_2 is defined everywhere. Furthermore, there is a neutral element $id : S \to S, s \mapsto s$ with $id^{-1} = id$, so modifications form the algebraic structure of a monoid. However, the identity modification does not need to be valid (in fact, in the applications, most of the time it is not).

Clearly, the resulting function $m_{12} : \mathbb{D}_{m_1} \to R(m_2)$ is itself a modification, i.e. $m_{12} \in Mod(S)$ with $m_{12}^{-1} = m_1^{-1} \circ m_2^{-1}$ and $\mathbb{D}_{m_{12}} = \mathbb{D}_{m_1}$.

With these definitions, we can define a valid modification sequence for an initial scenario s_0 as a sequence $(m_i)_{i=1}^n$ with $n \in \mathbb{N}_0$ where $m_1, \dots, m_n$ are pairwise different and $m = m_1 \circ \dots \circ m_n$ is a valid composition and $m \in mods(s_0)$. In case of $n = 0$, the modification sequence is empty and represents the identity modification.

The optimization task now is the problem to find a valid modification sequence for a given initial scenario s_0 that minimizes the cost function, i.e. a valid modification sequence $m^* = (m_i^*)_{i=1}^{n^*}$ such that for each valid modification sequence $m = (m_i)_{i=1}^n$ holds that

$$cost(m^*(s_0)) \leq cost(m(s_0)). \tag{3.3}$$

To create more algebraic structure in the optimization modifications, the modifications are extended by a function $impact : Mod(S) \to \mathcal{P}(O)$, where O is an arbitrary set of modification elements, such that for each modifications $m_1, m_2 \in Mod(S)$ with disjunct impact, $impact(m_1) \cap impact(m_2) = \emptyset$, it holds that

$$\forall s \in S : m_1 \in mods(s) \wedge m_2 \in mods(s) \Rightarrow m_2 \in mods(m_1(s)). \tag{3.4}$$

The impact describes a set of elements that are affected by a modification. If two modifications operate on disjunct impact sets, then the application of one of these modifications does not interfere with the other modification.

3.3. Abstract syntax

Unlike NMF TRANSFORMATIONS, the abstract syntax of NMF OPTIMIZATIONS does not contain an execution semantic, which is just executed by an engine. Instead, the abstract syntax of NMF OPTIMIZATIONS model an optimization task that can be solved by an optimizer, but the execution semantic of this optimization is inside the optimizer alone. However, this procedure allows developers to write their optimizations in a declarative way, as the execution semantic of the optimization does not have to be specified, but is reused from existing optimizer implementations instead.

Applying the terminology of Fowler [Fow10], NMF OPTIMIZATIONS creates an adaptive model representing an alternative computational model, where the concrete computational model is chosen from some various implementations of optimizers.

Figure 3.1 shows a simplified class diagram containing the abstract syntax of NMF OPTIMIZATIONS. An optimization in NMF OPTIMIZATIONS consists of several static or dynamic modification patterns that specify the possible optimization modifications at a given point in time. These modification patterns in turn consist of modifications that they allow. Furthermore, an optimization is aware of its optimization items. The impact specified by the optimization modifications is also meant to target at these optimization items.

During optimization, the possible outcomes of the optimization process are represented by optimization scenarios. These scenarios have a cost property assigned to them that represent the total cost of this scenario. This property is to be populated by the optimizer.

Similar to NMF TRANSFORMATIONS, the implementation of the abstract syntax is a bit different due to several reasons. First of all, the concept of optimization items is not

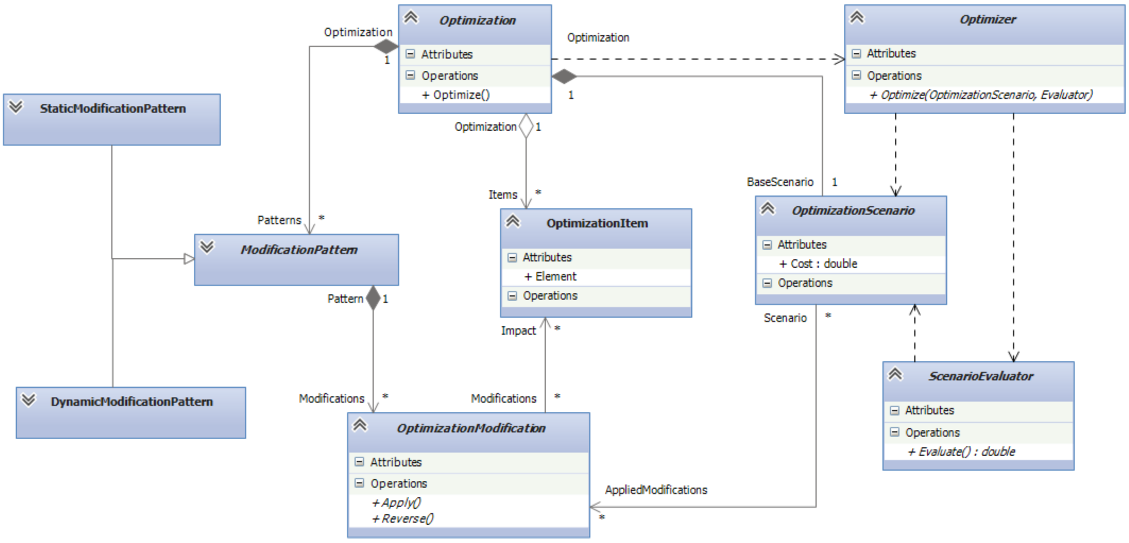

Figure 3.1.: The abstract syntax of NMF Optimizations

represented by separate classes. The sole purpose of this lack of representation is to ease the syntax for the optimization developer. The implementation is presented in figure 3.2.

The reference to the optimization modifications is turned into a method that returns a snapshot of the modifications. Whereas static modification patterns use a collection of available optimization modifications, dynamic modification patterns use an observable source of optimization modifications. Such observable sources are collections that can be observed for CRUD (Create-Read-Update-Delete) operations. Whenever any of these operations is executed, the according operation on the observers is fired. The details of this procedure are presented in section 3.4.2.

3.4. Specification of Modification Patterns

Modification patterns can either be specified as static or dynamic modification patterns. We will introduce static modification patterns in section 3.4.1 and dynamic modification patterns in section ch:nmfo:sec:ModificationPatterns:dynamic.

3.4.1. Static Modification Patterns

Static modification patterns represent collections of available modifications that do not change during the optimization, besides that an already applied modification cannot be applied once again.

An example of such optimizations where all possible modifications of the model do not change during optimization is the computation of a weighted set cover. In the weighted set cover problem, a finite set U called the universe must be covered by sets $s \in S \subset \mathcal{P}(U)$. Furthermore, a cost function $c : S \to \mathbb{R}_{\geq 0}$ determines the cost of including sets to set cover. A solution M for the set cover problem now must fulfill the following properties:

- M must be a valid selection of sets within S, i.e. $M \subset S$
- M must cover U, i.e. $\bigcup M = U$
- M must have minimal cost, i.e. $\forall N \subset S : \bigcup N = U \Rightarrow \sum_{s \in M} c(s) \leq \sum_{s \in N} c(s)$.

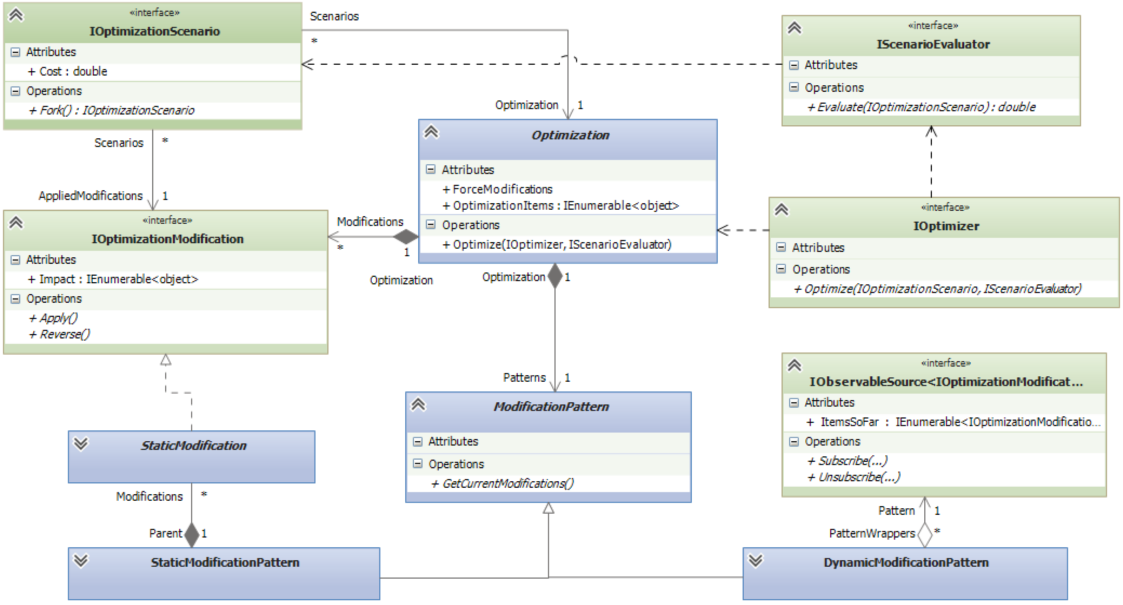

Figure 3.2.: Implementation model of NMF OPTIMIZATIONS

In this case, each of the sets in S can be represented by an optimization modification that inserts this set into the solution candidate, or removes it respectively in the `Reverse` operation. The possible sets that a solution for the set-cover problem can consist of is static meaning that it does not change during the optimization.

This example shows that optimizations in the commonality as treated by NMF OPTIMIZATIONS can easily be very hard, as the set cover problem is known to be NP-hard, even if the cost function is constant to 1.

To specify a static modification pattern, it suffices to specify the statically known optimizations. These optimizations need to be derived from `StaticModification`. The static modification pattern is then specified by calling the `AddPattern` method of the optimization class providing the collection of modification as parameter.

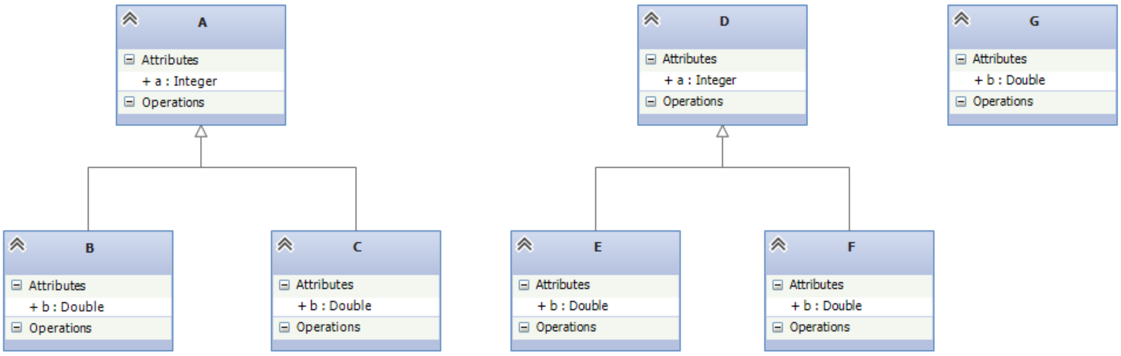

Figure 3.3.: The second test case of the class diagram restructuring case

However, if considers the restructuring of UML class diagrams (see section 2) as an optimization, the situation is different there. If one considers the testcase 2 that is included in the case description (see figure 3.3), simply applying the rules leads to an incomplete optimization, as not all modifications are applied. The resulting class diagram is shown in figure 3.4.

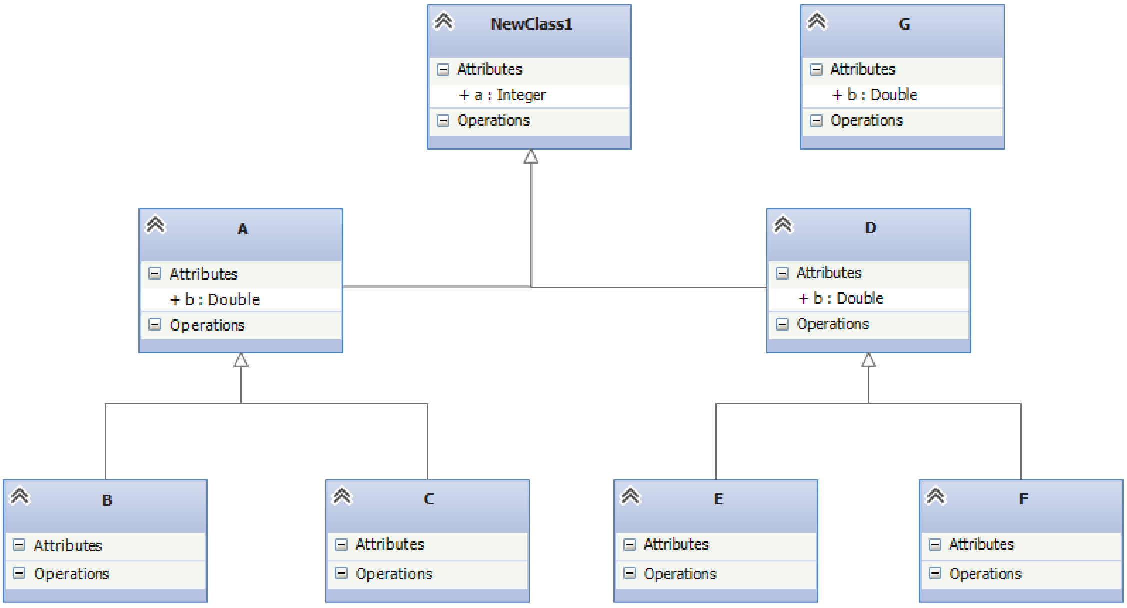

Figure 3.4.: Testcase 2 after applying all modification directly derivable from the initial model

Some rules like to pull the attributes `b` to the superclasses `A` and |D|, as well as pulling the attribute `a` to a newly created class can be derived from the initial class diagram. However, as the new class `NewClass1` is introduced, further modifications are possible, such as further pulling the attribute `b` to `NewClass1` and as soon as this happened, pulling it further to a newly created class, thereby merging it with the attribute `b` of class `G`.

Static modification patterns cannot support such situations, where modifications make other modifications possible that could not be determined in a static manner. Thus, in such cases, it is necessary to use dynamic modification patterns as featured in the next section.

3.4.2. Dynamic Modification Patterns

In contrast to static modification patterns, dynamic modification patterns can represent modification patterns specify patterns of optimization modifications that can vary during the optimization process. At first, the specification of dynamic modification patterns in a NMF OPTIMIZATIONS optimization is presented in section 3.4.2.1. Next, the update mechanism is explained in more detail in section 3.4.2.2.

3.4.2.1. Specification

Dynamic modification patterns are specified through observable sources of modifications. These observable sources can be iterated for their current elements. Furthermore, it is possible to register observers that are notified, whenever an item of that source is added, updated or removed. The default implementation of such an optimization source is represented by the class `ObservableSource`. This class provides a constructor that accepts a collection where the items should be read from. Observable sources can be composed using the query syntax of C# (or any other host language, respectively).

As an example, the optimization to solve the class diagram restructuring case (see section 2) optimzes the entities within the model. As the set of classes inside the class diagram will change during optimization, it is necessary to use an optimization source that enables change propagation instead of a pure collection. Thus, the class representing the optimization of the class restructuring has a property specifying the entities inside the model as an optimization source. Listing 3.1 shows its definition. This property is initialized in the constructor by a new instance of `ObservableSource` that persists changes to the underlying set of entities within the class diagram model.

Listing 3.1: The optimization item source for the entities

```
public OptimizationItemSource<Entity> Entities { get; set; }
```

This class `ObservableSource<T>` has three important operations: `Publish`, `PushUpdate` and `Revoke`. The main purpose of these methods is that they notify their observers that a new item is added to the source, an item is updated or an item is removed from the source. In case of `Publish` or `Revoke`, this change is persisted to the underlying collection. In any case, the changes are propagated to the observers.

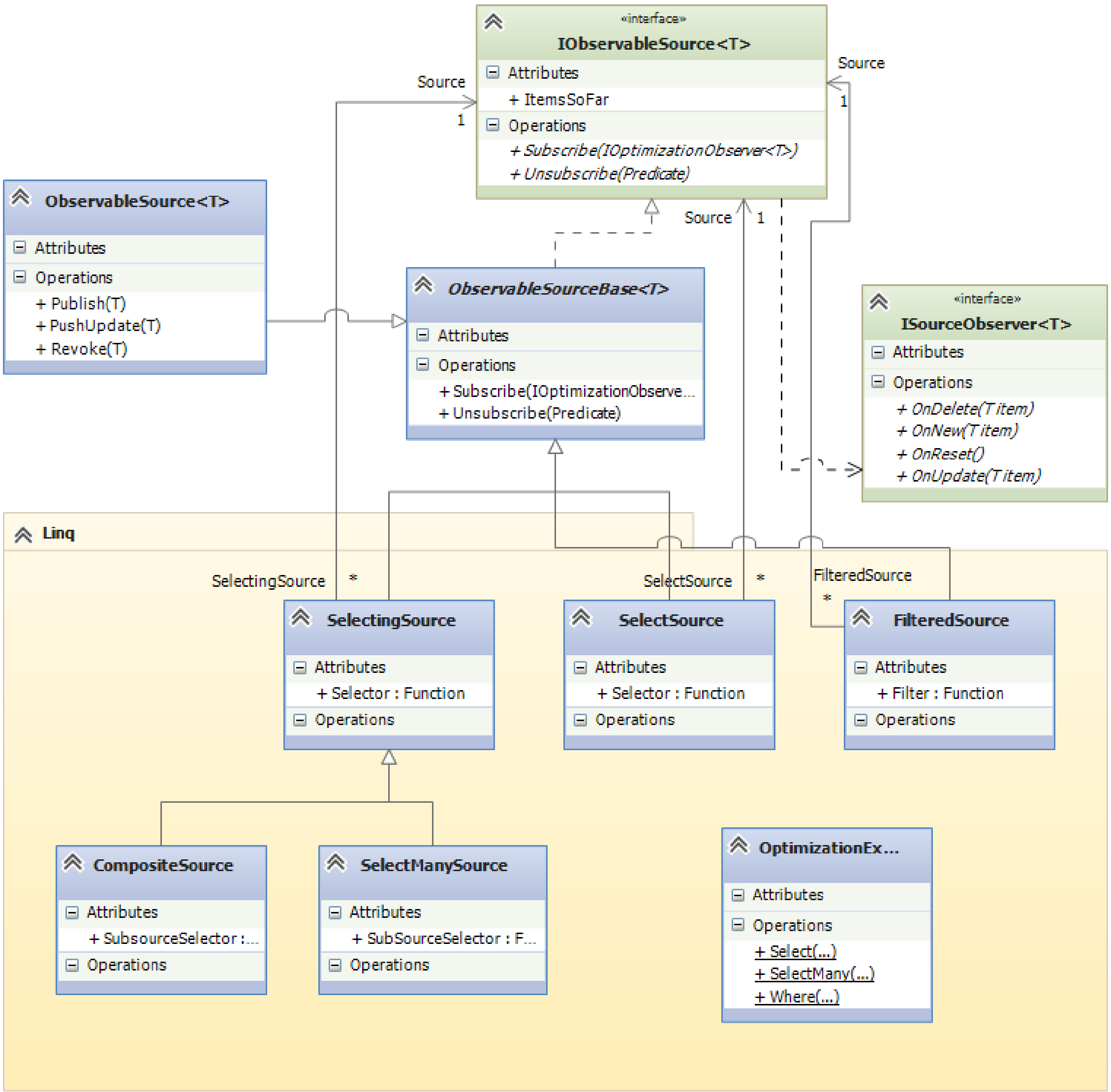

Figure 3.5.: The implementation of the query syntax for dynamic modification patterns

These observers chain the notification - insertion, update or removal of an element - to subsequent implementations that form the backend for the query syntax. Figure 3.5 gives an overview on these classes. While the static class `OptimizationExtensions` includes the necessary extension methods for the observable source monad, the classes `CompositeSource`, `SelectManySource`, `SelectSource` and `FilteredSource` form the backend for the monadic implementation of the keywords `from`, `where` and `select`. Grouping remians unimplemplemented, so far. With these classes, it is possible to specify dynamic modification patterns with the query syntax of C# (or any other host language, respectively).

In the case of the class diagram restructuring case from the TTC, see section 2, there is only one type of modifications, namely to pull common attributes to a common base class. We can represent this modification with the class *PullUpCommonAttributes*. This modifications now needs a pattern to let NMF OPTIMIZATIONS know when it is applicable. Listing 3.2 shows an implementation for a modification pattern that is triggers when there are any two class that share an attribute and can be merged. This unifies the three rules given in the case description.

Listing 3.2: A modification pattern when it is applicable to pull up common attributes

```
1 AddPattern(from c1 in Entities
2  from c2 in Entities
3  where c1 != c2
4  && c1.Generalization.Select(g => g.General).FirstOrDefault() ==
      c2.Generalization.Select(g => g.General).FirstOrDefault()
5  from a1 in c1.OwnedAttribute
6  from a2 in c2.OwnedAttribute
7  where a1.Name == a2.Name && a1.Type == a2.Type
8  select new PullUpCommonAttributes(c1.Generalization.Select(g =>
      g.General).FirstOrDefault(), a1, this));
```

In listing 3.2, the first two lines create tuples of any two entities labeled as c_1 and c_2. Line 3 requires c_1 and c_2 to be different and line 4 adds another condition that they must share the same base class or both not have a base class at all. Line 5 and 6 select the attributes of the entities c_1 and c_2 and finally line 7 restricts these tuples that the names and types of the properties must match. Note that this pattern specifies every occurrence of restructuring rules as in [LR13] all at once. This has been done to avoid duplicate code. Line 8 specifies that in this case, the *PullUpCommonAttributes* modification should be applied to the model with the given parameters.

Because *Entities* is an observable source, NMF OPTIMIZATIONS can recognize changes to this collection and propagate changes to this collection that cause to create or delete tuples and eventually create or delete the appropriate modifications. Because it is used in the context with an observable source, the compiler can also interfere that *OwnedAttribute* collection of an *Entity* should be treated as observable source. NMF OPTIMIZATIONS can receive updates of this collection, because it implements the interface `INotifyCollectionChanged`. However, syntax restrictions in C# make it impossible to check that during compile time. Thus, if the class representing the *OwnedAttribute* collection of an *Entity* did not implement the *INotifyCollectionChanged* interface, NMF OPTIMIZATIONSwould not receive any updates, but would not throw an exception, neither at runtime nor at compile time.

The comiler will convert the above listing into the sequence of extension methods shown in listing 3.3. These extension methods use the classes from figure 3.5 inside their implementation.

Listing 3.3: The above modification as sequence of extension methods

```
Entities.SelectMany(c1 => Entities, (c1, c2) => new { c1 = c1, c2
    = c2 })
```

```
        .Where((c1,c2) => c1 != c2
&& c1.Generalization.Select(g => g.General).FirstOrDefault() == c2.Generalization.Select(g => g.General).FirstOrDefault())
        .SelectMany((c1,c2) => c1.OwnedAttributes, (c1,c2,a1) => new {c1=c1, c2=c2, a1=a1})
        .SelectMany((c1,c2,a1) => c2.OwnedAttributes, (c1,c2,a1,a2) => new {c1=c1,a1=a1,a2=a2})
        .Where((c1,a1,a2) => a1.Name == a2.Name && a1.Type == a2.Type)
        .Select((c1,a1,a2) => new PullUpCommonAttributes(c1.Generalization.Select(g => g.General).FirstOrDefault(), a1, this));
```

However, the pattern in listing 3.2 is quite long. It is thus a good idea to divide into multiple parts. An approach could be to separate the creation of tuples of entities, e.g. because they are also needed for another pattern. In this way, duplicate code can be avoided.

Listing 3.4: Separating the creation of tuples

```
var tuples = from c1 in Entities
 from c2 in Entities
 where c1.Element != c2.Element
 select new { Entity1 = c1, Entity2 = c2 };
```

An example of this procedure is shown in section in listing 3.4 where the creation of tuples has been set saved into a dedicated variable. The anonymous class language feature of C# allows the syntax to create an instance of an anonymous class with properties *Entity1* and *Entity2*. Although this anonymous class is invisible in figure 3.2, the compiler will also create this anonymous class in the above case, although it is completely hidden from the developer. These tuples can now be used to specify the modification patterns in a slightly modified version of the code from listing 3.2, see listing 3.5.

Listing 3.5: The above modification pattern reusing the tuples

```
AddPattern(from tuple in tuples
 where tuple.Entity1.Generalization.Select(g => g.General).FirstOrDefault() == tuple.Entity2.Generalization.Select(g => g.General).FirstOrDefault()
 from a1 in tuple.Entity1.OwnedAttribute
 from a2 in tuple.Entity2.OwnedAttribute
 where a1.Name == a2.Name && a1.Type == a2.Type
 select new PullUpCommonAttributes(tuple.Entity1.Generalization.Select(g => g.General).FirstOrDefault(), a1, this));
```

However, this version of the mofification pattern is inefficient, as it creates 2-tuples of all 2-tuples of entities. Given a large set of entities, this is going to be expensive. A better idea would be to separate the cases where a common base entity exists. This can be easily achieved by using the modification pattern presented in listing 3.6.

Listing 3.6: A modification pattern for entities with a base entity

```
AddPattern(from c in Entities
 where c.Generalization.Count > 0
 from a in c.OwnedAttribute
 where IsCandidate(c.Generalization[0].General, a.Name, a.Type)
 select new PullUpCommonAttributes(c.Generalization[0].General, a, this));
```

3.4.2.2. The Update mechanism

It is crucial for dynamic modification patterns to understand how they are dynamic and how changes within the chains of such observable sources are propagated through these patterns. In this subsection, the update mechanism is explained for all `ObservableSourceBase` implementations.

In this section, we will treat calling the observers in the same way as calling an event. The explicit observers have been chosen over events mainly for performance reasons.

CompositeSource, SelectManySource

These source types describe tuples of elements from other collections and thus implement the `from` keyword in the C# query syntax. Whenever an item of a subsequent observable collection is added, updated or removed, all tuples within the composite source are added, updated or removed, respectively.

SelectSource

This source simply selects a feature from a observable source of objects. When an element is added to the underlying observable source, the selection for this element is created, saved and the `SelectSource` fires an event that a new element has been added to the collection. When an element is removed, the `SelectSource` finds the appropriate selection and fires an event that the selected object is removed. The update process is more complex. If an element is updated, the `SelectSource` compares the old and the new representation for this element. In case the object references are equal, the element is updated to be sure that no update information is lost. Otherwise, the old representation is deleted (with the appropriate event) and the new representation is added. Thus, an update process is turned into a deletion and an insertion.

FilteredSource

The `FilteredSource` works similar to the `SelectSource`. If an element is added to the underlying observable source, the `FilteredSource` checks whether the element fulfills the filter condition. If so, the element is added to a whitelist and an event is raised that the `FilteredSource` contains a new element. Otherwise, the element is ignored. Likewise, if an element is deleted in the underlying observable source and the elemnt is not in the whitelist, this is ignored. If the element is on the whitelist, the element is removed from the whitelist and an event is raised to inform clients that the element has been removed. In case where an element in the underlying observable source is updated, it is important whether the element is in the whitelist (i.e. previously satisfied the filter condition) and whether it currently satisfies the filter condition. There are four cases:

1. The element is in the whitelist and also fulfills the filter condition. In this case, the update is propagated to the client of the `FilteredSource`.
2. The element is in the whitelist, but does not fulfill the filter condition anymore. In this case, the element is removed from the whitelist and an event is raised that the element is removed from the `FilteredSource`.

3. The element is not in the whitelist, but fulfills the filter condition. In this case, the element was previously ignored by the filter source. Thus, the element is added to the whitelist and an event is raised to inform clients.
4. The element is not in the whitelist and does not fulfill the filter condition. No action is to be taken in this situation.

3.5. Optimizer implementations

This chapter introduces the three optimizers that are implemented in NMF OPTIMIZATIONS. These include the Brute-Force Optimizer that is presented in section 3.5.1, the Greedy Optimizer introduced in section 3.5.2 and the Impact-sensitive Optimizer presented in section 3.5.3.

While both the Brute-Force Optimizer and the Greedy Optimizer ignore the impact of a modification, the impact-sensitive optimizer does not and thus requires the modifications used within an optimization to specify proper impacts. However, none of these optimizers uses the difference of static and dynamic modification patterns and thus both treat them equally as dynamic modification patterns. Thus, there is a big unused potential to speed up the optimizations. However, as this thesis is about the maintainability, these potential performance gains will be ignored here, as they are out of the scope of this thesis.

However, the optimizer components presented in this chapter do not implement truly sophisticatzed algorithms. Instead, they are rather simple. As a reason, the purpose of NMF OPTIMIZATIONS was to create a framework to represent optimization tasks. Thus, the optimizer implementations within the framework can rather be seen as a proof of concept for the representation of optimization tasks, rather than sophisticated implementations of optimizers.

As the execution semantics of optimizations written in NMF OPTIMIZATIONS is hidden in the optimizer implementations, the implementation to the optimizers included in the NMF OPTIMIZATIONS framework are presented here in detail.

3.5.1. Brute-Force Optimizer

As the name implies, the brute-force optimizer applies the brute force algorithm to solve the optimization. The advantage of the Brute Force optimizer is that it does not require any assumptions. Especially, the impact of the optimization modifications is not required. Thus, the Brute-Force optimizer is an alternative when it is difficult or impossible to compute the impact of a modification.

The implementation of the Brute-Force Optimizer is presented in listing 3.7, where the method to process a scenario is shown (the parameters types are omitted). This method is called by the `Optimize` method with some initialization and afterwards, the best scenario is applied to the model. The optimizer selects the distinct set of available optimization modifications first. Afterwards, the scenario evaluator evaluates the current scenario and possible set to the new best scenario. If, however, the optimization is marked to force modifications and there is a modification available, this step is omitted. Afterwards, the optimizer applies these modifications to the model and searches for new scenarios in a depth-first process.

Listing 3.7: The implementation of the Brute-Force Optimizer

```
private void ProcessScenario(currentScenario, optimization,
    evaluator, ref bestScenario)
{
 var mods = optimization.GetCurrentModifications().Distinct().
    ToArray();
 if (!optimization.ForceModifications || mods.Length == 0)
 {
  currentScenario.Cost = evaluator.EvaluateScenario(
     currentScenario);
  if (currentScenario.Cost < bestScenario.Cost)
  {
   bestScenario = currentScenario;
  }
 }
 foreach (var mod in mods)
 {
  var scenario = currentScenario.Fork(mod);
  mod.Apply();
  FillAlternatives(scenario, optimization, evaluator, ref
     bestScenario);
  mod.Reverse();
 }
}
```

However, as the Brute-Force optimizer utilizes brute force, the execution time is exponential.

3.5.2. Greedy Optimizer

Since the Brute-Force Optimizer has an exponential execution time, it is not suitable in various occasions. A possible solution is to use approximative algorithms. A very simple approximative algorithm is the greedy algorithm. The greedy optimizer always checks which of the available modifications yields the gain in the cost function and applies it. If no modification can beat the current scenario, the curent scenario is returned as the best scenario. In this way, the greedy optimizer somehow finds a ”local maximum” where locality is expressed using the distance implied by the modification appliances. The execution time of the greedy optimizer is roughly the amount of applied modifications in the optimal scenario multiplied by the total amount of available modifications in the optimization, which is quadratic in the total amount of modifications in the optimization.

Listing 3.8: The implementation of the Greedy Optimizer

```
private static IOptimizationScenario GreedyExploreScenario(
   currentScenario, optimization, evaluator)
{
 double lowestCost = optimization.ForceModifications ? double.
    PositiveInfinity : currentScenario.Cost;
 IOptimizationScenario bestScenario = currentScenario;
 IOptimizationModification bestMod = null;

 var mods = optimization.GetCurrentModifications().Distinct().
    ToArray();
 foreach (var modification in mods)
 {
  var modifiedScenario = currentScenario.Fork(modification);
  modification.Apply();
```

```
    var cost = evaluator.EvaluateScenario(modifiedScenario);
    modifiedScenario.Cost = cost;
    if (cost < lowestCost)
    {
      lowestCost = cost;
      bestScenario = modifiedScenario;
      bestMod = modification;
    }
    modification.Reverse();
  }
  if (bestScenario != currentScenario)
  {
    bestMod.Apply();
    return GreedyExploreScenario(bestScenario, optimization,
        evaluator);
  }
  else
  {
    return currentScenario;
  }
}
```

The implementation of the greedy optimizer is presented in listing 3.8. The algorithm applies every available modification and looks for the best improvement that can be achieved applying a single modification. If no modification can achieve an improvement, the optimizer returns the initial scenario. The modification that achieved the best improvement is applied to the optimization model and the optimizer continues to process this scenario. The `Optimize` method just does some checks on the arguments and returns the result of the method shown above.

3.5.3. Impact-Sensitive Optimizer

Of the three implementation for optimizers included in NMF Optimizations, the impact-sensitive optimizer is the most sophisticated one. Unlike the other two, it uses the information on the impact of modifications to decide when to apply an optimization modification.

The basic idea behind the impact-sensitive optimizer is the assumption that the order in which modifications are applied does not matter in case the impact of these modifications is disjunct. If two modifications A and B have disjunct impacts, then B will still be available if A has been applied to the model and vice versa. Furthermore, the modifications available after applying the sequence A, B will be the same as after B, A. Thus, It is not necessary to explore both sequences in depth and hence one exploration step can be pruned. Furthermore, if the optimization requires forced modification, the impact-sensitive optimizer makes the assumption that both A and B must be applied together, unless there is another modification C that also has an impact disjunct to A, but interferes with B (or vice-versa). In this case, the algorithm also considers the combination A before C. However, as the algorithm is agnostic of the optimization domain, it might be possible that after applying A and C, the modification B is still available, as the application of C does not necessarily prevents the application of B.

If one considered not just two but in general $n \in \mathbb{N}$ modifications with disjunct impact, such procedure for forced modifications cuts down the effort spent to an almost linear complexity.

Listing 3.9: The implementation of the impact-sensitive optimizer

```
1  public void ProcessScenario(currentScenario, forbidden)
2  {
3   var mods = CreateInitialModificationGroups(forbidden);
4   if (!Optimization.ForceModifications || mods.Count == 0)
5   {
6    TestAndSetScenario(currentScenario);
7   }
8   ExpandModificationGroups(mods);
9   ApplyGroups(currentScenario, forbidden, mods);
10 }
```

The implementation of the Impact-Sensitive Optimizer is presented in listing 3.9. The `ProcessScenario`-method is run inside a dedicated optimization context object, which for example contains the property `Optimization` from line 4. Unlike the `ProcessScenario` method, it also contains a collection of forbidden modifications that must not be used in this step. The algorithm consists of four steps. In the first steps, the algorithm creates an initial list of modification groups that represent the available modifications of the optimization in groups of one. Next, the current scenario is evaluated and set as new best scenario, similar to the Brute-Force Optimizer from section 3.5.1. The step introducing the biggest difference is the step to expand the modification groups. After the expansion, the groups are applied to the model.

Listing 3.10: Expanding modification groups

```
protected virtual List<ModificationGroup>
    CreateInitialModificationGroups(IEnumerable<
    IOptimizationModification> forbidden)
{
 var hashMods = new HashSet<IOptimizationModification>(
     Optimization.GetCurrentModifications());
 hashMods.ExceptWith(forbidden);

 var mods = new List<ModificationGroup>(hashMods.Count);
 var counter = 0;
 foreach (var item in hashMods)
 {
  var box = new SingleModification(item);
  box.Index = counter;
  counter++;
 }
 return mods;
}
```

The implementation of the first step is shown in listing 3.10. Basically, the available modifications are inserted into a hash set. This hash set deletes possible duplicate modification entries in an efficient way. Furthermore, it allows to remove forbidden modifications efficiently. Next, the remaining modifications are wrapped in a `ModificationGroup` and put into a separate list of modification groups. A class diagram of modification groups is shown in figure 3.6. A `SingleModification` represents a group of just a single modification, whereas a `CompositeModificationGroup` represents a composite group consisting of two other modification groups. The `SingleModificatio` is assigned their index within the list of modification groups for easier reference.

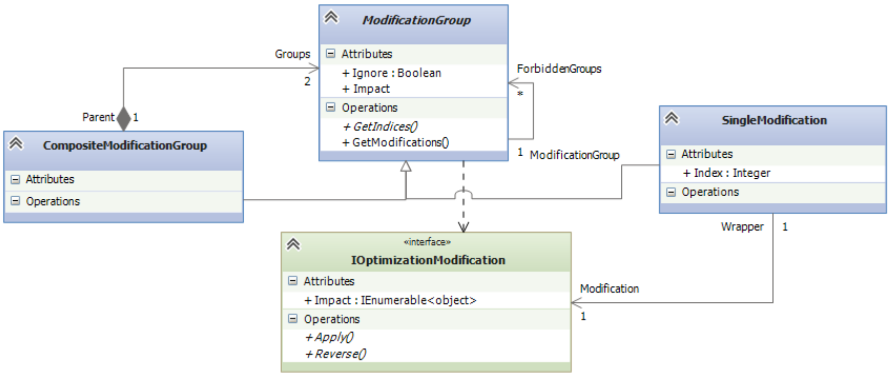

Figure 3.6.: Modification groups used in the Impact-Sensitive Optimizer

In the expansion of the modification groups, the modification groups are merged into larger groups, provided they have a disjunct impact. The impact of a modification group is just the conjunction of the impacts of the modifications they are representing. To allow efficient processing, these impact sets are represented using hash sets.

The implementation of this expansion step is shown in listing ch:nmfo:impactSensitive:expandGroups.

Listing 3.11: Expanding modification groups

```
protected virtual void ExpandModificationGroups(List<ModificationGroup> mods)
{
 var simples = mods.Count;
 for (int i = 1; i < mods.Count; i++)
 {
  for (int j = 0; j < i && j < simples; j++)
  {
   var complex = mods[i];
   var simple = mods[j] as SingleModification;
   if (!complex.Impact.IntersectsWith(simple.Impact))
   {
    if (!Optimization.ForceModifications)
    {
     complex.Forbidden.Add(simple);
     simple.Forbidden.Add(complex);
    }
    else
    {
     complex.Ignore = true;
     simple.Ignore = true;
    }
    if (simple.Index < complex.GetIndices().First())
    {
     var compound = new CompositeModificationGroup(simple, complex);
     mods.Add(compound);
    }
   }
  }
 }
}
```

```
}
```

The algorithm creates tuples of modification groups and single modifications that reside in the initial list. A composite is added to the modification group list, if two modification groups have disjunct impacts. This is guarded with the restriction that the index of the simple modification is less than the smallest index within the composite. This condition is used to ensure that each combination of modifications is only added to modification group list once at most. Furthermore, if the impacts of `simple` and `complex` are disjunct, then there will definitely be a further modification group that consists of all modifications of `simple` and `complex`. Thus, when applying these modification groups, it is not allowed to use the modifications of the other modification group in order to avoid duplicate processing of these modifications.

The implementation of the other two steps is omitted here, but can be reviewed in the online source code repository at the CodePlex project homepage[1].

3.6. Testing

Very similar to model transformations, (model) optimizations contain a lot of possibilities where developer can make mistakes. Thus, it is important that model optimizations are properly tested.

Furthermore, testing optimizations has the same problems regarding black box tests. Despite their usefulness, it is usually better to also test bits of the optimizations separately.

Optimizations consist of modification patterns that in turn call optimization modifications. As the classes involved in specifying modification patterns are a part of NMF OPTIMIZATIONS, they already are tested. However, optimization developers can still make mistakes when specifying modification patterns, especially dynamic modification patterns. Sadly, it is complicated to test such modification patterns separately. An approach could be to modify the model in the testing environment after the optimization has been initialized and is thus observing the model and afterwards asserting the available modifications of the optimization.

Unlike modification patterns, optimization modifications can be tested very easily. Being classes, testing such optimization modifications is as easy as testing the `Apply` and `Reverse` method and the impact property, if the impact is used at all. This can be achieved by any testing framework. As the `Apply` and `Reverse` methods do not have parameters at all, setting up the test environment entirely depends on the properties of the modification class, which in turn is created by the optimization developer and thus, the optimization developer is responsible that testing the modifications is possible just like with any other piece of code.

3.7. Conclusions

Wrapping up, NMF OPTIMIZATIONS provides a way to represent optimization tasks inside a framework. Thus, optimizations can be written in a declarative way, by just specifying what needs to be optimized rather than specifying how this optimizaion task is to be solved. The question how it is to solved can just be answered by reusing existing optimizer implementations.

However, so far, there are only few optimizer implementations available, but this is a clear point of improvement and sophisticated algorithms can be implemented in such optimizers, making this implementation available to a wide range of optimization tasks.

---

[1] `http://nmf.codeplex.com/`

Bibliography

[FB99] M. Fowler and K. Beck, *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999.

[Fow10] M. Fowler, *Domain-specific languages.* Addison-Wesley Professional, 2010.

[Hin13] G. Hinkel, "An approach to maintainable model transformations using internal DSLs," 2013.

[Hor13] T. Horn, "Solving the Class Diagram Restructuring Transformation Case with FunnyQT," in *Sixth Transformation Tool Contest (TTC 2013)*, ser. EPTCS, vol. to appear, 2013.

[LR13] K. Lano and S. K. Rahimi, "Case study: class diagram restructuring," in *Sixth Transformation Tool Contest (TTC 2013)*, ser. EPTCS, vol. to appear, 2013.

[SR13] W. Smid and A. Rensink, "Class Diagram Restructuring with GROOVE," in *Sixth Transformation Tool Contest (TTC 2013)*, ser. EPTCS, vol. to appear, 2013.

List of Figures

Listings

A. Appendix

A.1. Restructuring UML class diagrams

Listing A.1 shows the complete optimization to solve the class diagram restructuring case from the TTC 2013.

Listing A.1: An NMF Optimizations solution for the class diagram restructuring case from the TTC 2013

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using My;

using NMF.Optimizations;
using NMF.Utilities;

namespace RestructureClassDiagrams
{
 public static class Facade
 {
  public static void RestructureClassDiagram(Model diagram)
  {
   var restructuring = new ClassDiagramRestructuring(diagram);
   restructuring.Optimize(GreedyOptimizer.Instance,
      BasicEvaluator.Instance);
  }
 }

 class ClassDiagramRestructuring : Optimization<Model>
 {
  public ClassDiagramRestructuring(Model model) : base(model)
  {
   Entities = new ObservableSource<Entity>(model.Entitys);

```

```
    ForceModifications = true;

    AddPattern(from entity in Entities
        where entity.Generalization.Count > 0
        from baseEntity in entity.Generalization
        from attribute in entity.OwnedAttribute
        where AtLeastTwice(baseEntity
          .General
          .Specialization
          .Select(g => g.Specific), attribute.Name, attribute.Type)
        select new PullUpCommonAttributes(entity.Generalization.
            First().General, attribute, this));

    var withoutBase = from c1 in Entities
          where c1.Generalization.Count == 0
          select c1;

    AddPattern(from entity in withoutBase
        from a in entity.OwnedAttribute
        where AtLeastTwice(withoutBase, a.Name, a.Type)
        select new PullUpCommonAttributes(null, a, this));
  }

  public class PullUpCommonAttributes : IOptimizationModification
      , IEquatable<PullUpCommonAttributes>
  {
    public Entity BaseEntity { get; private set; }
    public Property Property { get; private set; }
    public ClassDiagramRestructuring Parent { get; private set; }
    public Entity NewEntity { get; private set; }

    private List<Entity> Candidates { get; set; }

    public PullUpCommonAttributes(Entity baseEntity, Property
        property, ClassDiagramRestructuring parent)
    {
      if (parent == null) throw new ArgumentNullException("parent
          ");
      if (property == null) throw new ArgumentNullException("
          property");

      Parent = parent;
      Property = new Property() { Name = property.Name, Type =
          property.Type };
      BaseEntity = baseEntity;
    }

    public IEnumerable<Entity> GetCandidates()
    {
      IEnumerable<Entity> initialCandidates;
      if (BaseEntity == null)
      {
```

```
     initialCandidates = Parent.Model.Entitys.Where(box => box.Generalization.Count == 0);
    }
    else
    {
     initialCandidates = BaseEntity.Specialization.Select(gen => gen.Specific);
    }

    return initialCandidates
     .Where(e => e.OwnedAttribute.Exists(p => p.Name == Property.Name && p.Type == Property.Type));
  }

  public bool Apply()
  {
    Candidates = GetCandidates().ToList();

    if (Candidates.Count <= 1)
    {
     return false;
    }

    foreach (var candidate in Candidates)
    {
     RemovePropertyFrom(candidate);
    }

    if (BaseEntity != null && Candidates.Count() == BaseEntity.Specialization.Count)
    {
     BaseEntity.OwnedAttribute.Add(Property);
     Parent.Model.Propertys.Add(Property);
    }
    else
    {
     if (NewEntity == null) NewEntity = new Entity() { Name = Parent.GetNewClassName() };
     NewEntity.OwnedAttribute.Add(Property);
     Parent.Model.Propertys.Add(Property);
     if (BaseEntity != null)
     {
      Parent.Model.Generalizations.Add(new Generalization() { General = BaseEntity, Specific = NewEntity });
      foreach (var candidate in Candidates)
      {
        ChangeBaseEntity(candidate, BaseEntity, NewEntity);
      }
     }
     else
     {
      foreach (var derived in Candidates)
```

```
        {
          var g = new Generalization()
          {
          General = NewEntity,
          Specific = derived
          };
          Parent.Model.Generalizations.Add(g);
        }
       }
       foreach (var candidate in Candidates)
       {
        Parent.Entities.Refresh(candidate);
       }
       Parent.Entities.Publish(NewEntity);
      }

      return true;
    }

    private void ChangeBaseEntity(Entity candidate, Entity oldBase, Entity newBase)
    {
      var generalization = candidate.Generalization.FirstOrDefault(g => g.General == oldBase);
      candidate.Generalization.Remove(generalization);
      generalization.General = null;
      Parent.Model.Generalizations.Remove(generalization);
      Parent.Model.Generalizations.Add(new Generalization() { General = newBase, Specific = candidate });
    }

    private void RemovePropertyFrom(Entity entity)
    {
      var prop = entity.OwnedAttribute.First(p => p.Name == Property.Name && p.Type == Property.Type);
      entity.OwnedAttribute.Remove(prop);
      Parent.Model.Propertys.Remove(prop);
    }

    public void Reverse()
    {
      if (Candidates.Count <= 1) return;

      if (NewEntity == null)
      {
       RemovePropertyFrom(BaseEntity);
       foreach (var candidate in Candidates)
       {
        AddPropertyClone(candidate);
       }
      }
      else
      {
       RemovePropertyFrom(NewEntity);
       var g = NewEntity.Generalization.FirstOrDefault();
       if (g != null)
       {
        g.General = null;
        g.Specific = null;
        Parent.Model.Generalizations.Remove(g);
        foreach (var derived in Candidates)
        {
          ChangeBaseEntity(derived, NewEntity, BaseEntity);
          Parent.Entities.Refresh(derived);
          AddPropertyClone(derived);
        }
       }
       else
       {
        foreach (var derived in Candidates)
        {
          var generalization = derived.Generalization
          .Where(gen => gen.General == NewEntity).First();

          generalization.General = null;
          generalization.Specific = null;
          Parent.Model.Generalizations.Remove(generalization);

          Parent.Entities.Refresh(derived);
          AddPropertyClone(derived);
        }
       }
       Parent.Entities.Revoke(NewEntity);
      }
    }

    private void AddPropertyClone(Entity candidate)
    {
      var propertyClone = new Property()
      {
       Name = Property.Name,
       Type = Property.Type
      };
      candidate.OwnedAttribute.Add(propertyClone);
      Parent.Model.Propertys.Add(propertyClone);
    }

    public IEnumerable<object> Impact
    {
      get
      {
       if (BaseEntity != null) yield return BaseEntity;
       foreach (var candidate in GetCandidates())
       {
```

```
       yield return candidate;
      }
     }
   }

   public override bool Equals(object obj)
   {
     return Equals(obj as PullUpCommonAttributes);
   }

   public override int GetHashCode()
   {
     var i = 0;
     if (BaseEntity != null) i ^= BaseEntity.GetHashCode();
     i ^= Property.Name.GetHashCode();
     i ^= Property.Type.GetHashCode();
     return i;
   }

   public bool Equals(PullUpCommonAttributes other)
   {
     if (other == null) return false;
     return other.BaseEntity == BaseEntity && other.Property.Name == Property.Name && other.Property.Type == Property.Type;
   }
  }

  private bool AtLeastTwice(IEnumerable<Entity> entities, string name, My.Type type)
  {
   return entities
     .Where(c => c.OwnedAttribute
      .Exists(p => p.Name == name && p.Type == type))
     .Count() > 1;
  }

  public ObservableSource<Entity> Entities { get; set; }

  private int classCounter = 0;
  internal string GetNewClassName()
  {
   classCounter++;
   return ”NewClass” + classCounter.ToString();
  }
 }
}
```

A.2. NMF

This section briefly introduces NMF, an open-source project to provide support for model-driven techniques on the .NET platform. The abbreviation NMFstands for .NET Modeling

Framework. It is an open-source framework initiated in July 2012 and now hosted on Codeplex at `http://nmf.codeplex.com/`. The following sections explain the purpose of the projects that NMFconsists of.

- TRANSFORMATIONS for M2M transformations presented in section A.2.1 or [Hin13]
- OPTIMIZATIONS for domain-specific optimizations presented in chapter 3
- ECOREINTEROP for EMF interopability presented in section A.2.3
- SERIALIZATIONS for XMI serialization presented in section A.2.4
- ANYTEXT, a parser based on language descriptions similar to XTEXT. However, this project retired.
- Several others projects, including projects for collections and utilities

A.2.1. Transformations

The TRANSFORMATIONS project exists since July 2012. It has been subject of various refactorings and extensions in the master thesis [Hin13] and is also first described in this thesis. The project consists of a core framework (NMF TRANSFORMATIONS CORE) and an internal DSL built on top of it, NTL. This internal DSL is designed to be used with C# as host language and enables transformation developers to write rule-based M2M transformations in C# code.

A.2.2. Optimizations

The OPTIMIZATIONS project is the project that this technical report is based on. It is described in detail in chapter 3.

A.2.3. EcoreInterop

The ECOREINTEROP is a project to provide interopability support with EMF, especially the Ecore meta-metamodel. In this way, it provides a code generator that can create classes representing the metaclasses of an Ecore-metamodel. These classes are further decorated with attributes that enable the SERIALIZATIONS project to load and save these models from and to XMI. This interopability was needed to participate in the TTC2013, as the resulting models were in the Ecore XMI serialization format. The code generation for Ecore packages is an important use case of NMF TRANSFORMATIONS, as it is a model transformation that targets the metamodel defined in the `System.CodeDOM` namespace, much like the ABB case study from chapter **??**.

A.2.4. Serializations

The SERIALIZATIONS project is a framework to easily serialize objects from and to XML. Unlike the XML-serializer that is contained in the .NET framework, the XML-serializer from the SERIALIZATIONS project is capable of serializing models with cyclic references. However, it is still possible to influence the format of the resulting XML files. Furthermore, the XML document is not kept in memory, but the model is created on-the-fly.

Compared to the EMF builtin serializer, this serializer has some drawbacks, as it is not capable to load models split among multiple files. Neither is it possible to serialize a model to multiple files.